Encyclopaedia of Classes

Volume 3

VERSION 2020.0.02

Contents

Before You Begin

The *JADE Encyclopaedia of Classes* is intended as a major source of information when you are developing or maintaining JADE applications.

Who Should Read this Encyclopaedia

The main audience for the *JADE Encyclopaedia of Classes* is expected to be developers of JADE application software products.

What's Included in this Encyclopaedia

The *JADE Encyclopaedia of Classes* has two chapters, and is divided into three volumes.

Chapter 1	Gives a reference to system classes, and the constants, properties, and methods that they provide
Chapter 2	Gives a reference to **Window** classes, and the constants, properties, methods, and events that they provide

Note that this third volume contains Chapter 2 only. Chapter 1 is divided into two volumes: Volume 1 (that is, **EncycloSys1.pdf**) contains system (non-GUI) classes in the range **ActiveXAutomation** class through **JadeSkin** class, inclusive, and Volume 2 (that is, **EncycloSys2.pdf**) contains system (non-GUI) classes in the range **JadeSkinApplication** class through **WebSession** class, inclusive.

Related Documentation

Other documents that are referred to in this encyclopaedia, or that may be helpful, are listed in the following table, with an indication of the JADE operation or tasks to which they relate.

Title	Related to…
JADE Database Administration Guide	Administering JADE databases
JADE Development Environment Administration Guide	Administering JADE development environments
JADE Development Environment User’s Guide	Using the JADE development environment
JADE Encyclopaedia of Primitive Types	Primitive types and global constants
JADE Installation and Configuration Guide	Installing and configuring JADE
JADE Initialization File Reference	Maintaining JADE initialization file parameter values
JADE Object Manager Guide	JADE Object Manager administration
JADE Platform Differences Guide	Platform differences when running JADE applications
JADE Synchronized Database Service (SDS) Administration Guide	Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)

Conventions

The *JADE Encyclopaedia of Classes* uses consistent typographic conventions throughout.

Convention	Description
Arrow bullet (»)	Step-by-step procedures. You can complete procedural instructions by using either the mouse or the keyboard.
Bold	Items that must be typed exactly as shown. For example, if instructed to type **foreach**, type all the bold characters exactly as they are printed.
	File, class, primitive type, method, and property names, menu commands, and dialog controls are also shown in bold type, as well as literal values stored, tested for, and sent by JADE instructions.
Italic	Parameter values or placeholders for information that must be provided; for example, if instructed to enter *class-name,* type the actual name of the class instead of the word or words shown in italic type.
	Italic type also signals a new term. An explanation accompanies the italicized type.
	Document titles and status and error messages are also shown in italic type.
Blue text	Enables you to click anywhere on the cross-reference text (the cursor symbol changes from an open hand to a hand with the index finger extended) to take you straight to that topic. For example, click on the "**borderStyle"** cross-reference to display that topic.
Bracket symbols ([ ])	Indicate optional items.
Vertical bar (|)	Separates alternative items.
`Monospaced font`	Syntax, code examples, and error and status message text.
ALL CAPITALS	Directory names, commands, and acronyms.
Small font	Keyboard shortcut keys.

Key combinations and key sequences appear as follows.

Convention	Description
Key1+Key2	Press and hold down the first key and then press the second key. For example, "press Shift+F2" means to press and hold down the Shift key and press the F2 key. Then release both keys.
Key1,Key2	Press and release the first key, then press and release the second key. For example, "press Alt+F,X" means to hold down the Alt key, press the F key, and then release both keys before pressing and releasing the X key.

Chapter 2 Window Classes

This chapter covers the following topics.

    - Ocx Class
    - OleControl Class
    - OptionButton Class
    - Picture Class
        - JadeMask Class
    - ScrollBar Class
    - StatusLine Class
    - Table Class
    - TextBox Class
        - WebHTML Class
- Window, Form, and Control Properties
- Window, Form, and Control Methods
- Window, Form, and Control Events
- Graphics Properties and Methods

For details about JADE system (non-GUI) classes, see Chapter 1, "System Classes", in Volume 1 and Volume 2.

Window Class

The **Window** class is the abstract superclass of all **Form** and **Control** classes. The **Window** class provides properties and methods that apply to all forms and controls; for example, all graphical properties and methods are defined at the **Window** class level and therefore apply to any form or control. You cannot add a subclass to the **Window** class.

The **backColor** and **borderStyle** properties may have no effect in certain controls but are defined for internal reasons; for example, showing three-dimensional effects. For example, the **borderStyle** property has no meaning for a scroll bar control, as that control always has a border.

A local variable can be described as a window in logic and access all properties and methods of the **Window** class.

Note **Window**, **Form**, and **Control** methods and events cannot be invoked from a server method.

JADE supports mouse wheel requests, as follows.

1. If the window under the mouse has a vertical scroll bar, the mouse wheel request is treated the same as a scroll bar line up or line down, depending on the way that the wheel is turned (that is, the scroll wheel is treated the same as clicking on the up or down arrow of the scroll bar).

2. If the window under the mouse has no vertical scroll bar but has a horizontal scroll bar, the mouse wheel request is treated the same as a scroll bar line left or line right, depending on the way that the wheel is turned (that is, the scroll wheel is treated the same as clicking on the left or right arrow of the scroll bar).

3. If the window under the mouse does not have scroll bars, the window that has focus or the first parent of that window with a scroll bar is scrolled.

   If the window has a vertical scroll bar, the mouse wheel request is treated the same as a scroll bar line up or line down, depending on the way that the wheel is turned. If the window (or its parent) has no vertical scroll bar but has a horizontal scroll bar, the mouse wheel request is treated the same as a scroll bar line left or line right, depending on the way that the wheel is turned.

When a control receives focus and it is not within the visible region of a container (or containers) control that has scroll bars (for example, a **BaseControl** or **Picture** control), the container is automatically scrolled to bring the control into view. The scroll events for the container (or containers) are also called for this process. The position of children within a **ListBox**, **Table**, or **TextBox** control remains absolute and is not affected by any scrolling within those controls.

The automatic scrolling of a control into view does not occur if the focus is received as result of a mouse click (that is, if the window is only partially visible, clicking on it does not cause the window to move to show more of the window).

When multiple monitors are running on one workstation, the following points apply. (For details about terminating GUI applications with no visible forms when running multiple desktops, see "Showing an Invisible Form", in Chapter 1 of the *JADE Runtime Application Guide*.)

- JADE handles only the first nine monitors running on one workstation. Additional monitors are ignored.
- The **Window::getDeskTopWorkArea** method returns the available desktop area of the primary monitor.
- When a form is created, it is created in the monitor area of the control with the current focus. If no control has focus, the form is created in the same monitor as the mouse.

If the control that has focus spans multiple monitors, the monitor containing the greater area of the control that has focus is used. If the window does not fall within the bounds of any monitor, the primary monitor is used.

By default, Microsoft Windows displays a message box on the monitor where the application last had focus.

- The **Form**::**centreWindow** method centers a non-MDI form within the monitor on which the form resides. An MDI child form continues to be centered within the client area of its parent MDI frame.
- The **Window**::**getMonitorArea** method returns the full area of the current monitor on which the window resides and the **Window**::**getMonitorWorkArea** method returns the position of the available desktop area of the monitor on which the window resides.
- When a form is saved in the JADE Painter, the values of the **left** and **top** properties are converted to be relative to the top and left positions of the primary monitor.

Note For the arrays associated with control and menu item children (for example, the **Window** class **allControlChildren** and **MenuItem** class **children** properties), the only methods that are implemented are **at** (which allows the use of square brackets to access the elements), **createIterator** (which allows logic to do a **foreach** over the array), **size**, and **size64**.

For a summary of the constants, properties, and methods defined in the **Window** class, see "Window Class Constants", "Window Properties", and "Window Methods", in the following subsections. For details about the graphics properties and methods defined in the Window class, see "Graphics Properties and Methods", later in this document.

For details of system classes and their associated constants, properties, methods, and events, see "System Classes", in Chapter 1. For details about primitive types and their associated methods, see "Primitive Types", in Chapter 1 of the *JADE Encyclopaedia of Primitive Types*.

Inherits From: Object

Inherited By: Control, Form

Window Class Constants

The constants provided by the **Window** class are listed in the following table.

Constant	Value	Constant	Value
AllowDocking_All	#7f	AllowDocking_AllHorizontal	#10
AllowDocking_AllVertical	#20	AllowDocking_AnyEdge	#f
AllowDocking_Bottom	#2	AllowDocking_Inside	#40
AllowDocking_Left	#4	AllowDocking_None	0
AllowDocking_Right	#8	AllowDocking_Top	#1
AnimateWindow_Flags_Activate	#20000	AnimateWindow_Flags_Blend	#80000
AnimateWindow_Flags_ BottomToTop	#8	AnimateWindow_Flags_Center	#10
AnimateWindow_Flags_LeftToRight	#1	AnimateWindow_Flags_RightToLeft	#2
AnimateWindow_Flags_Slide	#40000	AnimateWindow_Flags_ TopToBottom	#4

Constant	Value	Constant	Value
BackBrushStyle_Tile	0	BackBrushStyle_Stretch	1
BackBrushStyle_Center	2	BackBrushStyle_StretchProport	3
BorderStyle_Double	3	BorderStyle_None	0
BorderStyle_Single	1	BorderStyle_Sizable	2
Color_3DDkShadow	21	Color_3DFace	15
Color_3DHighlight	20	Color_3DLight	22
Color_3DShadow	16	Color_ActiverBorder	10
Color_ActiveCaption	2	Color_AppWorkspace	12
Color_Background	1	Color_BtnFace	15
Color_BtnHighlight	20	Color_BtnShadow	16
Color_BtnText	18	Color_CaptionText	9
Color_DeskTop	1	Color_GrayText	17
Color_Highlight	13	Color_HighlightText	14
Color_InactiveBorder	11	Color_InactiveCaption	3
Color_InactiveCaptionText	19	Color_InfoBk	24
Color_InfoText	23	Color_Menu	4
Color_MenuText	7	Color_Scrollbar	0
Color_Window	5	Color_WindowFrame	6
Color_WindowText	8	DragMode_Drag	1
DragMode_Drop	2	DragMode_None	0
DragOver_Continue	1	DragOver_Enter	0
DragOver_Leave	2	DrawFillStyle_4DotDiamond53	32
DrawFillStyle_4DotDiamond55	24	DrawFillStyle_4DotDiamond95	16
DrawFillStyle_4DotDiamond99	8	DrawFillStyle_8DotDiamond55	40
DrawFillStyle_8DotDiamond99	30	DrawFillStyle_AltSquares2	31
DrawFillStyle_AltSquares4	39	DrawFillStyle_Balls	55
DrawFillStyle_Checkered	25	DrawFillStyle_Cross	6
DrawFillStyle_Cross55	15	DrawFillStyle_Cross99	23
DrawFillStyle_DbleDownDiag	34	DrawFillStyle_DbleHorzLine	51
DrawFillStyle_DbleUpDiagonal	26	DrawFillStyle_DbleVertLine	43
DrawFillStyle_DiagonalCross	7	DrawFillStyle_DiagonalHatch	47
DrawFillStyle_DottedCross	22	DrawFillStyle_DownDiagonal	5
DrawFillStyle_DownDiagonal4	18	DrawFillStyle_DownRectangle	29
DrawFillStyle_EveryOther	9	DrawFillStyle_FilledDiamond	17

Constant	Value	Constant	Value
DrawFillStyle_HalfDownDiagonal	20	DrawFillStyle_HalfUpDiagonal	12
DrawFillStyle_HorzDash	28	DrawFillStyle_HorzLine	2
DrawFillStyle_HorzLine2	35	DrawFillStyle_HorzLine4	19
DrawFillStyle_HorzRectangle	37	DrawFillStyle_HorzWaves3	21
DrawFillStyle_HorzWaves4	13	DrawFillStyle_Interlocked	45
DrawFillStyle_InvertedCross	48	DrawFillStyle_PatchworkSquares	52
DrawFillStyle_Rev4DotDiamond55	33	DrawFillStyle_Rev4DotDiamond95	41
DrawFillStyle_Rev4DotDiamond99	49	DrawFillStyle_ReverseHorzDash	46
DrawFillStyle_ShinyBalls	54	DrawFillStyle_Solid	0
DrawFillStyle_Speckled	44	DrawFillStyle_Tartan	53
DrawFillStyle_Transparent	1	DrawFillStyle_Triangles	14
DrawFillStyle_TripleDownDiag	42	DrawFillStyle_TripleUpDiagonal	50
DrawFillStyle_UpDiagonal	4	DrawFillStyle_UpDiagonal4	10
DrawFillStyle_UpKeyShape	38	DrawFillStyle_VertDash	36
DrawFillStyle_VertLine	3	DrawFillStyle_VertLine2	27
DrawFillStyle_VertLine4	11	DrawGrid_Crosses	1
DrawGrid_Dots	2	DrawGrid_Lines	0
DrawMode_Black	1	DrawMode_Copy	13
DrawMode_Invert	6	DrawMode_MaskNotPen	3
DrawMode_MaskPen	9	DrawMode_MaskPenNot	5
DrawMode_MergeNotPen	12	DrawMode_MergePen	15
DrawMode_MergePenNot	14	DrawMode_Nop	11
DrawMode_NotCopyPen	4	DrawMode_NotMaskPen	8
DrawMode_NotMergePen	2	DrawMode_NotXorPen	10
DrawMode_White	16	DrawMode_Xor	7
DrawStyle_Dash	1	DrawStyle_DashDot	3
DrawStyle_DashDotDot	4	DrawStyle_Dot	2
DrawStyle_InsideSolid	6	DrawStyle_Solid	0
DrawStyle_Transparent	5	DrawTextAlign_Center	2
DrawTextAlign_Left	0	DrawTextAlign_Right	1
KeyState_Alt	#4	KeyState_Ctrl	#2
KeyState_Shift	#1	MouseButton_Left	1
MouseButton_Middle	3	MouseButton_None	0
MouseButton_Right	2	MousePointer_Arrow	1

Constant	Value	Constant	Value
MousePointer_Cross	2	MousePointer_Cursor	4
MousePointer_Default	0	MousePointer_Drag	13
MousePointer_HandPointing	16	MousePointer_HorizontalLine	14
MousePointer_HourGlass	11	MousePointer_IBeam	3
MousePointer_NESW	6	MousePointer_NS	7
MousePointer_NWSE	8	MousePointer_NoDrop	12
MousePointer_Size	5	MousePointer_UpArrow	10
MousePointer_VerticalLine	15	MousePointer_WE	9
PictureType_Bitmap	1	PictureType_Cursor	5
PictureType_Gif	9	PictureType_Icon	3
PictureType_Jpeg	7	PictureType_Jpeg2000	10
PictureType_MetaFile	4	PictureType_None	0
PictureType_Png	8	PictureType_Tiff	6
RegisterKeys_Alt	#80000000	RegisterKeys_Ctrl	#40000000
RegisterKeys_Shift	#20000000	SM_DBCSEnabled	42
SM_Debug	22	SM_MenuDropAlignment	40
SM_MousePresent	19	SM_PenWindows	41
SM_SwapButton	23	SM_cxBorder	5
SM_cxCursor	13	SM_cxDlgFrame	7
SM_cxDoubleClk	36	SM_cxFrame	32
SM_cxFullScreen	16	SM_cxHScroll	21
SM_cxHThumb	10	SM_cxIcon	11
SM_cxIconSpacing	38	SM_cxMin	28
SM_cxMinTrack	34	SM_cxScreen	0
SM_cxSize	30	SM_cxVScroll	2
SM_cyBorder	6	SM_cyCaption	4
SM_cyCursor	14	SM_cyDlgFrame	8
SM_cyDoubleClk	37	SM_cyFrame	33
SM_cyFullScreen	17	SM_cyHScroll	3
SM_cyIcon	12	SM_cyIconSpacing	39
SM_cyKanjiWindow	18	SM_cyMenu	15
SM_cyMin	29	SM_cyMinTrack	35
SM_cyScreen	1	SM_cySize	31
SM_cyVScroll	20	SM_cyVThumb	9

Constant	Value	Constant	Value
ScaleMode_Pixels	0	ScaleMode_Point	3
ScaleMode_Twip	2	ScaleMode_User	1
ScrollBar_Horizontal	1	ScrollBars_HorzPermanentVert	7
ScrollBars_PermanentBoth	6	ScrollBars_PermanentHorizontal	4
ScrollBars_PermanentVertical	5	ScrollBars_VertPermanentHorz	8
ScrollBar_Vertical	2	ScrollBars_Both	3
ScrollBars_Horizontal	1	ScrollBars_None	0
ScrollBars_Vertical	2	WebPageColumnWidth	10
WebPageRowHeight	25		

For details about the system metrics (*SM_*) constants, see Win32 C++ online help.

Window Properties

The properties defined in the **Window** class are summarized in the following table.

Property	Description
allControlChildren	Contains an array of all controls contained in the window
backBrushStyle	Specifies the way in which the **Control** class and **Form** class **backBrush** property images are displayed
backColor	Contains the background color of a window
borderStyle	Contains the border style for a window
bubbleHelp	Contains the text that can be displayed as bubble help
controlChildren	Contains an array of the immediate children of the window
description	Contains a textual description of the object of the window
disableEvents	Specifies whether all events associated with a form or control are currently disabled
disableReason	Contains the reason the control or menu is disabled
dragCursor	Contains a specific cursor image for display during the drag process
dragMode	Contains the drag mode for a form or control
enabled	Specifies whether the object can respond to user-generated events
height	Contains the height of an object
helpContextId	Contains an associated context number for an object
helpKeyword	Contains the text used to access the help file
ignoreSkin	Specifies whether the window uses a skin
left	Contains the distance between the internal left edge of an object and the left edge of the client area of the container
mouseCursor	Contains a cursor that is not provided by the system

Property	Description
mousePointer	Contains the type of mouse pointer that is displayed
name	Contains an application, form, control, or menu item object
scaleHeight	Contains the number of units for the internal vertical measurement of an object
scaleLeft	Contains the horizontal coordinates for the left edge of an object
scaleMode	Contains the unit of measurement for coordinates of an object
scaleTop	Contains the vertical coordinates for the top edges of an object
scaleWidth	Contains the number of units for the internal horizontal measurement of an object
securityLevelEnabled	Specifies whether the form, control, or menu is automatically disabled
securityLevelVisible	Specifies whether the form, control, or menu is automatically made invisible
skinCategoryName	Contains the name of the skin category
tag	Contains a data value for the form or control
top	Contains the distance between the internal top edge of an object and the top edge of the client area
userObject	Contains an object to associate with any form or control object
userScript	Contains scripts that are included as part of the HTML generation
visible	Specifies whether an object is visible or hidden
width	Contains the width of an object

For details, see "Window, Form, and Control Properties", later in this document. See also the graphics properties defined in the **Window** class, under "Graphics Properties and Methods", later in this document.

Window Methods

The methods defined in the **Window** class are summarized in the following table.

Method	Description
aboutBox	Calls the About box for the window
addWebEventMapping	Adds functions to be invoked when a specified Web event occurs
animateWindow	Enables special effects when showing or hiding a form or control
captureMouse	Sets the mouse capture to the specified window
centreWindow	Centers the window that is being opened
clearWebEventMappings	Removes all Web event mappings for a specified window
controlNamePrefix	Prototype method that you can reimplement to prefix your own control names
createPicture	Creates a picture for the form or control
createPictureAsType	Creates a picture with the specified image type for the form or control

Method	Description
doWindowEvents	Processes all pending Window events for this window and all its children
enableEvent	Provides control at run time of JADE logic execution of events associated with forms or controls
getDeskTopWorkArea	Returns the work area of the desktop
getHwnd	Returns the Microsoft Windows handle for a form or control
getMonitorArea	Returns the full area of the current monitor on which the window resides
getMonitorWorkArea	Returns the position of the available desktop area of the monitor on which the window resides
getPersistentObject	Returns a reference to the persistent object corresponding to the receiver
getPropertyDisplay	Returns the text displayed by the Painter's Properties dialog for the property
getSystemColor	Returns a Windows system color
getSystemMetrics	Returns a Windows metrics value
getTextExtent	Returns the length required to display a text string
getTextHeight	Returns the height required to display a text string
getTextHeightForWidth	Returns the height of the text string in pixels as it would be displayed when word wrapped within a rectangle of the specified width using the font of the window
getWebEventMappings	Returns all of the Web event mappings for a specified window
getWindowHandle	Returns the Microsoft Windows handle as a MemoryAddress for a form or control
hasPropertyPage	Returns whether the window has its own property page dialog
hwnd	Returns the Windows handle to a form or control
isEventEnabled	Returns whether the specified event is currently enabled for the window
move	Enables logic to move and size a form or control from a single method call
refresh	Forces a repaint of a form or control
refreshNow	Forces an *immediate* repaint or update of a form or control
releaseMouse	Releases the mouse capture from a window
removeWebEventMapping	Removes the specified Web event mapping for a specified window
screenToWindow	Converts an absolute screen position into a position relative to the window
setDefaultPainterControlProperties	Called by the JADE Painter when a new control is created (re-implement the method on the **Control** class or subclass to set any user default properties)

Method	Description
setDragAndDropFiles	Enables files and folders to be dragged and dropped onto a form or control
setEventMapping	Enables the method to be dynamically set at run time
setEventMappingEx	Enables the method to be dynamically set at run time
setFocus	Sets the focus to a window
showHelp	Invokes Window’s help if the **helpKeyword** or **helpContextId** properties of the object are set
showPropertyPage	Displays the property page dialog for the window
windowToScreen	Converts a position relative to the window into an absolute position on the screen
zOrder	Places a form or control at the front or back of the z-order within its graphical level

For details, see "Window, Form, and Control Methods", later in this document. See also the graphics methods defined in the **Window** class, under "Graphics Properties and Methods", later in this document.

Form Class

A *form* is a window that acts as a container for controls that display information and that permit user input. As the **Form** class is a subclass of the **Window** class, it inherits all of the properties and methods defined in that class. A form can be one of the following types.

- Standalone form
- MDI frame that can contain other forms
- MDI child form owned by an MDI frame (the default)

An MDI form without a caption can always be maximized programmatically. However, if the user switches between MDI children (for example, by using the CTRL+F6 key combination), the next form is displayed in a maximized state only if the value of the **maxButton** property for that form is **true**. This property is used for this situation even if the value of the **controlBox** property is **false**, meaning that none of the MDI buttons or the control icon are displayed on the menu line.

Forms have properties that determine aspects of their appearance (for example, position, size, and color), and aspects of their behavior (for example, whether they can be resized). Forms can also respond to events initiated by the user or triggered by the system. For example, you could write logic in the **click** event method of a form that would enable the user to change the color of a form by clicking it. (For details about creating forms when a workstation has multiple monitors, see "Window Class", earlier in this document.)

In addition to properties and events, you can use methods to manipulate forms from logic. For example, you can use the **move** method of the **Window** class to change the location of a form or control. This effectively changes more than one of the **left**, **top**, **width**, or **height** properties of the form or control. As each change causes the control to resize, potentially each control resizes each time the properties are set (or the control is aligned to the form or control if the window is a container of other controls). The **move** method ensures that one resize only occurs. For example, if you individually change the **left**, **top**, **width**, or **height** properties for a folder, the folder resizes four times. It then resizes the sheets four times, which can resize the contents, and so on.

If a form has the **mdiFrame** property set to **true**, that form is automatically created as an MDI frame; that is, a special kind of form that can contain other forms (MDI child forms). If a form has the **mdiChild** property set to **true**, that form is automatically placed in a special kind of form called an MDI frame that can contain these MDI child forms.

New instances of a form can be created in logic by using the **create** method, and then displayed by using the **show** method or the **showModal** method, as shown in the following example.

```
vars
    form : InquiryForm;      // declare the form to be of type InquiryForm
begin
    create form;             // create the form window
    form.show;               // display the window
end;
```

The **show** method or the **showModal** method causes the windows to be made visible. Using the **showModal** method causes all other forms of that application to be disabled until that form is unloaded. Execution of the **showModal** method completes only when the form is unloaded, or made invisible. The **show** method is invoked and starts execution before the **load** event method is invoked.

Within the **show** method, it is the presence of an **inheritMethod** call that causes the **load** event method to be invoked. Consequently, any user logic positioned prior to the **inheritMethod** call is executed before the **load** event method executes.

The **load** event of a form is executed when the first event for the form occurs. This is normally when the initial **show** or **showModal** statements are executed. This delayed execution of the form **load** event means that logic can set and alter properties and controls on the created form before the **load** event is executed.

Set the **borderStyle** property to define the border of a form, and set the **caption** property to put text in the title bar. Setting the **borderStyle** property to **BorderStyle_None** removes the border (if the form does not have a menu). If you want your form to have a border without the title bar, Control-Menu, **Maximize** button, or **Minimize** button, delete any text from the **caption** property of the form, and set the **controlBox**, **maxButton**, and **minButton** properties for the form to **false**.

Windows will not create a form that has a **caption** line with a single border but instead creates a form with a fixed double border. If the value of the **borderStyle** property is set to **BorderStyle_Single** (**1**) and the form has a caption line, the value of the **borderStyle** property is modified to fixed **BorderStyle_Double** (**3**), to reflect the actual border style in use.

To force a form defined with the **mdiChild** property set to **0** (use application MDI default) to be specifically an MDI form or non-MDI form, set the **Application** class **defaultMdi** property before the **create** method is used.

When printing the text of a control or displaying the text on a form, an ampersand character (&) is displayed as an underscore to the next character of the text string, with the following exceptions.

- When the ampersand character (&) is the last character of the text
- When the control is a text box, combo box, or list box
- When the control is a table and the **displayHotKey** property is set to **false** (the default)
- When the **drawTextAt** or **drawTextIn** method is called for a window

For controls that display an ampersand character (**&**) as an underscore, two ampersand characters (**&&**) are printed or displayed as a single ampersand character; for example, **&&Print** is printed or displayed as **&Print**.

For details about applying skins to forms in your application, see "JadeSkinApplication Class" and "JadeSkinForm Class", in Chapter 1. For a summary of the constants, properties, methods, and events defined in the **Form** class, see "Form Class Constants", "Form Properties", "Form Methods", and "Form Events", in the following subsections.

For details about programmatically providing the functionality to float, dock, and pin MDI child forms in your own application logic and providing tabs on MDI forms, see "Floating, Docking, and Pinning MDI Child Forms" and "MDI Child Form Tabs", respectively, in later subsections.

For details about testing controls on a form, see "Testing Tools and Control Identification", later in this document. For details about the support of mouse wheel requests to scroll up, down, or across a form, see "Window Class", earlier in this document. For details about monitoring the basic operating style of forms for a JADE application, see "Monitoring the Basic Operating Style of Forms", in the following subsection.

Inherits From: Window

Inherited By: ControlAboutBox, JadeBackupDatabaseDialog, JadeFrameForm, JadePrintPreviewFind, JadeSkinMaint, JadeSkinMaintenance, JadeSkinSelect, JadeSkinSelection, JadeTestDialog, JadeWebServiceShowStats, and user-defined Form classes

Monitoring the Basic Operating Style of Forms

When you run a JADE application, a Microsoft Windows **Window** class is created for each kind of window, to establish its basic operating style. This class is not visible to you unless you use a Windows or user-written utility to monitor the style of forms.

The **Window** classes for forms in a JADE application have the following names.

- **Jade::Form** for a standard JADE form.
- **JadeMsgBox::Form** for a message box created using the **msgBox** method of the **Application** class when using skins or message box caption translations; otherwise, a standard Windows message box is used.
- **JadeException::Form** for an exception dialog; that is, the standard exception, lock exception, lock retry, or ODBC exception dialog.

Floating, Docking, and Pinning MDI Child Forms

You can programmatically provide the functionality to float, dock, and pin MDI child forms in your own application logic, as follows.

- You can control whether users can invoke an MDI menu on an MDI child form by right-clicking on the caption of the form. The MDI popup menu can contain the following commands.

Menu Command	Action
Close	Closes the form (the same as clicking the **Close** button or the Context-Menu **Close** command)
Close All But This	Closes all other MDI children in the current MDI frame except for the current form
Close All But Pinned	Closes all MDI child forms that are not pinned and have the **allowClose** property set to **true**
Float	Floats the current MDI child form
Dock	Re-docks the MDI child form in its MDI frame
Pin	Toggles the pinned status of an MDI child form

The **Form** class provides the following **Boolean** primitive type properties, all of which have a default value of **false**, which means that the MDI menu is not displayed by right-clicking the MDI child caption:

Property	Contains the...
showMdiCloseAllButPinnedMenu	**Close All But Pinned** command
showMdiCloseAllButThisMenu	**Close All But This** command
showMdiCloseMenu	**Close** command
showMdiDockMenu	**Dock** command
showMdiFloatMenu	**Float** command
showMdiPinMenu	**Pin** command

If the value of all of these properties is **true**, right-clicking the MDI child caption displays a menu.

These properties are ignored if the form is not an MDI child form. In addition, the **Close All But This**, **Close**, and **Close All But Pinned** commands are ignored if the **allowClose** property is set to **false**.

The property values can be set at run time and in the JADE Painter.

Note The **Float** command is disabled if the form is floating and the **Dock** command is disabled if the MDI child form is already docked in the MDI frame.

The **Form** class also provides the **Boolean** type **mdiPinned** property, which defaults to **false**. This property is read-only at run time, to allow access from your application logic to the pinned status of a form.

- The **Form** class provides the methods listed in the following table.

Method	Description
floatMdi	Floats an MDI child. It does nothing if the form is already floating or if the form is not an MDI child.
dockMdi	Docks an MDI child. It does nothing if the form is not floating or if the form is not an MDI child.
isMdiFloating	Returns **true** if the MDI child is floating or **false** if it is docked or it is not an MDI child.

- The **Form** class provides the event methods listed in the following table.

Event Method	Description
mdiFloated	Called after the user floats an MDI child. Appears in the **Form Event Methods** list box in the JADE development environment. This event is not called if the **floatMdi** method is called to float the form.
mdiDocked	Called after the user docks an MDI child. Appears in the **Form Event Methods** list box in the JADE development environment. This event is not called if the **dockMdi** method is called to dock the form.

Note When floated, an MDI child form is positioned on the screen in the same position, except it is not a child restricted to the MDI frame; for example, it can be dragged to another monitor.

The MDI child form is always on top of the MDI frame. To reposition the form programmatically, set the **left** and **top** properties, or use the **Window** class **move** method.

When an MDI child form is docked, the position and size of the form is restored to the values it had when it was floated, if the current top-most MDI child in the MDI frame is not maximized. If the current top-most MDI child in the MDI frame is maximized, the docked form is also maximized.

Applies to Version: 2020.0.01 and higher

MDI Child Form Tabs

You can programmatically provide the functionality to display tabs for MDI child forms in user systems, by controlling the style of MDI child form styles. The styles are:

- Standard MDI child forms (the default).
- Standard MDI child forms with a tab for each child form on the MDI frame. The child forms can be maximized, minimized, and restored.
- MDI with a tab only for each child form on the MDI frame. Only the top MDI child form is visible and it is always maximized.

For standard MDI child forms with a tab and MDI forms with a tab only, when tabs are displayed:

- The tab contains the caption of the child form. Clicking on the tab brings that form to the top. The child form with focus has its tab highlighted. If the caption is too long to fit in the tab, the first and last part of the caption are displayed separated by points of ellipsis (...).
- Moving the mouse over the tab displays the full caption in a bubble help display.
- A down arrow button is displayed at the end of the tabs. Clicking the button displays the full list of captions of open child forms.

  Selecting an entry in the list brings that form to the front (that is, the same functionality that the Window menu provides for arranging and manipulating child windows in the JADE development environment). The order of the list can be alphabetic, the last-used, or creation order.
- Not all tabs are displayed if there is not sufficient room. (Use the down arrow or Window menu to locate a form that is not displayed.)
- JADE provides the ability to pin selected tabs to the left of the displayed tabs. The user can also drag tabs to another position, by clicking the tab and dragging it. (It can be dragged only within the pinned or non-pinned grouping.)

The **Application** class provides the following properties and class constants.

Note These properties apply only to forms that have been created in version 2020.0.01 and higher.

- The Integer type **mdiStyle** property, which defaults to **MdiStyle_Mdi** (0), sets the default MDI style for an application at run time. You can also set the application style in the Mdi Style group box on the **Form** sheet of the Define Application dialog in the JADE development environment.

  Set this property to one of the following values.

  - **MdiStyle_Mdi** (0)
  - **MdiStyle_Mdi_With_Tabs** (1)
  - **MdiStyle_Tabs_Only** (2)
- The Integer type **mdiWindowListOrder** property, which defaults to **MdiWindowList_Order_Creation** (0), sets the order that child forms are displayed in the Window menu list of child forms and in the MDI Tabs down arrow list. You can also set the application window list order in the Mdi Window List Order group box on the **Form** sheet of the Define Application dialog in the JADE development environment. Set this property to one of the following values.

  - **MdiWindowList_Order_Creation** (0)

    The Window List menu and the MDI tabs down arrow show the list of MDI child forms in creation order.
  - **MdiWindowList_Order_LastUse** (1)

    The Window list menu and the MDI tabs down arrow show the list of MDI child forms in last-use order.
  - **MdiWindowList_Order_Alphabetic** (2)

    The Window list menu and the MDI tabs down arrow show the list of MDI child forms in caption alphabetic order.

Applies to Version: 2020.0.01 and higher

Form Class Constants

The constants provided by the **Form** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
MdiChild_IsMdi	2	MdiChild_NotMdi	1
MdiChild_UseAppDefault	0	QueryUnload_MdiChild	4
QueryUnload_TaskManager	3	QueryUnload_UnloadMethod	1
QueryUnload_User	0	QueryUnload_Windows	2
TaskBar_State_Error	4	TaskBar_State_Indeterminate	1
TaskBar_State_NoProgress	0	TaskBar_State_Normal	2
TaskBar_State_Paused	8	TrayIcon_LeftClick	1
TrayIcon_LeftDblClick	3	TrayIcon_RightClick	2
TrayIcon_RightDblClick	4	WindowState_Maximized	2
WindowState_Minimized	1	WindowState_Normal	0

Form Properties

The properties defined in the **Form** class are summarized in the following table.

Property	Description
allMenuItems	Contains an array of all menu items on the form
allowClose	Contains the status of the Control-Menu **Close** command
allowDocking	Controls the type of docking that is allowed by the window
backBrush	Contains the bitmap to tile the background of the form
caption	Contains the text displayed in the title bar of the form
clientHeight	Contains the height of the client area of a form in pixels
clientWidth	Contains the width of the client area of a form in pixels
clipControls	Specifies whether the Windows environment creates a clipping region that excludes controls contained by the object
controlBox	Specifies whether a Control-Menu icon is displayed on a form at run time
icon	Contains the icon displayed for a form or sheet at run time
maxButton	Specifies whether a form has a **Maximize** button
mdiChild	Specifies whether a form is displayed as an MDI child form inside an MDI form
mdiClientScrollHorzPos	Contains the horizontal scroll positions of the MDI client windows of a form
mdiClientScrollVertPos	Contains the vertical scroll positions of the MDI client windows of a form
mdiFrame	Specifies whether the form is always built as an MDI frame

Property	Description
minButton	Specifies whether a form has a **Minimize** button
mdiPinned	Allows access from your application logic to the pinned status of a form
minimumHeight	Specifies the minimum height of the form in pixels
minimumWidth	Specifies the minimum width of the form in pixels
modalResult	Contains the returned value for the **showModal** method call in runtime forms that are initiated as modal
scaleForm	Specifies whether the form and all of its contents are scaled to match the current font scaling attribute
scrollBars	Specifies whether an object has horizontal or vertical scroll bars
scrollHorzPos	Contains the position of the horizontal scroll bar
scrollVertPos	Contains the position of the vertical scroll bar
secureForm	Specifies whether the form is secure
showMdiCloseAllButPinnedMenu	Specifies whether the **Close All But Pinned** command is displayed in the popup menu when a user right-clicks on the caption of an MDI child form
showMdiCloseAllButThisMenu	Specifies whether the **Close All But This** command is displayed in the popup menu when a user right-clicks on the caption of an MDI child form
showMdiCloseMenu	Specifies whether the **Close** command is displayed in the popup menu when a user right-clicks on the caption of an MDI child form
showMdiDockMenu	Specifies whether the **Dock** command is displayed in the popup menu when a user right-clicks on the caption of an MDI child form
showMdiFloatMenu	Specifies whether the **Float** command is displayed in the popup menu when a user right-clicks on the caption of an MDI child form
showMdiPinMenu	Specifies whether the **Pin** command is displayed in the popup menu when a user right-clicks on the caption of an MDI child form
topLevelMenuItems	Contains an array of all top-level menu items on the form
webBrowserAutoRefreshInterval	Specifies the number of seconds after which the Web page is automatically refreshed
webBrowserAutoRefreshURL	Specifies a string value containing the URL to invoke when the automatic refresh interval is reached
webBrowserDisableBackButton	Specifies whether the previous Web page is displayed when the user clicks the **Back** button
webEncodingType	Contains the content type used to submit the form to the JADE application
webFileName	Contains the name of the background image displayed on a Web page
windowState	Contains the visual state of a form window at run time

For details, see "Window, Form, and Control Properties", later in this document.

Form Methods

The methods defined in the **Form** class are summarized in the following table.

Method	Description
activeChild	Returns the current active MDI child for an MDI frame form
addControl	Adds a control to a form at run time
allowWebPrinting	Enables the correct printing of the contents of the Web page in Microsoft Internet Explorer 4 and higher
alwaysOnTop	Specifies whether the visible form is placed above all other forms on the desktop
centreWindow	Centers the form in the window
controlCount	Returns the number of controls on the form
controls	Enables logic to access the controls on an active form at run time
create	Calls a constructor that builds a description of the form
delete	Calls a destructor that removes the form and its associated window
dockMdi	Docks a floating MDI child form back into its MDI frame
ensureCaptionIsVisible	Ensure the caption is visible on at least one monitor
flagControlForSave	In Painter, saves the specified control that was not painted on the form using Painter functionality
floatMdi	Floats an MDI child form; that is, it takes the MDI child form out of the MDI frame and allows it to be moved independently from the MDI frame
generateHTML	Generates the HyperText Markup Language (HTML) string for the form
generateHTMLStatic	Generates the static HyperText Markup Language (HTML) string for the form and builds the full Uniform Resource Locator (URL) action line
getControlByName	Returns the control on the form with the specified name
getFormParent	Returns the parent of a form set by the **setFormParent** method or if the parent form was set directly by using a Windows Application Programming Interface (API) call
getMdiFrame	Accesses the current MDI frame for a form
getRegisteredFormKeys	Returns an array of the form keys that are in effect for the form
getScrollRange	Returns the scroll range information for the window
hasSystemTrayEntry	Specifies whether the form currently has a system tray entry
isCaptionVisible	Specifies whether the caption is visible on at least one monitor
isMdiFloating	Specifies whether an MDI child form is floating
isModal	Specifies whether the form was displayed using the **showModal** method
isPrinterForm	Returns whether the form was declared as a printer form on the New Form dialog in the JADE Painter
menuItemCount	Returns the number of items in the menu
menuItems	Enables logic to access the menu items in a menu at run time

Method	Description
moveMdiClient	Positions the client window
paintIfRequired	Causes the form to be repainted if a repaint is required
popupMenu	Invokes a popup menu for a form
registerFormKeys	Establishes the entire set of key codes in which the key events of a form are interested
removeSystemTrayEntry	Removes the system tray entry for the form
registerWindowMsg	Registers a Windows message with the JADE GUI environment
replyAsBinary	Returns the Binary message to the Web browser without modification
resetFirstChange	Resets the first change status of the referenced form and all **TextBox**, **JadeRichText**, and **JadeEditMask** controls on the form
setApplicationSkin	Sets the skin for a specific form and its controls
setBackDrop	Sets the background image for the form
setFormParent	Sets the parent of a form to be another form
setFormSkin	Sets the skin for a specific form, regardless of the setting of the **skinCategoryName** property
setScrollRange	Enables control of the scroll ranges for form and picture controls
setSkin	Sets a skin for the form, overriding any skin that is set for the application
setSystemTrayEntry	Places an entry in the system tray for the form
setTaskBarProgress	Sets the extent of the progress to be displayed on the application icon in the taskbar
setTaskBarState	Sets the state of the taskbar icon for the application
show	Makes the form visible in its current state
showModal	Makes the form visible in its current state, disabling all other application forms
tabNext	Tabs to the next visible control that can have focus, in the enabled tab order
tabPrior	Tabs to the prior visible control that can have focus, in the enabled tab order
unloadForm	Unloads an active form

For details, see "Window, Form, and Control Methods", later in this document.

Form Events

The event methods defined in the **Form** class are summarized in the following table.

Event	Description
activate	Occurs when a form becomes the active window
click	Occurs when the user presses and then releases the left mouse button
contextMenu	Occurs after the right **mouseUp** event and after the **keyUp** event

Event	Description
dblClick	Occurs when the user presses and releases the left mouse button and then presses and releases it again
deactivate	Occurs when a different form becomes the active window
dragDrop	Occurs when a dragged window is dropped over a form belonging to the same application
dragOver	Occurs for each form of the application over which a window is dragged
firstChange	Occurs when the contents of a control on the form change
formMove	Occurs when a form is moved on the screen
gotFocus	Occurs when a control receives the focus
keyDown	Occurs when the user presses a key
keyPress	Occurs when the user presses and releases an ANSI key
keyUp	Occurs when the user releases a key
load	First event called for the form and its controls (except the **windowCreated** event for the controls)
lostFocus	Occurs when a control loses the focus
mdiDocked	Occurs after the user docks an MDI child form
mdiFloated	Occurs after the user floats an MDI child form
mouseDown	Occurs when the user presses a mouse button
mouseEnter	Occurs when the user moves the mouse onto a form
mouseLeave	Occurs when the user moves the mouse off a form
mouseMove	Occurs when the user moves the mouse
mouseUp	Occurs when the user releases a mouse button
paint	Occurs when part or all of a form is exposed after it has been moved or enlarged
queryUnload	Occurs before a form closes, to give the logic the opportunity to reject the closure
resize	Occurs when the size of a form is changed
scrolled	Occurs when the user scrolls a form
sysNotify	Occurs when a specified JADE system event occurs
trayIconClicked	Occurs when a user clicks a system tray entry created by the **setSystemTrayEntry** method
unload	Occurs when a form is about to be removed from the screen
userNotify	Occurs when triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Control Class

The **Control** class enables you to define properties and methods that apply to all controls. As the **Control** class is an abstract subclass of the **Window** class, it inherits all of the properties and methods defined in that class. A *control* is a specialized child window that is resident on top of a form or another control. To create your own user-defined controls, subclass the **BaseControl** class (for details, see Chapter 5 of the *JADE Developer's Reference).*

For details about testing controls on a form, see "Testing Tools and Control Identification", in the following subsection.

Some controls can be parents of other controls in the JADE Painter; for example, **BaseControl**, **Frame**, **GroupBox**, **JadeDockBar**, **JadeDockContainer**, **JadeMask**, **Picture**, **Sheet**, and **StatusLine**.

Controls have properties that determine aspects of their appearance (for example, position, size, and color), and aspects of their behavior (for example, whether they receive the focus). Controls can also respond to events initiated by the user or triggered by the system. For example, you could write logic in the **click** event of a control that would enable the user to indicate acceptance of a process by clicking it.

In addition to properties and events, you can use methods to manipulate controls from logic.

Note Use the **move** method of the **Window** class to change more that one of the **left**, **top**, **width**, or **height** properties for a form or control. As each change causes the control to resize, if the window is a container of other controls, potentially each control resizes each time the properties are set or the control is aligned to the form or control.

The **move** method ensures that one resize only occurs. For example, if you individually change the **left**, **top**, **width**, or **height** properties for a folder, the folder resizes four times. It then resizes the sheets four times, which can **resize** the contents, and so on.

Some **Control** instance properties have no meaning for certain controls, but are defined for other reasons; for example, as a scroll bar control does not have any text, the font properties have no relevance.

When printing the text of a control or displaying the text on a form, an ampersand character (&) is displayed as an underscore to the next character of the text string, with the following exceptions.

- When the ampersand character (**&**) is the last character of the text
- When the control is a text box, combo box, or list box
- When the control is a table and the **displayHotKey** property is set to **false** (the default)
- When the **drawTextAt** or **drawTextIn** method is called for a window

For controls that display an ampersand character (**&**) as an underscore, two ampersand characters (**&&**) are printed or displayed as a single ampersand character; for example, **&&Print** is printed or displayed as **&Print**.

For details about printing a background picture over which is drawn the report itself, see "Layering Print Output", under the **Printer** class "Defining Your JADE Report Layouts", in Chapter 1.

User-defined subclassed controls are not supported on forms defined as Web pages, and are ignored when HTML is generated.

When an event method defined at the **Control** class is deleted, the user is warned of this. If the deletion is confirmed, the event methods that correspond to this control event are also deleted. Renaming a control event method automatically renames all event implementations.

For details about applying skins to controls in your application, see "JadeSkinApplication Class" and "JadeSkinControl Class and Subclasses", in Volume 2.

For a summary of the constants, properties, methods, and event defined in the **Control** class, see "Control Class Constants", "Control Properties", "Control Methods", and "Control Event", in the following subsections. For details about the support of mouse wheel requests to scroll up, down, or across a control, see "Window Class", earlier in this document.

Inherits From: Window

Inherited By: ActiveXControl, BaseControl, BrowseButtons, Button, CheckBox, ComboBox, Folder, Frame, GroupBox, JadeDockBase, JadeDotNetVisualComponent, JadeEditMask, JadeRichText, JadeTextEdit, JadeXamlControl, Label, ListBox, MultiMedia, Ocx, OleControl, OptionButton, Picture, ScrollBar, StatusLine, Table, TextBox, user-defined control classes

Testing Tools and Control Identification

JADE creates each control with an assigned identification number that is included in the creation description passed to Windows. Windows accepts only 16-bit identifiers; that is, 1 through 65,535. This control identifier is used by some testing tools to locate the control elements involved in the script processing via a Windows API call.

Call the **Control** class **getControlWindowId** method if you want to use the control identification number in your JADE code.

Although Windows requires only that child identifiers be unique for their direct parent, JADE creates the identifiers so that they are unique for the whole form. An identifier assigned to each control defined in the JADE Painter does not change when the painted form is altered unless the control is deleted and re-added; that is, you do not need to update testing scripts because the identifiers have changed.

Controls Saved in the JADE Painter

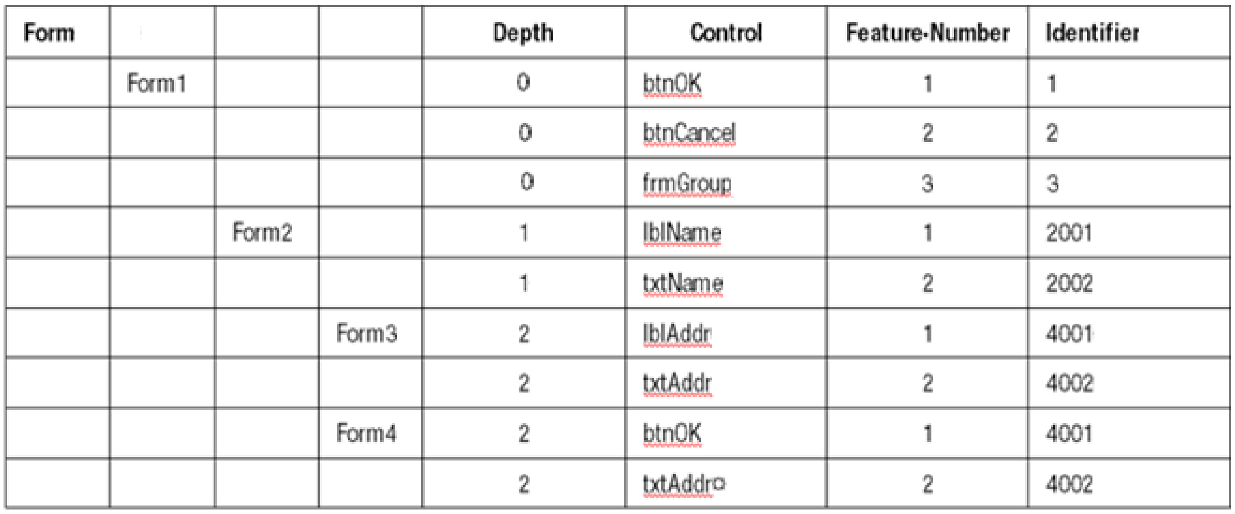
For controls painted on a form, an identifier is created based on the JADE feature number assigned to each control property on a form and the subclass depth of the form on which the control is defined.

Feature numbers are persistent and are unique only within each class.

The identifier is assigned as follows.

```
(subclass-depth-from-the-Form-class - 1) * 2000 + control-property-feature-number
```

In the example in the following table, the form hierarchy consists of four forms, where **Form1** is a subclass of the **Form** class, **Form2** is a subclass of **Form1**, and **Form3** and **Form4** are subclasses of **Form2**.



Form				Depth	Control	Feature Number	Identifier
	Form1			0	btnOK	1	1
				0	btnCancel	2	2
				0	frmGroup	3	3
		Form2		1	lblName	1	2001
				1	txtName	2	2002
			Form3	2	lblAddr	1	4001
				2	txtAddr	2	4002
			Form4	2	btnOK	1	4001
				2	txtAddro	2	4002

This allows for a subclass depth of up to 31 from the **Form** class. If more than 31 levels exist, controls defined on subclasses beyond the 31 depth are assigned a sequential identifier starting at 31000. If the feature number of a control exceeds the permitted range (which is initially 2000), the range for that **Form** subclass is increased by another 2000 and the subclass depth that can be handled is reduced accordingly. For this to happen, more than 2000 properties, methods, and constants would need to have been defined specifically on that form subclass (not counting superclass elements).

The **extractControlIdsCSV** and **extractControlIdsCSVforSchema** methods of the **Schema** class output to a file comma-separated value (CSV) entries containing the control ids of controls, in the following format.

```
schema_name, form_name, control_name, control_id
```

Controls Created by Runtime Logic

Controls created at run time are sequentially assigned the next available number in the range beyond the last-used form subclass range. In the previous table, the next available number would be set to 6000.

The number assigned to each dynamically created control is affected by the order in which the control is created.

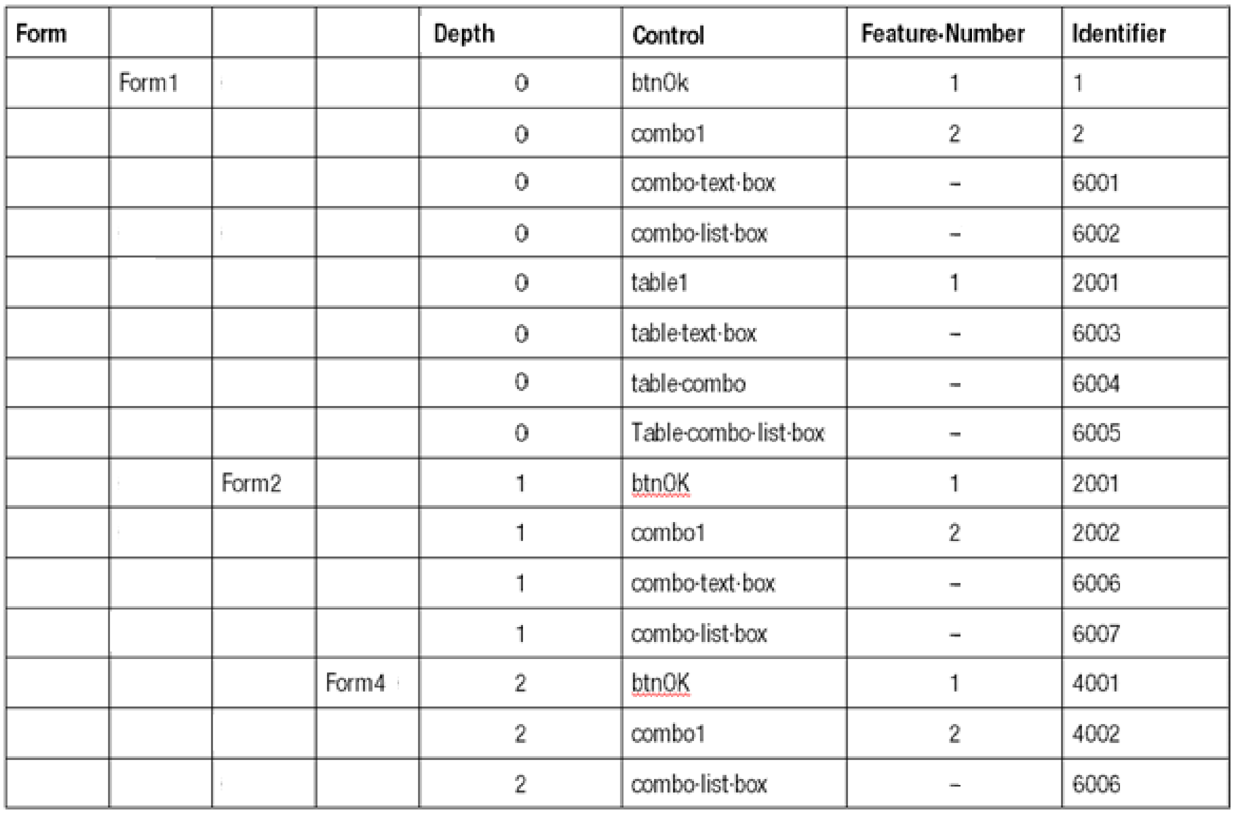
Internal Controls

Some of the standard JADE controls create component controls that are implied by their definition. These child controls do not have a JADE object and are accessed implicitly via logic through the parent control. The controls in which this situation occurs are:

- **ComboBox**, which has a list box associated with it. For a simple combo box or a spin box, the list box is a child of the combo box. For all other combo box styles, the list box is a top-level window.
- **ComboBox**, which can have a child text box.
- **JadeEditMask**, which has from zero through *n* child text boxes, depending on the definition of the edit mask.
- **Table** when the **inputType** property is used to determine the cell control input style, in which case a table can create a text box, a combo box, and an edit mask child.

The control identifier is assigned in the same way as it is for controls that are created at run time; that is, they are sequentially assigned the next available number in the range beyond the last-used form subclass range.

In the previous table, the next available number would be set to 6000. The number assigned to each internally created control is affected by the order in which it is created, as shown in the example in the following table.



Form				Depth	Control	Feature·Number	Identifier
	Form1			0	btnOk	1	1
				0	combo1	2	2
				0	combo·text·box	–	6001
				0	combo·list·box	–	6002
				0	table1	1	2001
				0	table·text·box	–	6003
				0	table·combo	–	6004
				0	Table·combo·list·box	–	6005
		Form2		1	btnOK	1	2001
				1	combo1	2	2002
				1	combo·text·box	–	6006
				1	combo·list·box	–	6007
			Form4	2	btnOK	1	4001
				2	combo1	2	4002
				2	combo·list·box	–	6006

Control Class Constants

The constants provided by the **Control** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
BorderStyle_3DRaised	3	BorderStyle_3DSunken	2
ParentAspect_AnchorBoth	#c	ParentAspect_AnchorBottom	#8
ParentAspect_AnchorRight	#4	ParentAspect_CenterBoth	#30
ParentAspect_CenterHorz	#10	ParentAspect_CenterVert	#20
ParentAspect_None	0	ParentAspect_StretchBoth	#3
ParentAspect_StretchBottom	#2	ParentAspect_StretchRight	#1
Show3D_Not3D	1	Show3D_Use3D	2
Show3D_UseAppDefault	0	Show3D_UseBorderStyle	3

Control Properties

The properties defined in the **Control** class are summarized in the following table.

Property	Description
automaticCellControl	Specifies whether cells of **Table** controls are handled automatically
backBrush	Contains the bitmap to tile the background of the control
borderColorSingle	Contains the RGB scheme color of the control border when the **borderStyle** property is set to **BorderStyle_Single** (1)
focusBackColor	Contains the background color of a control when that control has focus
focusForeColor	Contains the foreground color of a control when that control has focus
fontBold	Specifies whether the font style is bold
fontItalic	Specifies whether the font style is italic
fontName	Contains the font used to display text in a control
fontSize	Contains the size of the font used for text displayed in a control
fontStrikethru	Specifies whether the font style is strikethrough
fontUnderline	Specifies whether the font style is underlined
foreColor	Contains the foreground color used to display text in a window
form	Provides access to a form of a control from the object
index	Contains an identifier to differentiate between control copies that have the same name
parent	Contains the direct parent of the control
parentAspect	Contains the aspect of the control in relation to its parent
parentBottomOffset	Contains the pixel offset of the bottom edge of a control from the bottom of the client area of its parent
parentRightOffset	Contains the pixel offset of the right edge of a control from the right side of the client area of its parent
relativeHeight	Specifies whether the size and position of a control is relative to the size and position of its parent
relativeLeft	Specifies whether the size and position of a control is relative to the size and position of its parent
relativeTop	Specifies whether the size and position of a control is relative to the size and position of its parent
relativeWidth	Specifies whether the size and position of a control is relative to the size and position of its parent
show3D	Specifies whether automatic three-dimensional effects are added to the appearance of a control
tabIndex	Contains the tab order of a control within its parent form
tabStop	Specifies whether a user can use the Tab key to set the focus to a control

For details, see "Window, Form, and Control Properties", later in this document.

Control Methods

The methods defined in the **Control** class are summarized in the following table.

Method	Description
aboutBox	Calls the About box for the control
canBeChildOf	Returns whether the control can be placed on the specified form or control
canControlHaveChildren	Returns whether a control is permitted to be the parent of other controls
canHaveAsChild	Returns whether the specified control can be placed on the control
centreWindow	Centers the control on its parent
clientHeight	Returns the height of the client area of a control in pixels
clientWidth	Returns the width of the client area of a control in pixels
convertFormPosition	Converts the specified horizontal and vertical positions into coordinates relative to the control
createPicture	Creates a picture for the control
createPictureIndirect	Creates a short binary that contains an instruction concerning the window to be copied
delete	Deletes the control
getControl	Accesses dynamically created controls of the same type as the receiver
getControlWindowId	Returns the Windows identifier that JADE creates and assigns to each control window
getFormLeft	Returns the absolute left position of the control on the form in pixels
getFormTop	Returns the absolute top position of the control on the form in pixels
getRegisteredKeys	Returns an array of the keys that are in effect for the control
is3D	Returns whether the control was drawn three-dimensionally
isInPainter	Returns whether the control is on the Painter form
isMoveable	Returns whether the control can be moved in the JADE Painter
isMySheetVisible	Returns whether the sheet is the current visible sheet
isSelectable	Returns whether instances of the control can be selected in the JADE Painter
isSizeable	Returns whether the control can be resized in the JADE Painter
loadControl	Enables an existing control to be cloned at run time
registerKeys	Establishes the entire set of key codes in which the key events of a control are interested
resetFirstChange	Resets the **firstChange** event status of the control and all children of the control
saveProperties	Allows local properties edited in the Painter to be saved with the painted form
setSkin	Sets the skin for a specific control, regardless of the setting of the **skinCategoryName** property
setFocus	Sets the focus to a control

Method	Description
setFontProperties	Sets the **fontBold**, **fontName**, and **fontSize** properties for text associated with the control
showMySheet	Provides a control with the ability to make the sheet the top sheet of a folder control

For details, see "Window, Form, and Control Methods", later in this document.

Control Events

The event methods defined in the **Control** class are summarized in the following table.

Event	Description
contextMenu	Occurs after the right **mouseUp** event and after the **keyUp** event
windowCreated	Called for all controls when the window for the control is created, so that a control can be initialized when the window for the control is present

For details, see "Window, Form, and Control Events", later in this document.

ActiveXControl Class

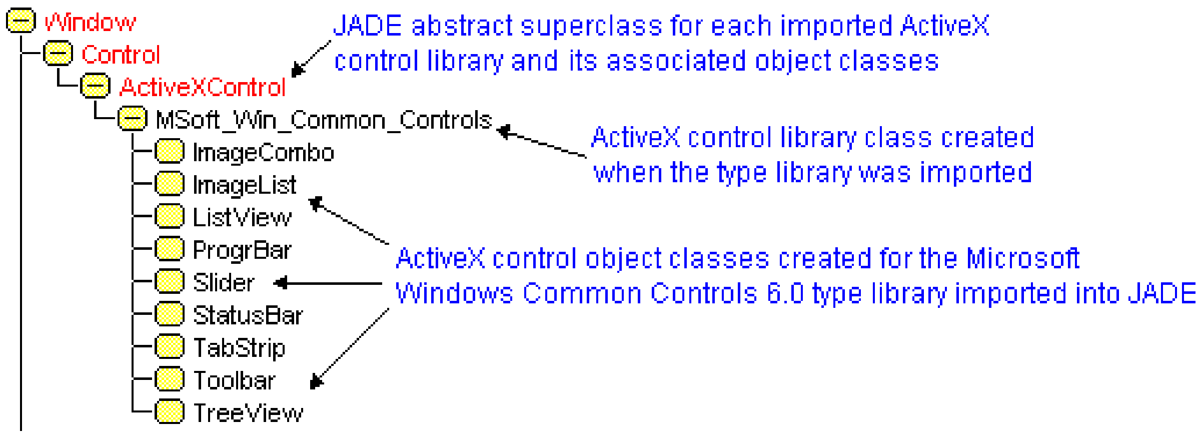
The **ActiveXControl** class provided by JADE supports all interfaces of an ActiveX control type library imported into JADE. An ActiveX control is a special type of COM object. It is a user interface object that implements an interface that supports its use on forms. (This control was formerly known as an OCX control.)

An ActiveX control can optionally have a graphical user interface and can fire events. This enables existing third-party functions, such as highly specialized controls, to be used within your JADE applications.

Caution To register and run an ActiveX control, all libraries used by the control must be available.

When you import an ActiveX control library into JADE, an abstract class of the specified library name is created as a subclass of the **ActiveXControl** class. This abstract class becomes the superclass for all classes that are subsequently generated corresponding to objects in the imported ActiveX control library. This ActiveX control library class is the superclass that supports the control object classes within that library, which have properties, methods, and events created that correspond to the default interfaces.

ActiveXControl subclasses inherit all of the standard properties and methods of the **Window** and **Control** classes, although not all of these inherited properties and methods have meaning to the control. (For a caveat on the use of SVG files when printing **ActiveXControl** controls, see "Portable Printing" under "Printer Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*.) The following example shows the hierarchy of the Microsoft Windows Common Controls type library imported into the JADE **ActiveXControl** class.


Window
Control
ActiveXControl
MSoft_Win_Common_Controls
ImageCombo
ImageList
ListView
ProgrBar
Slider
StatusBar
TabStrip
Toolbar
TreeView
JADE abstract superclass for each imported ActiveX control library and its associated object classes
ActiveX control library class created when the type library was imported
ActiveX control object classes created for the Microsoft Windows Common Controls 6.0 type library imported into JADE

Creating an instance of the ActiveX control in JADE does not create an instance of the **ActiveXControl** object, which occurs only when the control is added to a form. At run time, JADE translates the property and method requirements into ActiveX control equivalents and then calls the control to perform the function.

If you have added a property to an imported **ActiveXControl** object and flagged that property as a design time property (for details, see "Selecting Your Design Time Properties", in Chapter 5 of the *JADE Developer's Reference*), any value assigned to that property by using the JADE Painter Properties dialog will not be propagated through to the runtime instance of that control when the form is created.

If the property value is required in the runtime instance, you can copy it from the persistent instance to the runtime instance in the **windowCreated** method on the **ActiveXControl** subclass. (As the **windowCreated** method will be a reimplementation of the **Control**::**windowCreated** method, you must include an **inheritMethod** instruction in your **windowCreated** method.)

To refresh an ActiveX control that has changed but is already imported into JADE, simply import the ActiveX control type library again and give it the same name that it had previously.

An imported ActiveX control is added to the **Control** palette of the JADE Painter and cannot be distinguished from standard JADE-supplied controls. The **ActiveXControl** class is supported on forms defined as Web pages only when running on a Microsoft Internet Explorer browser.

Notes As **ActiveXControl** class methods run only on the client node (including when running in JADE thin client mode), all methods generated for imported ActiveX controls include the **clientExecution** option in the method signature.

In JADE thin client mode, ActiveX control objects run only on the presentation client.

As the ActiveX control is external to JADE, no documentation about the control is contained within JADE, and any problems in functionality of the control should be taken up with the suppliers of that control. If an error occurs, JADE is dependent on the control returning error information that can be displayed to the user.

Transparent sibling controls are painted before an **ActiveXControl**, regardless of their **zOrder** settings. It is not possible to handle the painting of transparent controls in the correct **zOrder** when some controls are directly painted by JADE and others are painted by Windows separately.

If the control object returns a reference to another ActiveX interface in response to a method call or the get of a property, JADE creates an instance of the corresponding JADE interface class and returns a reference to that instance instead. (A mapping is maintained between JADE interface instances and COM interface instances.) For details about importing ActiveX control and automation type libraries into JADE, see Chapter 4 of the *JADE External Interface Developer's Reference*.

For a summary of the JADE property, methods, and event defined in the **ActiveXControl** class, see "ActiveXControl Class Property", "ActiveXControl Class Methods", and "ActiveXControl Class Event", in the following subsections. (Refer to your COM documentation for details about properties, methods, or constants provided by imported ActiveX control type libraries.)

ActiveXControl Class Property

The property defined in the **ActiveXControl** class is summarized in the following table.

Property	Description
usePresentationClient	Specifies whether the ActiveX control runs on the presentation client (the default) or application server if it was created by using the **makeAutomationObject** method.

For details, see "Window, Form, and Control Properties", later in this document.

ActiveXControl Class Methods

The methods defined in the **ActiveXControl** class are summarized in the following table.

Method	Description
beginNotifyAutomationEvent	Registers the receiver to be notified when an event occurs on an ActiveX control created as an automation object by using the **makeAutomationObject** method
endNotifyAutomationEvent	Terminates a previous **beginNotifyAutomationEvent** event
getInterface	Returns the specified ActiveX interface.
loadPicture	Creates a picture object from an external file.
makeAutomationObject	Creates an ActiveX automation object instead of a control. Although the ActiveX object was imported as a control, it can be used as an automation object.
makePicture	Creates a picture object from a JADE binary.
processInputFromWeb	When reimplemented, processes ActiveX controls used on Web pages.
savePicture	Saves the image of a picture to the specified external file.
setEventMapping	Enables an event method to be dynamically set at run time

For details, see "Window, Form, and Control Methods", later in this document.

ActiveXControl Class Event

The event method defined in the **ActiveXControl** class is summarized in the following table.

Event	Description
paint	Occurs when part or all of an ActiveX control needs to be painted

For details, see "Window, Form, and Control Events", later in this document.

BaseControl Class

The **BaseControl** class is an abstract control that you can subclass to create your own controls. For details, see "Creating Your Own Control Classes", in Chapter 5 of the *JADE Developer's Reference*.

Instances of the **BaseControl** class provide all of the base properties and methods available from the **Window** and **Control** superclasses and have the following capabilities.

- Can be a parent control.

  If this is not wanted, override the **canControlHaveChildren** method defined for controls, returning **false** (the default value is **true**).

- Can have the focus.

  If this is not wanted, set the **canHaveFocus** property to **false** (the default value is **true**).

- Can have scroll bars and control of those scroll bars.

  For details about the support of mouse wheel requests to scroll up, down, or across a base control, see "Window Class", earlier in this document.

- Can be transparent.
- Can use the **clipControls** property.
- Definition of all standard events for controls.

You need only add a subclass to the **BaseControl** class for it to be automatically added to the Painter toolbar. You must write logic for each event controlling the subclass behavior.

The **BaseControl** class is not supported on forms defined as Web pages and is ignored when HTML is generated.

Note If you have mapping method logic on subclassed controls that rely on this logic when executing, you must protect that logic from situations where properties of subclassed controls are accessed or referenced by JADE processes such as the JADE Painter, JADE Translator utility, or the loading of schemas.

For a summary of the properties, methods, and events defined in the **BaseControl** class, see "BaseControl Properties", "BaseControl Methods", and "BaseControl Events", in the following subsections.

BaseControl Properties

The properties defined in the **BaseControl** class are summarized in the following table.

Property	Description
canHaveFocus	Specifies whether the control can have focus

Property	Description
clipControls	Specifies whether the Windows environment creates a clipping region that excludes controls contained by the object
scrollBars	Specifies whether an object has horizontal or vertical scroll bars
scrollHorzPos	Contains the position of the horizontal scroll bar
scrollVertPos	Contains the position of the vertical scroll bar
transparent	Causes the control to be placed above all other sibling controls and the controls or form underneath to be visible

For details, see "Window, Form, and Control Properties", later in this document.

BaseControl Methods

The methods defined in the **BaseControl** class are summarized in the following table.

Method	Description
getScrollRange	Returns the scroll range information for the window
setScrollRange	Enables control of the scroll ranges for form and picture controls

For details, see "Window, Form, and Control Methods", later in this document.

BaseControl Events

The event methods defined in the **BaseControl** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button, and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each form or control of the application over which a window is dragged
gotFocus	When a control receives the focus
keyDown	When the user presses a key while the control has the focus
keyPress	When the user presses and releases an ANSI key
keyUp	When the user releases a key while the control has the focus
lostFocus	When a control loses the focus
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer

Event	Occurs…
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
scrolled	When the user scrolls the control
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

BrowseButtons Class

The **BrowseButtons** class is a subclass of the **Control** class and inherits all of the properties and methods defined in the **Control** and **Window** classes. Browse buttons are not supported on Web page forms and are ignored when HTML is generated.

The **BrowseButtons** control consists of four buttons that can be clicked. The buttons are normally interpreted to indicate:

- First
- Prior
- Next
- Last

Clicking a button invokes an event describing which button was clicked. If a button is painted in a **StatusLine** control, that control resizes to position the browse control within the inner border of the **StatusLine** control.

For a summary of the constants and events defined in the **BrowseButtons** class, see "BrowseButtons Class Constants" and "BrowseButtons Events", in the following subsections.

BrowseButtons Class Constants

The constants provided by the **BrowseButtons** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Browse_First	1	Browse_Last	4
Browse_Next	3	Browse_Prior	2

BrowseButtons Events

The event methods defined in the **BrowseButtons** class are summarized in the following table.

Event	Occurs…
browse	When the user presses the left mouse button over a browse button

Event	Occurs…
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each control of the application over which a window is dragged
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a form or control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Button Class

The **Button** class is a subclass of the **Control** class and inherits all of the properties and methods defined in the **Control** and **Window** classes.

The user selects a command button to begin, interrupt, or end a process. When selected, a command button appears as though it was pushed in, and may also be called a *push button*. To display text on a command button, set its **caption** property. A user can always choose a command button by clicking it.

The **Button** class has the following variations, which are controlled by the **style** property.

- A normal push down and spring up button.
- A two-state button. Press an up button to cause it to go and stay down. Press a down button to cause it to come and stay up.
- An automatic two-state button. Press the button to cause it to go and stay down. All other automatic two-state buttons with the same parent come up; that is, only one button of a group can be down at any time.

A button is drawn with a border only when it has focus or it is the default button.

The **Button** control is drawn using the current Windows theme that is in use, unless the currently active JADE skin defines the button look and feel. The exception to this is any button that has the button **backColor** value set to any color other than the default. In this case, the button is drawn in the JADE style of earlier releases.

When a **Button** control is skinned and at least one of the skin images is 32-bit (which supports transparency), the control is treated as though it is transparent; that is, the control is painted on its parent without the area being erased with the effective value of the **backColor** property. Instead, the parent shows through any transparent areas of the images (for rounded corners, for example). In addition, any semi-transparent parts of the images are anti-aliased with the parent image so that they are displayed with smooth corners over any background color.

For a summary of the constants, properties, and events defined in the **Button** class, see "Button Class Constants", "Button Properties", and "Button Events", in the following subsections.

Button Class Constants

The constants provided by the **Button** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
AutoSize_Button	1	AutoSize_None	0
AutoSize_Picture	2	ButtonPicture_Asterisk	7
ButtonPicture_Bin	10	ButtonPicture_Cross	2
ButtonPicture_Door	8	ButtonPicture_Exclamation	6
ButtonPicture_No	3	ButtonPicture_None	0
ButtonPicture_Question	5	ButtonPicture_RecycleBin	9
ButtonPicture_Stop	4	ButtonPicture_Tick	1
Style_2State	2	Style_Auto2State	1
		Style_Normal	0

Button Properties

The properties defined in the **Button** class are summarized in the following table.

Property	Description
autoSize	Specifies whether a control is automatically resized to fit its contents
buttonPicture	Contains predefined bitmaps to be placed on a button control
cancel	Marks the button as the **Cancel** button
caption	Contains an access key to assign to a control
default	Marks the button as the default button
picture	Contains a graphic to be displayed in a control
pictureDisabled	Contains the picture displayed when the picture box is disabled
pictureDown	Contains the picture displayed when the button is down
style	Contains the style of push button
value	Specifies whether the state of the button is up (**false**) or down (**true**)
webFileName	Contains the name of the image displayed on a button on a Web page

For details, see "Window, Form, and Control Properties", later in this document.

Button Events

The event methods defined in the **Button** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button

Event	Occurs…
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button, and then presses and releases it again
dragDrop	When a dragged window is dropped over a control belonging to the same application
dragOver	For each form or control of the application over which a window is dragged
gotFocus	When a control receives the focus
keyDown	When the user presses a key while the control has the focus
keyPress	When the user presses and releases an ANSI key
keyUp	When the user releases a key while the control has the focus
lostFocus	When a control loses the focus
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a button
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a button
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

CheckBox Class

A check box control displays a check mark. When selected, the check mark is removed when the check box is cleared. Use the **CheckBox** control class to give the user a True/False or Yes/No option. You can use check boxes in groups to display multiple choices from which the user can select one or more options.

As the **CheckBox** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** class and the **Window** class.

Check boxes and option buttons function in a similar manner except that any number of check boxes on a form can be selected at the same time. In contrast, only one option button in a group can be selected. To display text next to the check box, set the **caption** property. Use the **value** property to determine the state of the selected box.

The check box automatically sets the height of the control to fit the caption.

Check boxes are drawn using the current Windows theme that is in use, unless the currently active JADE skin defines the check box look and feel.

For a summary of the constants, properties, and events defined in the **CheckBox** class, see "CheckBox Class Constants", "CheckBox Properties", and "CheckBox Events", in the following subsections.

CheckBox Class Constants

The constants provided by the **CheckBox** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Alignment_Left	0	Alignment_Right	1

CheckBox Properties

The properties defined in the **CheckBox** class are summarized in the following table.

Property	Description
alignment	Specifies whether the text is placed before or after the button bitmap image
autoSize	Specifies whether a check box control is automatically resized to fit its contents
caption	Contains a caption to assign to a check box control
readOnly	Specifies whether a check box control is read-only for user input
transparent	Causes a check box control to be placed above all other sibling controls and the controls underneath to be visible
value	Specifies whether the state of a check box control is checked or unchecked

For details, see "Window, Form, and Control Properties", later in this document.

CheckBox Events

The event methods defined in the **CheckBox** class are summarized in the following table.

Event	Description
change	Indicates that the contents of the control have changed
click	Occurs when the user presses and then releases the left mouse button
contextMenu	Occurs after the right **mouseUp** event and after the **keyUp** event
dragDrop	Occurs when a dragged window is dropped over a window belonging to the same application
dragOver	Occurs for each window of the application over which a window is dragged
gotFocus	Occurs when a control receives the focus
keyDown	Occurs when the user presses a key while the control has the focus
keyPress	Occurs when the user presses and releases an ANSI key
keyUp	Occurs when the user releases a key while the control has the focus
lostFocus	Occurs when a control loses the focus
mouseDown	Occurs when the user presses a mouse button
mouseEnter	Occurs when the user moves the mouse onto a control

Event	Description
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	Occurs when the user moves the mouse off a control
mouseMove	Occurs when the user moves the mouse
mouseUp	Occurs when the user releases a mouse button
paint	Occurs when part or all of a control is exposed
sysNotify	Occurs when a specified JADE system event occurs
userNotify	Occurs when triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

ComboBox Class

A combo box control combines the features of a **TextBox** control and a **ListBox** control. Use a combo box to give the user the choice of typing in the text box portion or selecting an item from the list portion of the control. As the **ComboBox** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** class and **Window** class.

If the **ComboBox** class with a style other than **Style_Simple** or **Style_DropDown** accepts text, the entry that is alphabetically less than or equal to the entered text is selected, without issuing an event. The user then has only to enter as much as required to uniquely identify a specific entry. (You can enter text in **Style_Simple** or **Style_DropDown** combo boxes that does not relate to the entries in the list. For more details, see the **style** property.)

If the combo box list is open, the Esc key closes the combo box list. In addition, the Alt+UpArrow shortcut keys action closes the combo box list when it has focus and the Alt+DownArrow shortcut keys action opens the combo box list when it has focus.

Notes If the form has a **Button** control with the **cancel** property set to **true** and the open combo box has focus, the Esc key is processed by the combo box; not by the **Cancel** button. The **Cancel** button action occurs only if the combo box list is not open.

The **alternatingRowBackColor** and **alternatingRowBackColorCount** properties are available only for the **ListBox** control and the **JadeTableSheet** class, and they do not exist for a **ComboBox** control. However, you can achieve the alternating style on a combo box by using a skin. If a **ComboBox** skin references a **ListBox** skin that has the **alternatingRowBackColorCount** property value set (that is, it is greater than zero (**0**)) for the list box, the value of the **alternatingRowBackColor** list box skin property is used to draw the alternating background colors of the displayed combo box list.

A combo box, which is case-insensitive, can have a maximum of 32,000 items, although the maximum number of entries displayed in the list portion of the control is 20.

Tip It is much more efficient to copy a GUI value into a local variable for reuse rather than request the value again; for example, **comboBox.listCount** requires the calling of a combo box method to retrieve the value. Storing the value in a local property for reuse avoids a significant overhead for the second and subsequent requests for that value when it will not change.

The first of the following examples is much more expensive than the second of these examples.

```
while count <= comboBox.listCount do  // inefficient use of the variable

vars                                  // recommended use of the variable
    comboListCount : Integer;
begin
    comboListCount := comboBox.listCount;
    while count <= comboListCount do
        ...
    endwhile;
end;
```

You can implement filtering of combo box and list box entries to enable users to locate a required list item more quickly, using standard combo box and list box facilities to achieve this filtering. (For details, see "**Filtering Entries in Combo Box and List Box Text**", in Chapter 2 of the *JADE Development Environment User's Guide*.) You can achieve the filtering of entries in your own JADE systems, as follows.

- Set the value of the **style** property for the **ComboBox** control class to **ComboBox.Style_DropDown** (0).
- Set the value of the **hasPictures**, **hasTreeLines**, and **hasPlusMinus** properties to **false**.
- The value of the **sorted** property must be set to **false** after the list is loaded.
- A disabled **description** list item is added as the first entry in the list (after sorting is turned off).
- The **change** event on the combo box or associated text box calls a filtering routine to hide or show the entries based on the new text entered. Setting the level of the item to **2** hides the entries, which is why there has to be an entry at item position **1** that is always at level **1**. (A child item must have a parent.)

Note For a combo box, the **click** event is not normally implemented. The selection action is performed on the **closeup** event so that nothing happens until the user completes the selection action.

If the list item entry in a combo box is too long to fit in the list box portion of the control, bubble help showing the complete text is automatically displayed over the text portion of the entry when the mouse is moved over that entry. Bubble help is no longer displayed if the mouse is moved off that entry or after approximately three seconds.

Notes Clicking on the bubble help generates a **click** event for that list entry.

The automatic display of bubble help does not occur if the combo box already has bubble help defined (by using the **Window** class **bubbleHelp** property).

To disable the automatic bubble help for a combo box, set the **bubbleHelp** property of that control to a space.

Keystrokes that are entered while the list box area of a combo box has focus are concatenated into a string for searching the contents of the list box. For example, when you entering the letter **T** followed by the letter **O**, focus goes to the entry whose text began with **T** and then to an entry beginning with **TO**, provided that you enter the letter **O** within one second of the letter **T**. If the gap between the entry of the letters **T** and **O** is more than one second, focus goes to an entry whose text begins with **T** and then to an entry beginning with **O**. However, if the first character of the sequence is repeated (that is, the letter **T** in this example), the search finds the next entry that starts with that character (which is treated like two **T** characters separated by more than one second). For example, if you have list items **6001**, **6002**, **6601**, and **6602** and you quickly enter **66**, **6001** is first selected and then **6002**, so that you can access each list item rather than going directly from **6001** to **6601**.

Note The resulting search string for a series of keystrokes can be of any length, provided there is less than one second gap between each keystroke.

To implement the handling of a **ComboBox** control to achieve the same functionality as the default **Table** class **InputType_ComboBox** value, only the following is required when the combo box entries are already loaded:

- In the **click** event for the combo box:

```
table1.text := combo1.text
```

- In the **cellInputReady** event for the table:

```
combo1.text := table1.text
```

Note For the arrays associated with combo boxes (for example, **itemBackColor**), the only methods that are implemented are **at**, **atPut** (which enables you to use the square brackets notation to access the elements), **createIterator** (which allows logic to do a **foreach** over the array), **size**, and **size64**.

For more details, see the **Table** class **inputType** and **cellControl** properties. See also the **Control** class **automaticCellControl** property.

As the combo box can have the functionality of a **ListBox** control, see also "Setting Properties for Individual Items in a List Box" and "Using a List Box to Display a Hierarchy or Tree", later in this document.

For a summary of the constants, properties, methods, and events defined in the **ComboBox** class, see "ComboBox Class Constants", "ComboBox Properties", "ComboBox Methods", and "ComboBox Events", in the following subsections.

For details about the support of mouse wheel requests to scroll up, down, or across a combo box control, see "Window Class", earlier in this document.

ComboBox Class Constants

The constants provided by the **ComboBox** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
DisplayCollection_Forward	0	DisplayCollection_NoPrior	0
DisplayCollection_Prior	2	DisplayCollection_Reversed	1
ItemNotFound	-1	ItemPictureType_Closed	0
ItemPictureType_Leaf	2	ItemPictureType_Open	1
PictureClick_ItemPicture	4	PictureClick_KeyBoard	5
PictureClick_Picture	3	PictureClick_PlusMinus	1
PictureClick_TreeLine	2	Style_DropDown	0
Style_DropDownComboList	3	Style_DropDownList	2
Style_Simple	1	Style_SpinBox	4

ComboBox Properties

The properties defined in the **ComboBox** class are summarized in the following table.

Property	Description
defaultLineHeight	Specifies the default height of lines in a combo box independent of the font size

Property	Description
hasPictures	Specifies whether the picture images are displayed
hasPlusMinus	Specifies whether plus/minus images are displayed
hasTreeLines	Specifies whether the tree lines are drawn
itemBackColor	Contains the background color of each item in a combo box
itemData	Contains a specific number for each item in a combo box
itemEnabled	Specifies whether individual items can be enabled in a combo box
itemExpanded	Contains the expansion (or collapse) status of each item
itemForeColor	Contains the text color to assign to each item in a combo box
itemLevel	Contains the hierarchical level of each item
itemObject	Contains an object for each entry in a combo box
itemPicture	Contains a picture for individual items in a combo box
itemPictureType	Contains the type of picture of each item
itemText	Contains the text of an item in a combo box
listIndex	Contains the index of the currently selected item in the combo box
listObject	Contains the associated object of the currently selected item in the combo box
listWidth	Contains the width of the drop-down list box portion of the combo box
maxLength	Contains how much text can be entered into the text box part of a combo box
pictureClosed	Contains the qualifying picture image displayed for an entry
pictureLeaf	Contains the qualifying picture image displayed for an entry
pictureMinus	Contains the qualifying picture image displayed for an entry
pictureOpen	Contains the qualifying picture image displayed for an entry
picturePlus	Contains the qualifying picture image displayed for an entry
selLength	Contains the number of characters selected in a combo box
selStart	Contains the starting point of selected text
selText	Contains the string containing the currently selected text
sortCased	Specifies whether the sorting is case-sensitive
sorted	Specifies whether the elements of a combo box are automatically sorted alphabetically
style	Contains the type of combo box, and the behavior of its list box portion
text	Contains the text contained in the edit area of a combo box

For details, see "Window, Form, and Control Properties", later in this document.

ComboBox Methods

The methods defined in the **ComboBox** class are summarized in the following table.

Method	Description
addItem	Adds a new item to a combo box
addItemAt	Adds a new item at a specified item index to a combo box
clear	Clears the contents of a combo box
closeDropDown	Closes (hides) the drop-down list of the combo box
displayCollection	Attaches the specified collection to the list portion of the combo box
findObject	Searches the list entries of a combo box for the object specified in the **object** parameter
findString	Searches the entries in a combo box for an entry with the specified string
findStringCaseSensitive	Searches the entries in a combo box for an entry with the specified case-sensitive string
findStringExact	Searches the entries in a combo box for an entry that exactly matches the specified string
findStringExactCaseSensitive	Searches the entries in a combo box for an entry that exactly matches the specified case-sensitive string
getCollection	Returns the collection attached to the combo box by the **displayCollection** or **listCollection** method
isDroppedDown	Returns **true** if the drop-down list of the combo box is visible
itemHasSubItems	Returns a **Boolean** value that indicates whether an item has subitems
itemVisible	Returns a **Boolean** value that indicates whether an item is visible
listCollection	Enables combo boxes to have a collection attached to them
listCount	Returns the number of items in the list portion of a combo box
newIndex	Returns the index of the item most recently added to a combo box
refreshEntries	Refreshes the displayed list of entries in the combo box
removeItem	Removes an item (and its subitems) from a combo box
showDropDown	Opens (shows) the drop-down list of the combo box

For details, see "Window, Form, and Control Methods", later in this document.

ComboBox Events

The event methods defined in the **ComboBox** class are summarized in the following table.

Event	Description
change	Indicates that the contents of the control have changed
click	Occurs when the user presses and then releases the left mouse button

Event	Description
closeup	Indicates that the list portion of a combo box has closed up
contextMenu	Occurs after the right **mouseUp** event and after the **keyUp** event
dblClick	Occurs when the user presses and releases the left mouse button and then presses and releases it again
displayEntry	Occurs when a combo box is attached to a collection object and the text to be displayed for an entry in the collection is required
displayRow	Occurs for each entry in the collection of the list portion of the current combo box, to display the contents of the row
dragDrop	Occurs when a dragged window is dropped over a window belonging to the same application
dragOver	Occurs for each control of the application over which a window is dragged
dropDown	Occurs when the list portion of a combo box is about to drop down, and returns **true** if the drop-down list is visible
gotFocus	Occurs when a control receives the focus
keyDown	Occurs when the user presses a key while the window has the focus
keyPress	Occurs when the user presses and releases an ANSI key
keyUp	Occurs when the user releases a key while the window has the focus
lostFocus	Occurs when a control loses the focus
mouseDown	Occurs when the user presses a mouse button
mouseEnter	Occurs when the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	Occurs when the user moves the mouse off a control
mouseMove	Occurs when the user moves the mouse
mouseUp	Occurs when the user releases a mouse button
paint	Occurs when part or all of a control is exposed
pictureClick	Occurs when the picture area before the text of an item is clicked with the mouse
pictureDblClick	Occurs when the picture area before the text of an item is double-clicked with the mouse
scrolled	Occurs when the user scrolls a combo box
sysNotify	Occurs when a specified JADE system event occurs
userNotify	Occurs when triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Folder Class

A folder control is a special container that has one or more sheets. A sheet is the same as a **GroupBox** control, in that it holds a series of painted controls. One sheet only is visible at any time, but tabs for each sheet are displayed at the top of the folder or at a user-specified edge. These tabs can be clicked to make the selected sheet visible. When the user clicks on a tab, the corresponding sheet of the folder is displayed (that is, both the **click** and the **sheetChg** event methods occur). A specific sheet can also be selected by using the keyboard if the caption of the sheet includes an accelerator sequence (for example, keying Alt+C selects the sheet labeled **&Customers**).

A folder control provides the following standard behavior.

- The Ctrl+Page Down shortcut keys move the focus of a folder to the next enabled visible sheet when that folder or a child of the folder has focus.
- The Ctrl+Page UP shortcut keys move the focus of a folder to the prior enabled visible sheet when that folder or a child of the folder has focus.
- For a user *not* using accessibility software (for example, Freedom Scientific's JAWS), if the form with the focus is:
  - A Multiple Document Interface (MDI) form, Ctrl+Tab moves to the next MDI child form and Ctrl+Shift+Tab moves to the previous MDI child form.
  - Not an MDI form, Ctrl+Tab moves to the next sheet of the folder containing the focus or to the next folder in the tab order if the focus is not within a folder and Ctrl+Shift+Tab moves to the previous sheet of the folder containing the focus or to the previous folder in the tab order if the focus is not within a folder.
- For a user using accessibility software, regardless of whether the form is an MDI form or not, Ctrl+Tab moves to the next sheet of the folder containing the focus or to the next folder in the tab order if the focus is not within a folder and Ctrl+Shift+Tab moves to the previous sheet of the folder containing the focus or to the previous folder in the tab order if the focus is not within a folder.
- If a child of a **Folder** control has focus and the current sheet is changed, focus is moved to the folder and the activated sheet is highlighted.
- The left and right arrow keys enable you to cycle forwards and backwards through all sheets in the folder when focus is currently on a folder.
- The up and down arrow keys enable you to cycle up and down rows of sheets in the folder when focus is currently on a folder.

The **Folder** control uses the same area of the screen for multiple sheets, with the user and developer able to select the sheet that is visible.

For a form defined as a Web page, the folder control does not directly generate HTML, but the sheets of the folder are arranged horizontally and vertically.

The **Folder** control provides the following features.

- Choice of tab or button style for selecting the active sheet.
- Sheets can be displayed with or without simulated multiple sheet edges.
- Tabs can be displayed in multiple lines or in a single scrollable line.
- Tabs can be variable or fixed in size.
- Tabs can be stretched to fill each tab line or left as a ragged edge.
- The height of the tab area can be controlled or even hidden, if required.

- Icons can be assigned to each sheet and displayed on the appropriate tab. JADE creates a large and a small icon for use with a form if they are present in the icon file when the **app.icon** and **form.icon** properties are set.

- The alignment of the icon and the text on the tabs can be controlled.

For a **Folder** control with the **tabsStyle** property set to **Folder.TabsStyle_Buttons**, tabs are drawn using the skin of the **Button** control if the application has defined a button skin. As the **Folder** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** class and the **Window** class. In the JADE Painter, a folder has two sheets by default. Deleting a sheet also deletes all controls on that sheet.

For details about the support of mouse wheel requests to scroll up, down, or across a folder control, see "Window Class", earlier in this document.

» To add more sheets

1. Select the folder.

2. Click the sheet button.

3. Click on the tabs area of the folder.

» To display a specific sheet

- Click the tab of the appropriate sheet

Controls can then be painted on that sheet in the normal way.

Note The sheets in the folder are ordered according to their **tabIndex** values.

For a summary of the constants, properties, methods, and events defined in the **Folder** class, see "Folder Class Constants", "Folder Properties", "Folder Methods", and "Folder Events", in the following subsections.

Folder Class Constants

The constants provided by the **Folder** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
TabsAlignment_Center	0	TabsAlignment_IconLeft	2
TabsAlignment_Left	1	TabsLines_MultiLine	1
TabsLines_MultiLineEdged	0	TabsLines_SingleLine	2
TabsPosition_Bottom	3	TabsPosition_Left	1
TabsPosition_Right	2	TabsPosition_Top	0
TabsStyle_Buttons	1	TabsStyle_RightSloped	2
TabsStyle_Tabs	0		

Folder Properties

The properties defined in the **Folder** class are summarized in the following table.

Property	Description
tabActiveColor	Contains the color drawn for the active tab of multiple sheet folders
tabInactiveColor	Contains the color drawn for the inactive tabs of multiple sheet folders
tabsAlignment	Contains the placement of the icon and text in the tab
tabsFixedWidth	Contains the width of tabs
tabsHeight	Contains the height of each line of tabs
tabsLines	Specifies whether the tabs are displayed in multiple or single lines
tabsPosition	Contains the position of folder tabs
tabsRaggedRight	Specifies whether the tabs in a line are stretched to fill the available space for the tab line
tabsStyle	Specifies whether a folder control is displayed with sculptured tabs or push buttons
topSheet	Contains the sheet that is currently visible

For details, see "Window, Form, and Control Properties", later in this document.

Folder Methods

The methods defined in the **Folder** class are summarized in the following table.

Method	Description
canHaveAsChild	Returns whether the control can be placed on the folder, to ensure that only sheets are placed on folders
dragSheet	Returns the sheet of the tab that corresponds to the *x* and *y* location of the drag operation
sheets	Returns the number of sheets

Additionally, the methods summarized in the following table are defined in the **Control** superclass.

Method	Description
isMySheetVisible	Determines if the sheet on which it is placed is the current visible sheet
showMySheet	Makes the sheet on which it is placed the top sheet

For details, see "Window, Form, and Control Methods", later in this document.

Folder Events

The event methods defined in the **Folder** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
gotFocus	When a control receives the focus
keyDown	When the user presses a key while the control has the focus
keyPress	When the user presses and releases an ANSI key
keyUp	When the user releases a key while the control has the focus
lostFocus	When a control loses the focus
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
querySheetChg	Before the user causes a sheet change using the keyboard or mouse, to give the logic the opportunity of rejecting the change
sheetChg	When the user clicks on the tab of a sheet of a folder that is not currently the top sheet, to enable that sheet to become visible
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Frame Class

The **Frame** control provides a way of grouping controls in a container, aligning the frame positionally or by size within its container, and for the use of three-dimensional effects.

Frame text is displayed in the area bounded by the three-dimensional (3D) borders. If the borders are too large, the text area of the frame may not exist and no text is displayed.

Note The **StatusLine** control is a variation on the **Frame** control.

The frame can be a container for groups of controls. Use a frame or a group box when multiple groups of option buttons are required on the same form. An option button toggles other option buttons within the same container.

The frame (and the status line control) has several custom properties that enable you to adjust the three-dimensional appearance of the control.

To add controls to a frame

1. Select the frame in the Painter.
2. Draw the new controls inside the frame.

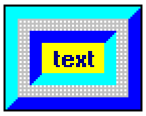
When the frame is moved, all of the child controls move with it. The top and left position of the child controls are relative to the internal area of the frame (that is, the client area).

For details about three-dimensional effects, see "Three-Dimensional Effects in Frame Controls", in the following subsection. For details about docking frame controls, see the **JadeDockBar**, **JadeDockBase**, and **JadeDockContainer** classes, later in this document.

For a summary of the constants, properties, methods, and events defined in the **Frame** class, see "Frame Class Constants", "Frame Properties", "Frame Methods", and "Frame Events", later in this section.

Three-Dimensional Effects in Frame Controls

The following image is an example of three-dimensional (3D) effects in a **Frame** control:



The following table lists the properties defined in the **Frame** class that enable you to create three-dimensional effects.

Region	Property	Description
Black border	borderStyle	The frame can be enclosed with a border.
Outer bevel rectangle	bevelOuter, bevelOuterWidth	The outer bevel can be empty, inset, or raised.
Second outer rectangle	boundaryWidth, boundaryColor, boundaryBrush	The boundary region between the inner and the outer bevels can be empty. It can also be a plain color or a dotted pattern.
Inner bevel rectangle	bevelInner, bevelInnerWidth	The inner bevel can be empty, inset, or raised.
Half of outer and inner bevel rectangles	bevelColor	The color of the bevel section. The position depends on whether the bevel is inset or raised.
Half of outer and inner bevel rectangles	bevelShadowColor	The color of the bevel shadow section. The position depends on whether the bevel is inset or raised.
Rectangle containing text	backColor, alignment, caption, foreColor	The actual client area of the frame surface, where any child controls are placed.

A frame also has properties that are designed to automatically resize the frame and its children when the container of the frame is resized. For a frame or status line control, the position (**0,0**) is the top left of the area inside the 3D frame. Controls with the frame as its parent are not painted in the border area.

Note The position (0,0) is the top left corner of the client area. Mouse positions are also relative to this, and negative position values may result when the mouse is in the top or left border area.

Frame Class Constants

The constants provided by the **Frame** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
AlignChildren_All	2	AlignChildren_None	0
AlignChildren_Width	1	AlignContainer_AllHorizontal	5
AlignContainer_AllVertical	6	AlignContainer_All	5
AlignContainer_Bottom	2	AlignContainer_None	0
AlignContainer_Stretch	4	AlignContainer_Top	1
AlignContainer_Width	3	Alignment_Center_Bottom	8
Alignment_Center_Middle	7	Alignment_Center_Top	6
Alignment_Left_Bottom	2	Alignment_Left_Middle	1
Alignment_Left_Top	0	Alignment_Right_Bottom	5
Alignment_Right_Middle	4	Alignment_Right_Top	3
Bevel_Inset	1	Bevel_None	0
Bevel_Raised	2	BoundaryBrush_Dotted	1
BoundaryBrush_Solid	0		

Frame Properties

The properties defined in the **Frame** class are summarized in the following table.

Property	Description
alignChildren	Specifies whether child controls placed inside the frame are aligned
alignContainer	Specifies whether the control aligns itself to its parent container so that the frame automatically resizes with the container
alignment	Contains the alignment of the text in a control
allowDocking	Controls the type of docking that is allowed by the window
bevelColor	Contains the color used to paint the bevel areas of a 3D control
bevelInner	Contains the style of the inner bevel of the frame and status line controls
bevelInnerWidth	Contains the width of the bevel along the four sides of the frame to determine the height of the three-dimensional shadow effect
bevelOuter	Contains the style of the outer bevel of the frame
bevelOuterWidth	Contains the width of the bevel along the four sides of the frame to determine the height of the three-dimensional shadow effect

Property	Description
bevelShadowColor	Contains the color used to paint the bevel areas of a 3D control
boundaryBrush	Specifies whether the boundary area is a plain color or is painted with a dotted brush
boundaryColor	Contains the color of the boundary area
boundaryWidth	Contains the width of the boundary area
caption	Contains the text displayed in the title bar
clipControls	Specifies whether the Windows environment creates a clipping region that excludes controls contained by the object
transparent	Causes the control to be placed above all other sibling controls and the controls underneath to be visible
wordWrap	Specifies whether text displayed in a caption advances to the next line when the current line is filled

For details, see "Window, Form, and Control Properties", later in this document.

Frame Methods

The methods defined in the **Frame** class are summarized in the following table.

Method	Description
clearHTML	Clears previously generated HTML code from the frame
writeHTML	Generates HTML code when a Web session is active, or outputs the frame to the printer if there is no active Web session

For details, see "Window, Form, and Control Methods", later in this document.

Frame Events

The event methods defined in the **Frame** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a control belonging to the same application
dragOver	For each control of the application over which a window is dragged
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control

Event	Occurs…
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

GroupBox Class

The group box control provides a way of grouping controls in a container.

Group box text is drawn within the top border.

The **GroupBox** class can be a container for groups of controls. Use a **GroupBox** class when multiple groups of option buttons are required on the same form. An option button toggles other option buttons within the same container.

To add controls to a group box

1. Select the group box in the Painter.
2. Draw the new controls inside the group box.

When the group box is moved, all of the child controls move with it. The top and left position of the child controls are relative to the internal area of the group box (that is, the client area).

The **GroupBox** control class provides the **Sheet** subclass.

For a summary of the properties, methods, and events defined in the **GroupBox** class, see "GroupBox Properties", "GroupBox Methods", and "GroupBox Events", in the following subsections.

GroupBox Properties

The properties defined in the **GroupBox** class are summarized in the following table.

Property	Description
caption	Contains the text displayed in the title bar
clipControls	Specifies whether the Windows environment creates a clipping region that excludes controls contained by the object
transparent	Causes the control to be placed above all other sibling controls and the controls underneath to be visible

For details, see "Window, Form, and Control Properties", later in this document.

GroupBox Methods

The methods defined in the **GroupBox** class are summarized in the following table.

Method	Returns false to specify that the control cannot be…
isMoveable	Moved in the JADE Painter
isSizeable	Resized in the JADE Painter

For details, see "Window, Form, and Control Methods", later in this document.

GroupBox Events

The event methods defined in the **GroupBox** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

JadeDockBar Class

The **JadeDockBar** control class is a subclass of the **JadeDockBase** class. The **JadeDockBar** control is intended as a container for a logical grouping of other standard JADE controls that can be dragged and docked at another position on the form or it can be floated.

The **JadeDockBar** control provides the following functionality.

- The control is a container for other controls.
- The ability to define a border with **borderStyle** property value of the:
  - **Window** class constant **BorderStyle_None** (**0**) or **BorderStyle_Single** (**1**)
  - **Control** class constant **BorderStyle_3DRaised** (**3**) or **BorderStyle_3DSunken** (**2**)
- The ability to define extra spacing in the border area to the left, top, right, and bottom positions.
- The ability to define how it is aligned to its parent container (that is, none, top, bottom, left, right, all vertical, or all horizontal). For details, see the form example in the image in the **JadeDockBase** class **alignContainer** property and the **JadeDockBase** class **drawGrip** property.
- The ability to define whether a grip bar is drawn on the control. The grip is drawn vertically, unless the control or its parent is aligned left or right, in which case it is drawn horizontally. The grip bar provides the user with a visible area on which to click for dragging.
- The ability to define whether the control can become a floating window, and if so, what type of floating window (that is, one that can or cannot be closed). The control must also have visible children before it can be floated or dragged. If the control is allowed to become a floating window, by definition it could also be dragged and docked into another window.
- The caption displayed in its floating window.
- The ability to dock the control into any docking position (with one exception), regardless of the value of its **alignContainer** property. The docking control then adopts the new **alignContainer** property value implied by the docking mode. For details, see the **alignContainer** property.
- The ability to define whether resize bars can be added to the control. Resize bars allow the user to drag that bar to increase or decrease the size of the control. A resize bar is displayed on the:
  - Right of the control if all of the controls that border the control on the right and that share the same parent know how to handle a user resize and they fill the entire right edge. (For details, see the **alignChildren** property.)
  - Bottom of the control for most cases if all of the controls bordering that control on the bottom edge and that share the same parent know how to handle a user resize and they fill the entire edge. (For details, see the **JadeDockBase** class **alignContainer** property.)
- The ability to define whether the children of the control are aligned. The possible values for the **alignChildren** property of the **JadeDockBar** control are **AlignChildren_None**, **AlignChildren_AllHorizontal**, **AlignChildren_AllVertical**, and **AlignChildren_Auto**.

  Automatic alignment means that the control repositions its children in the order of the **tabIndex** property, to achieve the minimum size that is required. The control is also automatically sized to show all of the children.

  Where the alignment of a child control within a container (using the **alignContainer** property) conflicts with the how the container specifies the child control should be aligned (using the **alignChildren** property), the **alignContainer** specification takes precedence.
- The ability to define the number of pixels between each control horizontally and vertically when automatic child alignment is on.

- A **JadeDockBar** control with a non-zero value of the **alignChildren** property does not need a **JadeDockContainer** control as its parent. It can be floated and docked by itself. However, to be able to re-dock the control after floating, it must have a target **Form**, **Frame**, or **JadeDockContainer** control that will accept its presence.

  Note If the control is a **JadeDockBar** control with an **AlignChildren_None** (**0**) value of the **alignChildren** property and it is being docked at the top, left, bottom, or right position of a **Form** or **Frame** control, it requires a **JadeDockContainer** control as its parent. Such a control is created (as a JADE object by cloning the original **JadeDockContainer** control parent of the dock bar or by creating a new one), unless the original **JadeDockContainer** is available as a hidden control (with the **visible** property set to **false**). The **JadeDockContainer** control is aligned as required and assigned as a child of the window into which it is being docked. The **JadeDockBar** control is made a child of the **JadeDockContainer** control.

  The **parentAspect** property is ignored for a **JadeDockBar** control except when the container has the **alignChildren** property set to **AlignChildren_None** (**0**). This setting means that it is your responsibility to position the children. All other settings cause the children to be automatically positioned.

- The ability to programmatically float the control.
- The ability to programmatically determine whether the control is floating.
- The ability to programmatically determine the floating position of the floating form.

For a summary of the constants, properties, and event methods defined in the **JadeDockBar** class, see "JadeDockBar Class Constants", "JadeDockBar Properties", and "JadeDockBar Events", in the following subsections. See also the **JadeDockBase** and **JadeDockContainer** control classes.

JadeDockBar Class Constants

The constants provided by the **JadeDockBar** class are listed in the following table.

Class Constant	Value	Class Constant	Value
AlignChildren_AllHorizontal	1	AlignChildren_AllVertical	2
AlignChildren_Auto	3	AlignChildren_None	0

JadeDockBar Properties

The properties defined in the **JadeDockBar** class are summarized in the following table.

Property	Description
alignChildren	Controls the alignment of the children within the control
autoSpacingX	Determines the horizontal spacing (in pixels) between each child in the control
autoSpacingY	Determines the vertical spacing (in pixels) between each child in the control

For details, see "Window, Form, and Control Properties", later in this document.

JadeDockBar Events

The event methods defined in the **JadeDockBar** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
docked	When the user has caused a control to be docked
dragDrop	When a dragged window is dropped over a control belonging to the same application
dragOver	For each control of the application over which a window is dragged
floated	When the user has caused a control to float
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
queryDock	When the user drags a docking control into a valid docking position accepted by the **JadeDockContainer** class **allowDocking** property of the window
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call
userResize	When the user drags the resize bar of the dock control to a new position or resizes a floating window

For details, see "Window, Form, and Control Events", later in this document.

JadeDockBase Class

The **JadeDockBase** control is an abstract superclass of the **JadeDockBar** and **JadeDockContainer** classes. Most of the functionality for the two control subclasses is defined in the **JadeDockBase** class. The two docking subclasses are very similar in operation and differ mostly in regard to the default values for the properties, the way that the children are aligned, and whether docking is allowed within the control subclass.

The control subclasses are as follows.

- A **JadeDockBar** control is a container for other standard JADE controls and it defines the logical unit that can be dragged and floated.

- A **JadeDockContainer** control is a container for **JadeDockBar** controls that are not aligned to their container but can also be used in many other circumstances. This control can also be dragged and floated.

Note As docking and floating features are not available in Web forms, you should treat docking controls in the same way as a **Frame** control.

You can use control docking if you want to drag a docking control (and all of its children) to a new position on the form or to cause that control to float in an independent floating window.

The horizontal and vertical minimum and maximum sizes of controls are always honored. This can mean that the client area of the parent may not be filled or the controls may not all fit the client area if all of the children have the horizontal and vertical maximum and minimum sizes set. If at least one child does *not* have a maximum or minimum size set, the filling of the client area of the parent occurs.

Docking controls enable you to have multiple **StatusLine** controls, multiple **Frame** controls aligned to the top or bottom of the form, to align all controls that use the space remaining on the parent, and to align all multiple controls in the same parent so that they share the space remaining after other siblings have been aligned.

Although docking and floating functionality is usually associated with the painter and tool-type actions, you can utilize this functionality in many other situations.

For a summary of the constants, properties, and methods defined in the **JadeDockBase** class, see "JadeDockBase Class Constants", "JadeDockBase Properties", and "JadeDockBase Methods", in the following subsections. See also "Floating a Docking Control" and "Docking a Control", later in this section.

JadeDockBase Class Constants

The constants provided by the **JadeDockBase** class are listed in the following table.

Class Constant	Value	Class Constant	Value
AlignContainer_AllHorizontal	5	AlignContainer_AllVertical	6
AlignContainer_Bottom	2	AlignContainer_Left	3
AlignContainer_None	0	AlignContainer_Right	4
AlignContainer_Top	1	DrawGripBar_Double	2
DrawGripBar_None	0	DrawGripBar_Single	1
FloatingStyle_Close	1	FloatingStyle_NoClose	2
FloatingStyle_None	0		

JadeDockBase Properties

The properties defined in the **JadeDockBase** class are summarized in the following table.

Property	Description
alignContainer	Controls how the control is aligned within its parent container so that the control automatically resizes with its container
borderStyle	Contains the border style for the control
borderHeightBottom	Indicates how many extra pixels are drawn in the border area at the bottom of the control
borderHeightTop	Indicates how many extra pixels are drawn in the border area at the top of the control
borderWidthLeft	Indicates how many extra pixels are drawn in the border area at the left of the control

Property	Description
borderWidthRight	Indicates how many extra pixels are drawn in the border area at the right of the control
caption	Determines the caption of the floating form parent of the control
drawGrip	Indicates whether a grip bar is drawn in the border area of the control
floatingStyle	Determines whether the control can be floated and dragged
maximumHeight	Specifies the maximum height of the control in pixels
maximumWidth	Specifies the maximum width of the control in pixels
minimumHeight	Specifies the minimum height of the control in pixels
minimumWidth	Specifies the minimum width of the control in pixels
showResizeBar	Defines whether resize bars can be added to the control

For details, see "Window, Form, and Control Properties", later in this document.

JadeDockBase Methods

The methods defined in the **JadeDockBase** class are summarized in the following table.

Method	Description
centreWindow	Centers the docking control
float	Causes the docking control to be floated by creating a floating form at the specified screen position
getFloatingPosition	Returns the most recent screen position and size of the floating form on which the control resides
isFloating	Returns whether the dock control is the first child of a floating form

For details, see "Window, Form, and Control Methods", later in this document.

Floating a Docking Control

To float a docking control

1. Left-click the mouse on any part of the docking control.
2. Drag the control to a non-docking position (indicated by a double dotted border on the dragging rectangle).
3. Release the mouse.

Alternatively, you can also float a control by clicking on the docking control, dragging the control while holding down the Ctrl or Shift key, and then releasing the mouse. The Ctrl key causes the floating window to adopt its current horizontal size and the Shift key causes the floating window to adopt the smallest possible horizontal size.

The **JadeDockBase** class **drawGrip** property optionally displays a grip bar on the control so that it is easier for the user to click on the control. It can also indicate to the user that the control can be dragged.

Note Pressing the Esc key before releasing the mouse abandons the dragging process so that docking or floating does not occur.

A docking control can be floated only if its **floatingStyle** property is not **FloatingStyle_None** (**0**) and it has visible children. If the floating style is **FloatingStyle_None** (**0**), the control is then effectively just another container with additional alignment capabilities.

Floating a docking control causes the following actions.

- A new form is created at the position and size shown by the last dragging rectangle. This form has no associated JADE object. The position is modified only if it is necessary to ensure that the top and left position of the form is on the screen.
- Like the **Form** class **setFormParent** method, the form is created as a child of the original form. The floating form therefore always sits above the JADE form on the screen. Minimizing the JADE form also minimizes the floating form. No entry is displayed on the taskbar for the form.
- The caption for the docking control is used as the title of the form. The caption line of the form is slightly smaller than that of a standard JADE form.
- The docking control becomes a child of this form. The control does not display any border area (including the grip bar) while it is floating. However, any children of the control are displayed normally.
- The **floatingStyle** property of the **JadeDockBase** class controls whether the form has a **Close** button.
- If the parent of the control is a **JadeDockContainer** control and the control that is being floated is the only visible child, the parent **JadeDockContainer** control is made invisible.
- As the **Control** class **parent** property of the docking control is a child of the floating form and has no JADE object associated with it, it is set to **null**. However, the **Control** class **form** property remains set to the original JADE form. Use the **JadeDockBase** class **isFloating** method of the docking control to determine whether the control is on a floating form.
- The position and size of the controls on the form are re-evaluated for the effects on their alignments caused by the removal of the control from the form.
- The **floated** event method of the docking control is called when the floating process has completed, enabling you to make the required programmatic adjustment to the JADE form.

Floating a docking control causes the form to re-evaluate the alignment of controls affected by the control that is being removed from the form. Floating a control does *not* cause the form size to be changed or non-aligned controls to be repositioned.

The **userResize** event method is called on a floating **JadeDockBar** or **JadeDockContainer** control if a user resizes a floating form (**JadeDockBase** control). If you want to differentiate between the **userResize** event being called when the user drags the resize bar of a control and when the user has resized the floating window, call the **isFloating** method.

To handle the removal of the docking control from the form, you should place other non-aligned controls on a **Frame** control that has the **alignContainer** property set to **AlignContainer_All** (**5**) and that is a sibling of the docking control. The **alignContainer** property automatically handles many floating and docking requirements. (For details, see the **alignContainer** property.)

If the form is an MDI frame, by default, the MDI client window that hosts the MDI child forms is automatically repositioned and resized (unless **Form**::**moveMdiClient** has been called, in which case it is your responsibility to handle this in your logic).

When floating docking controls, you should be aware of the following.

- All controls on the floating form logically belong to the original JADE form so that all logic associated with those controls is executed.

- The floating form passes only the **gotFocus**, **lostFocus**, **keyDown**, **keyUp**, and **keyPress** events to the parent JADE form. All other events on the floating form are ignored.
- The original JADE form and its floating forms are activated when any of those forms is activated.
- Accelerator keys function for controls that reside on the floating form.
- The Tab key cannot be used to switch between the JADE form and the floating form.
- Any number of docking controls can become floating windows.
- You can float a docking control programmatically, by using the **JadeDockBase** class **float** method.
- All JADE forms are constructed with all controls attached to a form in the JADE Painter. Floating a control can be done only by the user or dynamically by logic at run time.
- Clicking on the caption of a floating form enables the floating form to be dragged. Holding the Shift key down causes the size of the dragging rectangle to toggle between the controls being positioned horizontally and vertically after releasing the mouse.
- The size of the floating form is always the minimum that is required to host the docking control in its current alignment. The form can be resized only if the controls within the docking control can be automatically repositioned or if the **alignContainer** property of the docking control is set to **AlignContainer_AllHorizontal** (**5**) or **AlignContainer_AllVertical** (**6**).
- The floating form can only be resized in one direction at a time.
- If the form does not have a **Close** button, the form is destroyed only when the control is re-docked somewhere else or when the JADE form is closed.

  If the floating form has a **Close** button and it is closed, the docking control is transferred back to being an invisible child of the original JADE form and requires a JADE logic action to make the control visible to the user again.

  The dock control that is being hidden receives a **docked** event. To determine this situation, check the value of the **visible** property of the control.
- Use the **JadeDockBase** class **getFloatingPosition** method to obtain the most recent size and position of the floating form.
- You can re-dock floating controls by using logic to change the parent of the control parent back to a window of the JADE form.
- A floating **JadeDockContainer** control can also be docked; that is, another docking control could be docked into the floating window if the floating dock control allows such a docking.
- Before a docking container can be floated, it must have visible children.

Docking a Control

Docking a control changes the parent of the control and it may alter the alignment of the control to its parent.

To dock a docking control

1. Left-click the mouse on any part of the docking control.
2. Drag the control to a docking position (indicated by a single-line border on the dragging rectangle).
3. Release the mouse.

A docking rectangle is drawn in the approximate docking position that will result when the mouse is released. Holding down the Ctrl or Shift key ensures that a floating window will result. The Ctrl key causes the floating window to adopt its current horizontal size and the Shift key causes the floating window to adopt the smallest horizontal size possible.

The **JadeDockBase** control provides a **drawGrip** property that optionally displays a grip bar on dock controls to make it easier for the user to click on the control and possibly indicating to the user that the control can be dragged.

Notes Pressing the Esc key before releasing the mouse abandons the dragging process and no docking or floating results.

The only **Window** subclasses that accept docking are **Form**, **Frame**, and **JadeDockContainer** classes. The **allowDocking** property of these classes defines the type of docking that the window allows.

When a user drags a control around on the form, the process drills down through the windows under the mouse position to assess whether the windows are possible docking targets and whether that type of docking is allowed (for example, docking on the left of the window). This occurs even if non-eligible children cover the window.

If an eligible window is located, the **queryDock** event method is called for the docking control that is being dragged. That event passes the window with which docking is being requested and the type of docking. The **queryDock** event method must return whether the docking is accepted or rejected. If the method is not implemented, docking is accepted by default. The **queryDock** method allows you stricter control over what control can be actually docked where. The dragging rectangle is drawn in a way to indicate docking is possible only when the **queryDock** method returns **true**. The **queryDock** event method is called only once for each potential docking candidate and docking position combination during a dragging episode.

When a user drags a control and releases the mouse in a docking position, the following occurs.

1. The parent of the control is changed to the window that is accepting the docking.

2. The value of the **alignContainer** property of the control that is being docked is changed to the docking type.

3. If the control is the only visible child of a floating form, the floating form is destroyed.

4. If the control is the only visible child of a **JadeDockContainer** control, that container is hidden.

5. The position and size of the controls on the form are re-evaluated for the effects on their alignments caused by the removal of the control from the form.

   Note You can dock a **JadeDockBar** or **JadeDockContainer** control into any docking position (with one exception), regardless of the value of its **alignContainer** property. The docking control then adopts the new **alignContainer** property value implied by the docking mode. The exception is that a **JadeDockBar** control that has its **alignContainer** property set to **AlignContainer_None** cannot be docked **AlignContainer_AllHorizontal** or **AlignContainer_AllVertical**.

   A **JadeDockBar** control whose **alignContainer** property is set to **AlignContainer_None** is mostly useful as a toolbar sitting on a **JadeDockContainer** control as its parent and when docked somewhere else, it remains at its current size or the size derived from its children if the value of the **alignChildren** property is set to **AlignChildren_Auto**. Even when dragged and docked left, right, top, or bottom, it requires **JadeDockContainer** control as its parent. If it is not docked into a **JadeDockContainer** control, it clones its original **JadeDockContainer** parent or creates a new **JadeDockContainer** control if it did not have one.

   In addition, if the **JadeDockBar** control **alignChildren** property is not set to **AlignChildren_Auto** and the width of the control is greater that its height and the control is dragged to a left or right docking position (and it is not currently docked left or right), the values of **width** and **height** properties are exchanged when considering the docking position.

Similarly, if the height of the control is greater than its width and the control is dragged to a top or bottom docking position (and it is not currently docked top or bottom), the values of **width** and **height** properties are exchanged when considering the docking position. This is necessary because the control does not know how to resize itself. For example, if the control were docked at the top and then dragged to the left position and these exchanges were not done, the control would retain its current width, stretch its height, and therefore probably use all of the space in its parent.

6. The **docked** event method of the **JadeDockBar** or **JadeDockContainer** control that is being docked is called on completion of the process, allowing any JADE logic adjustments that may be required.

Floating a docking control causes the form to re-evaluate the alignment of controls affected by the control being removed from the form. Floating a control does *not* cause the form size to be changed or a non-aligned control to be repositioned.

To handle the removal of the docking control from the form, you should place other controls on a **Frame** control that is aligned ‘all’ and that is a sibling of the docking control. For details, see the **alignContainer** property.

If the form is an MDI frame, the MDI client window that hosts the MDI child forms will be automatically repositioned and resized by default (unless **Form**::**moveMdiClient** has been called, in which case it is your responsibility).

The rules for displaying a resize bar to the right or on the bottom of **JadeDockBar** and **JadeDockContainer** controls when the value of the **showResizeBar** property is **true** are as follows.

- A resize bar is displayed on the right of the control if all of the controls that border the control on the right and that share the same parent know how to handle a user resize and they fill the entire right edge. The specific cases handled are:

  - The value of the **alignContainer** property is **AlignContainer_AllVertical** and there is another control with the same parent directly to its right (this could be a **JadeDockBar**, **JadeDockContainer**, or a **Frame** control) and whose **alignContainer** property is also set to **AlignContainer_AllVertical**.

  - The control is a **JadeDockBar** control, the value of the **alignContainer** property is **AlignContainer_Left**, the value of its **alignChildren** property is not set to **AlignChildren_Auto** (otherwise the control automatically resizes), and there is another **JadeDockBar** control with the same parent directly to its right with the value of the **alignContainer** property also set to **AlignContainer_Left** and its **alignChildren** property also not set to **AlignChildren_Auto**.

    Note A left-aligned **JadeDockContainer** control cannot do this, because it always resizes the control to the required minimum width.

  - The control is a **JadeDockBar**, the value of the **alignContainer** property is **AlignContainer_Left**, the value of its **alignChildren** property is not **AlignChildren_Auto** (otherwise the control automatically resizes), and all controls bordering the right edge with the same parent have the value of the **alignContainer** property set to **AlignContainer_Top**, **AlignContainer_Bottom**, **AlignContainer_AllHorizontal**, or **AlignContainer_AllVertica**l and there is at least one **AlignContainer_AllHorizontal** or **AlignContainer_AllVertical** aligned control.

    Note A **StatusLine** control is aligned **AlignContainer_Bottom** by default.

- A resize bar is displayed on the bottom of the control for most cases if all of the controls bordering that control on the bottom edge and that share the same parent know how to handle a user resize and they fill the entire edge. The specific cases handled are:

  - The value of the **alignContainer** property is **AlignContainer_AllHorizontal** and there is another control with the same parent directly below the control (this could be a **JadeDockBar**, **JadeDockContainer**, or a **Frame** control) and whose **alignContainer** property is also set to **AlignContainer_AllHorizontal**.

- The control is a **JadeDockBar** control, the value of the **alignContainer** property is **AlignContainer_Top**, the value of its **alignChildren** property is not set to **AlignChildren_Auto** (otherwise the control automatically resizes), and there is another **JadeDockBar** control with the same parent directly below it with the value of its **alignContainer** property also set to **AlignContainer_Top** and its **alignChildren** property also not set to **AlignChildren_Auto**.

  Note A top-aligned **JadeDockContainer** control cannot do this, because it always resizes the control to the required minimum height.

- The control is a **JadeDockBar**, the value of the **alignContainer** property is **AlignContainer_Top**, the value of its **alignChildren** property is not **AlignChildren_Auto** (otherwise the control automatically resizes), and all of the controls bordering the bottom edge with the same parent with the values of the **alignContainer** property also set to **AlignContainer_Left**, **AlignContainer_Right**, **AlignContainer_AllHorizontal**, or **AlignContainer_AllVertical** and there is at least one **AlignContainer_AllHorizontal** or **AlignContainer_AllVertical** aligned control.

Note To restore the form view without a resize bar, set the value of the **showResizeBar** property to **false** in the JADE Painter or in your JADE code.

When using docked controls, you should be aware of the following points.

- A dock container must have visible children before it can be dragged.
- A dock control can be docked only in its original parent form. It cannot be docked into another form because the logic associated with the control has been compiled against the original form.
- A control can be moved from a floating window by changing its parent in your JADE code.
- A floating **JadeDockContainer** control is also a candidate for docking; that is, another docking control can be docked into the floating window if the floating dock control allows such a docking.
- Unless you set the appropriate values of the **JadeDockContainer**, **Frame**, or **Form** class **allowDocking** property for containers, a floated control does not have any suitable docking position available.
- A docking control that is aligned to its parent by using the **alignContainer** property will not scroll and it remains in place in its parent when the scroll bar of the parent is shifted. The control therefore remains visible and unchanged when the scroll bar of the parent is adjusted.
- Parent windows are evaluated as docking targets if their children are not valid targets, they do not have the appropriate **allowDocking** permission, or if the **queryDock** event method request of the child is rejected.

JadeDockContainer Class

The **JadeDockContainer** control class is a subclass of the **JadeDockBase** class and is intended as a container for **JadeDockBar** controls. The **JadeDockContainer** control provides functionality that is very similar to that of a **JadeDockBar** control, with the following differences.

- Children of the **JadeDockContainer** control that have no **alignContainer** property setting are automatically positioned into rows if the control is aligned horizontally or into columns if the control is aligned vertically.

  The children (which are usually **JadeDockBar** controls) are shifted if they overlap or if the container can shrink the number of rows or columns that are required.
- The spacing between the children cannot be defined.
- The control allows docking, depending on the value of its **allowDocking** property.

- The control allows docking when floating, depending on the value of its **allowDocking** property.
- The control has different initial values for the **JadeDockBase** class **floatingStyle**, **drawGrip**, and **alignContainer** properties.
- The **JadeDockContainer** control is hidden when the last visible child of the control is removed, floated, or docked elsewhere.

  The control is reused and reinstated if a previous child **JadeDockBar** control that has no **alignContainer** property setting is docked into a top, bottom, left, or right position again. The container adopts the **alignContainer** property value implied by the docking and the **JadeDockBar** is made a child of the container.

For more details, see the **JadeDockBar** class. For examples of docking containers, see "Multiple Group Toolbar Example" and "Multiple Group Toolbar on a Non-MDI Form Example", and "Using Align All with Multiple Panes", under the **allowDocking** property.

For a summary of the property and event methods defined in the **JadeDockContainer** class, see "JadeDockContainer Property" and "JadeDockContainer Events", in the following subsections. See also the **JadeDockBase** control class.

JadeDockContainer Property

The property defined in the **JadeDockContainer** class is summarized in the following table.

Property	Description
allowDocking	Controls the type of docking that is allowed by the window

For details, see "Window, Form, and Control Properties", later in this document.

JadeDockContainer Events

The event methods defined in the **JadeDockContainer** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
docked	When the user has caused a control to be docked
dragDrop	When a dragged window is dropped over a control belonging to the same application
dragOver	For each control of the application over which a window is dragged
floated	When the user has caused a control to float
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer

Event	Occurs…
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
queryDock	When the user drags a docking control into a valid docking position accepted by the **JadeDockContainer** class **allowDocking** property of the window
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call
userResize	When the user drags the resize bar of the dock control to a new position or resizes a floating window

For details, see "Window, Form, and Control Events", later in this document.

JadeDotNetVisualComponent Class

The **JadeDotNetVisualComponent** class is the superclass for classes that are proxies for GUI .NET classes; that is, .NET controls. This enables you to use existing third-party functions such as highly specialized controls within your JADE applications.

The corresponding .NET object is created in one of three ways, as follows.

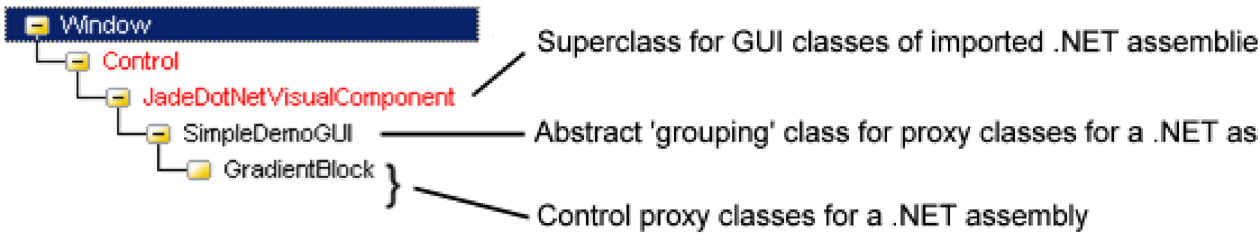
- When the control is dropped onto a form in the JADE Painter
- When a form (with the .NET control) is loaded at run time
- When the **addControl** method of the **Form** class is used to add a dynamically created control

Some controls include a *designer*, which is usually a form that enables you to set the properties of the control. These are particularly useful when the control is made up of a number of sub-controls or when properties have dependencies on other properties. The first time a control with a designer is dropped onto a form in the JADE Painter, the designer form is displayed. When the designer form is closed, the control modified by the designer is shown on the form being painted.

When a control with a designer has been displayed in the JADE Painter, the designer can usually be reactivated by using the context menu. The context menu in the JADE Painter can include up to 10 extra entries corresponding to the first 10 available *designer verbs* when it is activated over a .NET control. These are options defined in the designer that perform various designer options, including re-activating the designer, or altering the control in some way.

Transparent sibling controls are always painted before a **JadeDotNetVisualComponent** control, regardless of their **zOrder** settings. It is not possible to handle the painting of transparent controls in the correct **zOrder** when some controls are directly painted by JADE and others are painted by Windows separately.

When you import a .NET assembly that contains controls into JADE, an abstract class of the specified assembly name is created as a subclass of the **JadeDotNetVisualComponent** class. This abstract class becomes the superclass for all of the control classes that are subsequently generated, corresponding to controls in the imported .NET assembly, as shown in the following image.


Window
Control
JadeDotNetVisualComponent
SimpleDemoGUI
GradientBlock
Superclass for GUI classes of imported .NET assemblie
Abstract 'grouping' class for proxy classes for a .NET as
Control proxy classes for a .NET assembly

JadeDotNetVisualComponent subclasses inherit all of the standard properties and methods of the **Window** and **Control** classes, although not all of these inherited properties and methods have meaning to the control. (For a caveat on SVG files when printing **JadeDotNetVisualComponent** controls, see "Portable Printing" under "Printer Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*.)

To refresh a .NET control that has changed but is already imported into JADE, simply import the .NET assembly again and give it the same name that it had previously.

An imported .NET control is added to the **Control** palette of the JADE Painter and cannot be distinguished from standard JADE-supplied controls.

The **JadeDotNetVisualComponent** class is supported on forms defined as Web pages only when running on a Microsoft Internet Explorer browser.

Note In JADE thin client mode, .NET controls run only on the presentation client.

For a summary of the JADE method defined in the **JadeDotNetVisualComponent** class, see "JadeDotNetVisualComponent Class Method", in the following subsection. (Refer to your .NET documentation for details about properties, methods, or constants provided by imported .NET assembly.)

JadeDotNetVisualComponent Class Method

The method defined in the **JadeDotNetVisualComponent** class is summarized in the following table.

Method	Description
createEventNameMap	Defines a mapping between .NET events and JADE methods to be invoked

For details, see "Window, Form, and Control Methods", later in this document.

JadeEditMask Class

The **JadeEditMask** class is a subclass of the **Control** class and inherits all of the properties and methods defined in the **Control** and **Window** classes. The **JadeEditMask** class provides a number of features that improve the types of validation that can be performed as data is entered and the presentation of those data entry fields. (The **TextBox** control provides basic numeric data types for controlling the characters that can be entered as input.)

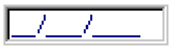
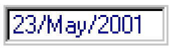
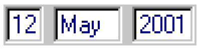
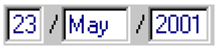
The **JadeEditMask** class provides the following functionality.

- Improves thin client performance due to the removal of validation requirements
- Ability to define an edit mask that dictates the type of data that can be entered by the user for each character position in the text box
- Presentation of input fields of the control using multiple text boxes that make up the logical text value
- Ability to define one or more labels as part of the control
- Ability to include literal values that remain in the text box text as visual prompts to the user
- Automatic thousand-separator insertion into a keyed numeric value
- Display of a prompt character at the position of each character that can be entered

As edit mask facilities are not provided by the Web browser, Web-enabled applications treat the control as another text box.

If a **JadeEditMask** is a cell control, the **JadeEditMask** uses its own mask definition if the cell on which it is activated does not have a mask defined. (Note, however, that if the cell control is activated on a cell that has a defined mask, its pre-defined mask is replaced by the cell's mask and the original **JadeEditMask** mask is lost.)

Examples of the use of the **JadeEditMask** control are listed in the following table.

Example	**Description**
 __/__/____	Empty date field with prompts and date slashes
 23/May/2001	Same field with the date entered
 12 May 2001	Date split into three separate fields
 23 / May / 2001	Date split into three separate fields with slash characters displayed as labels

By default, the control functions as a normal text box when you have not specified an edit mask (by using the **mask** property).

You can define an edit mask:

- In the JADE Painter at application development time, by using the Properties dialog to define a mask of the **JadeEditMask** control selected on the Painter form, specifying the edit mask that you require for the control in the **mask** property on the **Specific** sheet of the dialog
- Dynamically at run time, by constructing it in your JADE application logic

When running in JADE thin client mode and the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

An edit mask is simply a concatenated series of symbols that can define the following.

- The type of data that can be entered at a specific character position.
- The size and position of each field of the layout of the control (a label or text box). If you do not specify size and position values for each field, these are automatically calculated based on the potential size of that field.

  Note The size may be larger than required for two reasons: the prompt character can require more space than the character for which it is prompting and the control must allow space for the widest character. For example, **WWW** takes considerably more space than **iii** if you do not use a monospaced font.

- An indication to start a new label or text box section.
- Literal text to be displayed as the label caption or in the current text box field.
- Lists of characters that can be entered at a character position.
- Actions to be taken; for example, uppercase, lowercase, or numeric separator display.
- Right-aligned fields.
- The numeric range that is permitted for a numeric portion of a text box or label.

When a user presses a key, the control determines the validity of the keyboard action and updates the contents of the control accordingly.

As the **JadeEditMask** control is numeric-aware, you cannot place the cursor into an invalid empty position before or after numeric data. The cursor automatically positions itself appropriately when the focus is gained (if the entire text is not selected). The Home key also positions the cursor at the beginning of the numeric text, while the End key positions the cursor at the end of the numeric value. In addition, pressing the Delete key when the cursor is positioned directly before a thousands separator deletes the character after the separator.

Pressing the Backspace key when the cursor is positioned directly after a thousands separator deletes the character before the separator.

For a summary of the constants, properties, methods, and events defined in the **JadeEditMask** class, see "JadeEditMask Class Constants", "JadeEditMask Properties", "JadeEditMask Methods", and "JadeEditMask Events", later in this document. See also "Right-Aligned or Left-Aligned Text Boxes", in the following subsection.

Right-Aligned or Left-Aligned Text Boxes

The following applies when using right-aligned or left-aligned text boxes of the **JadeEditMask** class.

- If the character generated by the key is invalid according to the rules defined in the **mask** property, a beep is emitted and the character is ignored.
- Selecting text that includes literal text and prompt characters includes those characters in any text saved in the clipboard. Deleting or cutting text removes only the characters that can be entered and leaves the literal characters in place.
- The Backspace key, when positioned to the right of a literal value, acts like a left arrow key press.

For a left-aligned text box, the following applies.

- If the character is valid and the character position has a prompt character displayed or **Insert** mode is off, that character or prompt is overwritten.

- If the entered character is valid, not replacing a prompt, and **Insert** mode is on (the default value), the character is inserted into the text, shifting the following text along one character up to the start of the next edit type, literal, or the end of the text. For example, for the text **'1_/10/2001'**, where the first two characters are a day number followed by a **/** date delimiter, entering a **2** before the **1** results in **21/10/2001** (**_** is the default prompt character).
- Entering a valid character into a field where the next character in the mask is a literal causes the caret to automatically skip to the start of the next character position that can be entered. In the preceding example, entering **21** into the day field of the preceding example automatically positions the caret at the start of the **10/2001** sequence.

  Similarly, using the right arrow key skips literal characters unless the Ctrl or Shift key is also held down (selecting text). The left arrow key steps through the text one character at a time, regardless of type, to allow the user to position the caret at any character position.

  Deleting a character is the reverse of the above procedure. The deletion stops at the first character that has a different mask type. For example, deleting the **2** in **21/10/2001** results in **1_/10/2001**.
- If the caret is placed before a literal character, pressing the key for that literal character causes the caret to move to the next position in the text. The literal character cannot be replaced or removed.
- If all character positions from the caret up to the next literal are empty and the literal that is entered is invalid at the current position, the caret moves to after the literal. For example, if **1/** is entered in a date field of **__/__ /__**, the caret is positioned after the first **/** character.

For a right-aligned text box, the following applies.

- Characters are entered into right-aligned fields to the left of the current caret position instead of to the right, as is the case for a left-aligned field. After any insertion, the caret remains at the same position in the text.
- If the entered character is valid and the character position has a prompt displayed or **Insert** mode is off, that character is overwritten. The caret remains at the same position.
- If the entered character is valid, it is not replacing a prompt, and **Insert** mode is on (the default value), the character is inserted into the text and the characters to the left are shifted to the start of the current edit field or the preceding literal. For example, entering **5** for text **'12/03/_4'** when the caret is positioned to the right of the **4** results in **12/03/45** (**_** is the default prompt character). On completion, the caret remains after the **5**.
- Right and left arrow keys operate as they do in a standard **TextBox** control, moving one character at a time.
- When deleting a character, the deletion stops at the first character that has a different edit mask type. For example, deleting the **5** in **12/03/45** results in **12/03/_4**.
- Because of a Windows limitation, right-aligned text boxes do not scroll horizontally if they are not large enough to display the full text field.

Right-aligned fields are used mostly for numeric-only entry because the caret remains at the same position after entry. Users must use the arrow key to position the caret before a literal segment (that is, the **/** solidus character in the previous examples).

JadeEditMask Class Constants

The constants provided by the **JadeEditMask** class are listed in the following table.

Constant	Integer Value	Description
SelectionStyle_None	0	Text box portion of the edit mask is not selected (default value)
SelectionStyle_Select	1	Text box portion of the edit mask control that receives focus is selected, except when using the mouse to gain focus
SelectionStyle_Select_Always	2	Entire text box portion of the edit mask control that receives focus is selected, regardless of how the focus was achieved when the text box gains focus (by logic, the mouse, or keyboard)

JadeEditMask Properties

The properties defined in the **JadeEditMask** class are summarized in the following table.

Property	Description
autoSize	Specifies whether a control is automatically resized to fit its contents
autoTab	Specifies whether focus is automatically moved to the next control in the tab order of the form, or to the next accessible cell (for a **JadeEditMask** control in a table)
insertMode	Contains the initial setting of the control that indicates whether the control is in **Insert** or **Overwrite** mode
languageId	Specifies the locale associated with the language identifier to be used by the control
mask	Contains a concatenated series of symbols defining the characteristics of the editing requirement and the actions to be taken
promptCharacter	Contains the character to fill the enterable character positions of the text in the text box fields of the control
readOnly	Specifies whether a control is read-only for user input
selectionStyle	Determines whether a text box portion of the edit mask control that receives focus is selected
text	Contains the text contained in the edit area
textUser	Contains the concatenated user text of the text box fields on the control, excluding any literals

For details, see "Window, Form, and Control Properties", later in this document.

JadeEditMask Methods

The methods defined in the **JadeEditMask** class are summarized in the following table.

Method	Description
getTextAsDate	Returns the text value from the **textUser** property converted to a **Date** value

Method	Description
getTextAsDecimal	Returns the text value from the **textUser** property converted to a **Decimal** value
getTextAsInteger	Returns the text value from the **textUser** property converted to an **Integer** value
getTextAsInteger64	Returns the text value from the **textUser** property converted to an **Integer64** value
getTextAsReal	Returns the text value from the **textUser** property converted to a **Real** value
getTextAsTime	Returns the text value from the **textUser** property converted to a **Time** value
isEmpty	Specifies whether there is any data in the character positions in which text can be entered
isValid	Specifies whether the text is valid and complete
setTextFromDate	Sets the text from the **textUser** property value to a **Date** value converted to a **String** in the format of the locale that the control is using
setTextFromDecimal	Sets the text from the **textUser** property value to a **Decimal** value converted to a **String** in the format of the locale that the control is using
setTextFromInteger	Sets the text from the **textUser** property value to an **Integer** value converted to a **String** in the format of the locale that the control is using
setTextFromInteger64	Sets the text from the **textUser** property value to an **Integer64** value converted to a **String** in the format of the locale that the control is using
setTextFromReal	Sets the text from the **textUser** property value to a **Real** value converted to a **String** in the format of the locale that the control is using
setTextFromTime	Sets the text from the **textUser** property value to a **Time** value converted to a **String** in the format of the locale that the control is using

For details, see "Window, Form, and Control Methods", later in this document.

JadeEditMask Events

The events defined in the **JadeEditMask** class are summarized in the following table.

Event	Occurs…
change	When the text contents of a text box have been changed by the user. It does not occur when you change the **text** property from logic (that is, dynamically).
click	When the user presses and then releases the left mouse button.
contextMenu	After the right **mouseUp** event and after the **keyUp** event.
dblClick	When the user presses and releases the left mouse button and then presses and releases it again.
dragDrop	When a dragged window is dropped over a window belonging to the same application.
dragOver	For each form or control of the application over which a window is dragged.
firstChange	When the user makes the first change on the displayed control text or the **isEmpty** status of the control changes. It does not occur when you change the **text** property from logic.
gotFocus	When a control receives the focus.
keyDown	When the user presses a key while the control has the focus.

Event	Occurs…
keyPress	When the user presses and releases an ANSI key.
keyUp	When the user releases a key while the control has the focus.
lostFocus	When a control loses the focus.
mouseDown	When the user presses a mouse button.
mouseEnter	When the user moves the mouse onto a control.
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a form.
mouseMove	When the user moves the mouse.
mouseUp	When the user releases a mouse button.
paint	When part or all of a form is exposed after it has been moved or enlarged.
sysNotify	When a specified JADE system event occurs.
userNotify	When triggered from the JADE Object Manager by a user call.
validate	When the user attempts to shift the focus to another control or form.

For details, see "Window, Form, and Control Events", later in this document.

JadeEditor Class

The **JadeEditor** control class is a subclass of the **JadeTextEdit** class that provides a control for displaying and editing JADE source. The **JadeEditor** class therefore inherits all of the properties, methods, and events defined in the **JadeTextEdit** superclass.

If the **JadeEditor** control is not read-only, the editor receives the return (Enter) key when the editor has focus.

For a summary of the methods defined in the **JadeEditor** class, see "JadeEditor Methods", in the following subsection. For details about integrating the JADE editor into your applications, see "Using the JADE Editor", later in this section.

JadeEditor Methods

The methods defined in the **JadeEditor** class are summarized in the following table.

Method	Description
initializeJadeEditor	Initializes the control for JADE source editing
setCurrentSchema	Sets the color of text in method source for classes, global constants, imported packages, and JADE interfaces for a specific schema

For details, see "Window, Form, and Control Methods", later in this document.

Using the JADE Editor

If you design forms that contain **JadeEditor** controls, you can create the control and initialize it by calling the **initializeJadeEditor** method. If the options set in the control are to be used in your application, set the **useProfile** parameter to **false**.

Note Call the **initializeJadeEditor** method before you call the **setCurrentSchema** method, to ensure that the form displays entities in the correct color.

To set the text editor keyword lists for classes, global constants, imported packages, and JADE interfaces for a specific schema, call the **setCurrentSchema** method so that recognized keywords in method source can be displayed in the appropriate color.

The **JadeEditor** control overrides the keyboard settings of the **JadeTextEdit** control superclass listed in the following table.

Setting	Description
Ctrl+Shift+I	Indents text at the caret position. When the selected text includes at least one end-of-line sequence (for example, CR+LF), the keystroke sequences Ctrl+Shift+I or Tab indent the lines included in the selection. If the selected text does not include an end-of-line sequence, the selected text is replaced by a Tab character.
Ctrl+Shift+U	Removes the indent at the caret position.
Ctrl+K	Deletes the line on which the caret is positioned.
CTRL+N	Inserts a line above the line on which the caret is positioned.

You can dynamically create and manipulate JADE editor controls when a form is running.

JadeMask Class

The **JadeMask** control is a combination picture, button, and label. This control enables you to define images that are automatically presented when the mouse moves over the control and when the *button* is down or disabled.

The **JadeMask** control supports user-specified Web events (that is, you can write a function and use the **Window** class **addWebEventMapping** method to invoke it when a specified event occurs).

In addition, this control provides:

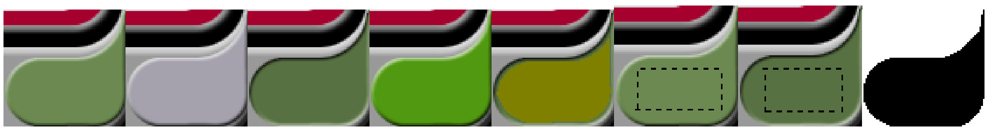
- A mask picture that defines a region within the control (the presentation can create the effect of a button of any shape)
- The ability to return the color of the pixel that the mouse is currently over in the mask

As the **JadeMask** class is a subclass of the **Picture** class, it inherits all of the properties and methods of the **Picture**, **Control**, and **Window** classes.

You can define the control as a single button, a two-state button, or an automatic two-state button based on the **style** property. You can also define the control as the **Cancel** button or the default button.

The **canHaveFocus** property determines whether the control can have focus. When the control has focus, it can be clicked with the mouse, the accelerator defined on the caption can be used, or the space key or return key pressed. If the control is also the **Cancel** button, pressing the Esc key also fires the **click** event.

The following examples show the bitmaps that can be provided.



These images represent the following states, respectively.

- Normal
- Disabled
- Down
- Roll over
- Roll under (when the button is down)
- Focus up
- Focus down
- The mask picture, which is never visible

These images are placed on a background picture so that the button gives the impression that it consists of the swirl shown by the mask in the above examples. The control responds only when the mouse is over that logical area.

A skinned button does not display the focus rectangle if the focus picture is included in the skin. If these pictures are not defined, the **pictureRollOver** state is displayed when the control has focus or the normal control **picture** property state if that **pictureRollOver** is not provided.

When an image containing transparency is displayed in a browser as part of a Web-enabled application, the following conditions must be true for the transparency to be displayed:

- The drawing methods of the **Window** class, such as **drawLine**, are not used on the control
- The value of the **caption** property of the control is null
- The client area of the control is the same size as the image, which is true when the **stretch** property of the control is set to **Stretch_ToControl** (**1**)

When a **JadeMask** control is skinned and at least one of the skin images is 32-bit (which supports transparency), the control is treated as though it is transparent; that is, the control is painted on its parent without the area being erased with the effective value of the **backColor** property. Instead, the parent shows through any transparent areas of the images (for rounded corners, for example). In addition, any semi-transparent parts of the images are anti-aliased with the parent image so that they are displayed with smooth corners over any background color.

For a summary of the constants, properties, method, and events defined in the **JadeMask** class, see "JadeMask Constants", "JadeMask Properties", "JadeMask Method", and "JadeMask Events", in the following subsections.

JadeMask Constants

The constants provided by the **JadeMask** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Alignment_Center_Bottom	8	Alignment_Center_Middle	7
Alignment_Center_Top	6	Alignment_Left_Bottom	2
Alignment_Left_Middle	1	Alignment_Left_Top	0
Alignment_Right_Bottom	5	Alignment_Right_Middle	4
Alignment_Right_Top	3	Style_2State	2
Style_Auto2State	1	Style_Mask_Color	3
Style_Normal	0		

JadeMask Properties

The properties defined in the **JadeMask** class are summarized in the following table.

Property	Description
activeColor	Contains the color to be matched in the mask picture for the definition of the logical area
alignment	Contains the alignment of the caption
canHaveFocus	Specifies whether the control can have focus
cancel	Specifies whether the control is the **Cancel** button
caption	Contains the caption for the mask control
captionHeight	Contains the height of the caption region
captionLeft	Contains the left position of the caption region
captionTop	Contains the top position of the caption region
captionWidth	Contains the width of the caption region
createRegionFromMask	Specifies whether a region is created around the mask picture on a control
default	Specifies whether the control is the default button
disabledForeColor	Determines the color of disabled displayed text in a **JadeMask** control unless the value of this property is zero (**0**)
pictureFocus	Contains the picture that defines the mask for the control when it has focus and it is in the up position
pictureFocusDown	Contains the picture that defines the mask for the control when it has focus and it is in the down position
pictureMask	Contains the picture that defines the mask for the control
pictureRollOver	Contains the picture used when the mouse is over the control in the up position
pictureRollUnder	Contains the picture used when the mouse is over the control in the down position

Property	Description
style	Contains the style of the mask control
value	Specifies whether the button is up (**false**) or down (**true**)

For details, see "Window, Form, and Control Properties", later in this document.

JadeMask Method

The method defined in the **JadeMask** class is summarized in the following table.

Method	Description
currentMaskColor	Returns the color of the pixel in the mask picture corresponding to the last position of the mouse when it was over the control

For details, see "Window, Form, and Control Methods", later in this document.

JadeMask Events

The event methods defined in the **JadeMask** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a control belonging to the same application
dragOver	For each form or control of the application over which a window is dragged
gotFocus	When a control receives the focus
keyDown	When the user presses a key while the control has the focus
keyPress	When the user presses and releases an ANSI key
keyUp	When the user releases a key while the control has the focus
lostFocus	When a control loses the focus
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

JadeRichText Class

The **JadeRichText** control is a type of **TextBox** control that allows the input and display of rich text. As the **JadeRichText** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** and **Window** classes.

Rich text consists of unformatted text and a set of control words and symbols (encoding) that format the text for display. These control words and symbols support a wide variety of formatting (for example, bulleting, fonts, and tables) and the insertion of other objects (for example, bitmaps or Word documents).

Note Like all third-party ActiveX controls for rich text, extensive use of the **JadeRichText** control to store large rich text documents in the JADE database may significantly increase the disk requirements of a user.

Transparent sibling controls are always painted before a **JadeRichText** control, regardless of their **zOrder** settings. It is not possible to handle the painting of transparent controls in the correct **zOrder** when some controls are directly painted by JADE and others are painted by Windows separately.

You can translate captions for the following system forms and dialogs that are related to the **JadeRichText** control by using the **getRootSchemaFormTranslation** method of the **Application** class.

- Find and Replace dialog
- Paragraph formatting dialog
- Insert Table dialog
- Popup menu

The **getRootSchemaFormTranslation** method dynamically loads the captions of all required entities when a form is created and when logic dynamically changes a caption.

The **JadeRichText** control, which supports both Unicode and ANSI formats, applies only to client nodes.

For a summary of the constants, properties, methods, and events defined in the **JadeRichText** class, see "JadeRichText Class Constants", "JadeRichText Properties", "JadeRichText Methods", and "JadeRichText Events", in the following subsections.

For details about designing forms containing **JadeRichText** controls and an example of a method that dynamically creates and formats text in a control at run time, see "Using the JadeRichText Control Class" and "JadeRichText Control Method Example", later in this section. See also Chapter 3, "Using Rich Text Controls on Runtime Forms", of the *JADE Runtime Application Guide*.

JadeRichText Class Constants

The constants provided by the **JadeRichText** class are listed in the following table.

Constant	Value	Constant	Value
Alignment_Center	2	Alignment_Justify	3
Alignment_Left	0	Alignment_Right	1
BulletStyle_Dot	1	BulletStyle_Lowercase	3
BulletStyle_LowercaseRoman	5	BulletStyle_None	0

Constant	Value	Constant	Value
BulletStyle_Number	2	BulletStyle_Uppercase	4
BulletStyle_UppercaseRoman	6	CharacterFormat_AutoColor	-1
CharacterFormat_NotSet	0	CharacterFormat_Set	1
CharacterFormat_Undefined	#80000000	Find_BeginningOfText	0
Find_Default	0	Find_EndOfText	-1
Find_MatchCase	4	Find_SearchBack	2
Find_WholeWord	8	GetLine_PlainText	1
GetLine_RTF	0	LoadFromFile_PlainText	1
LoadFromFile_RTF	0	LoadFromFile_ReplaceAll	0
LoadFromFile_ReplaceSelection	1	LoadFromFile_UnicodeText	2
Load_Append	3	Load_ReplaceAll	1
Load_ReplaceSelection	2	MenuOption_All	#7FFFFFFF
MenuOption_Bullet	#00008000	MenuOption_Copy	#00000008
MenuOption_Custom	#80000000	MenuOption_Cut	#00000004
MenuOption_Find	#00000020	MenuOption_Font	#00000080
MenuOption_InsertObject	#00010000	MenuOption_InsertTable	#00020000
MenuOption_None	0	MenuOption_Object	#00800000
MenuOption_ObjectProperties	#00040000	MenuOption_PageSetup	#00000200
MenuOption_Paragraph	#00000100	MenuOption_Paste	#00000010
MenuOption_Print	#00000400	MenuOption_Redo	#00000001
MenuOption_Replace	#00000040	MenuOption_SepCutCopyPaste	#02000000
MenuOption_SepFindReplace	#04000000	MenuOption_SepFontParaBullet	#08000000
MenuOption_SepInsert	#10000000	MenuOption_SepPrint	#20000000
MenuOption_SepRedoUndo	#01000000	MenuOption_Undo	#00000002
ParagraphFormat_Undefined	#80000000	Redo_Cut	4
Redo_Delete	2	Redo_DragDrop	3
Redo_Paste	5	Redo_Typing	1
Redo_Unknown	0	Replace_ReplaceAll	64
SaveInFile_All	0	SaveInFile_PlainText	1
SaveInFile_RTF	0	SaveInFile_Selection	1
SaveInFile_UnicodeText	2	SelectionStyle_Hide	1
SelectionStyle_Retain	0	SelectionStyle_SelectAll	2
TargetDevice_Printer	1	TargetDevice_Screen	0
TextProtection_Mixed	#80000000	TextProtection_NotSet	0

Constant	Value	Constant	Value
TextProtection_Set	1	Underline_Type_Dash	5.Byte
Underline_Type_DashDot	6.Byte	Underline_Type_DashDotDot	7.Byte
Underline_Type_Dotted	4.Byte	Underline_Type_Invert	254.Byte
Underline_Type_None	0.Byte	Underline_Type_Thick	9.Byte
Underline_Type_Underline	1.Byte	Underline_Type_Wave	8.Byte
Undo_Cut	4	Undo_Delete	2
Undo_DragDrop	3	Undo_Paste	5
Undo_Typing	1	Undo_Unknown	0

JadeRichText Properties

The properties defined in the **JadeRichText** class are summarized in the following table.

Property	Description
acceptTabs	Specifies whether the Tab key inserts a tab character in the control instead of moving the focus to the next control in the tab order
alignment	Contains the alignment of the current paragraph
autoURLDetect	Specifies whether the control automatically formats a URL
bulletIndent	Contains the indentation used when a bullet is applied to the current paragraph
bulletStyle	Contains the bullet style of the current paragraph
contextMenuOptions	Contains the context menu items that are visible when the popup menu is displayed
firstLineIndent	Contains the distance (in pixels) between the left edge of the first line of text in the selected paragraph and the left edge of subsequent lines in the same paragraph
initialContent	Contains the content of the control when it is initialized
leftIndent	Contains the distance (in pixels) between the left edge of the control and the left edge of the current text selection or text added after the insertion point
lineWidth	Contains the maximum width (in pixels) of the current line of text
maxLength	Contains the maximum number of characters that can be entered in the control
readOnly	Specifies whether the contents of the control can be updated
rightIndent	Contains the distance in pixels between the right edge of the control and the right edge of the text that is selected or added at the current insertion point
scrollBars	Contains the scroll bars that can be displayed when text extends beyond the client window co-ordinates of the control
scrollHorzPos	Contains the horizontal position in the virtual text space corresponding to the point shown on the left side of the control
scrollVertPos	Contains the vertical position in the virtual text space corresponding to the point shown at the top of the control

Property	Description
selectionStyle	Specifies whether selected text remains highlighted when the control loses focus
selBackColor	Specifies the background color of the currently selected text
selFontBold	Specifies whether the font of the selected text has the bold attribute applied
selFontItalic	Specifies whether the font of the selected text has the italics attribute applied
selFontName	Contains the name of the font used for the selected text
selFontSize	Contains the size of the font used for the selected text
selFontStrikethru	Specifies whether the font of the selected text has the strikethrough attribute applied
selFontUnderline	Specifies whether the font of the selected text has the underline attribute applied
selFontUnderlineType	Specifies the underline style of the currently selected text
selLength	Contains the length of the selected text
selLink	Specifies whether the currently selected text is a link that will be drawn as a URL
selStart	Contains the starting position of the selected text
selText	Contains the selected text in plain text format
selTextColor	Contains the color of the selected text
selTextRTF	Contains the selected text in rich text format
targetDevice	Contains the device used for "what you see is what you get" (WYSIWYG) printing
text	Contains the text of the control in plain text format
textRTF	Contains the text of the control in rich text format
wantReturn	Specifies whether carriage returns are passed to the rich text control when an enabled and visible default button is defined on the same form
zoom	Contains the factor by which the contents of the control are zoomed

For details, see "Window, Form, and Control Properties", later in this document.

JadeRichText Methods

The methods defined in the **JadeRichText** class are summarized in the following table.

Method	Description
append	Loads the specified text into the control, appending it to the end of the current contents
canPaste_	Returns whether there is content such as text or an image in the Windows clipboard that can be pasted into the **JadeRichText** control
clearUndoBuffer	Clears information from the undo buffer
find	Searches for the specified text within the contents of the control
firstVisibleLine	Returns the first visible line displayed at the top of the control
findReplaceDialog	Opens a find and replace dialog and returns a reference to the dialog

Method	Description
getCharacterFormat	Retrieves the formatting attributes of the selected text
getLine	Returns a string containing the specified line as rich text or plain text
getLineFromCharacterIndex	Returns the number of the line that contains the character at the specified position
getParagraphFormat	Retrieves the common formatting attributes of the paragraph
getRedoAndUndoState	Retrieves whether redo, undo, and pasting actions can be performed and the type of redo and undo operations that can be performed
getScrollRange	Retrieves the scrolling range for the specified scroll bar
getTabStops	Returns the tab stop position values of the control
getTextProtection	Returns the protection state of a specified range of text in the control
insertObject	Inserts a COM object at the current position
insertObjectDialog	Invokes the OLE Insert Object dialog
insertTable	Inserts a table
lineCount	Returns the number of lines of text in the control
load	Loads text into the control
loadFromFile	Loads the contents of the specified file into the control
objectPropertiesDialog	Invokes the OLE Properties dialog
pageMargins	Sets the margins around a printed page
paste_	Pastes the content from the Windows clipboard into the **JadeRichText** control at the current cursor position
print	Outputs the contents of the control to the printer
redo	Reapplies the last edit operation that was undone
replace	Replaces text within the control
saveInFile	Saves the contents of the control to the specified file
setCharacterFormat	Sets common character formatting attributes to the selected text
setParagraphFormat	Sets common character formatting attributes to the paragraph text
setTabStops	Sets the values of the tab stop positions in the control
setTextProtection	Sets the protection state of a specified range of text in the control
undo	Undoes the last edit operation

For details, see "Window, Form, and Control Methods", later in this document.

JadeRichText Events

The event methods defined in the **JadeRichText** class are summarized in the following table.

Event	Occurs…
change	When text within the control has changed.
click	When the user presses and then releases the left mouse button.
contextMenu	When the user right-clicks within the control and the **contextMenuOptions** property has a value of **MenuOption_Custom**. If this method returns **true**, the control displays its built-in menu.
dblClick	When the user presses and releases the left mouse button and then presses and releases it again.
dragDrop	When a dragged window is dropped over a window belonging to the same application.
dragOver	For each window of the application over which a window is dragged.
firstChange	When the contents of the control change.
gotFocus	When a control receives the focus.
keyDown	When the user presses a key while the control has the focus.
keyPress	When the user presses and releases an ANSI key.
keyUp	When the user releases a key while the control has the focus.
linkClicked	When the user clicks on a URL within the text of the **JadeRichText** control.
lostFocus	When a control loses the focus.
mouseDown	When the user presses a mouse button.
mouseEnter	When the user moves the mouse onto a control.
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control.
mouseMove	When the user moves the mouse.
mouseUp	When the user releases a mouse button.
paint	When part or all of a control is exposed.
protected	When the user attempts to alter text that is marked as protected.
scrolled	When the user scrolls.
selChanged	When the selection of text within the control has changed.
sysNotify	When it is triggered when a specified JADE system event occurs.
userNotify	When triggered from the JADE Object Manager by a user call.
windowCreated	When the window for the control is created, so that the control can be initialized when the window for the control is present.

For details, see "Window, Form, and Control Events", later in this document.

Using the JadeRichText Control Class

If you design forms that contain **JadeRichText** controls and you make the context (popup) menu and some or all of its menu items available to users, users can access edit, character formatting, and basic paragraph formatting operations (for example, setting bulleting, fonts, and indents). See also "JadeRichText Control Method Example", later in this section, and Chapter 3, "Using Rich Text Controls on Runtime Forms", in the *JADE Runtime Application Guide*.

You can dynamically create and manipulate rich text in controls when a form is running, by using the functionality provided by the **JadeRichText** class. For details, see the following subsections.

- Fonts in JadeRichText Controls
- Formatting and Selecting Text
- URL Detection
- Initializing the JadeRichText Control
- Clipboard Operations
- File Operations
- Finding and Replacing Text
- Printing Rich Text Control Contents
- Scrolling JadeRichText Controls
- Inserting Objects
- Inserting Tables
- Context Menu
- Unsupported RTF Specification Features

Fonts in JadeRichText Controls

All JADE controls have a set of font properties (that is, **fontName**, **fontSize**, **fontBold**, **fontItalic**, **fontStrikethru**, and **fontUnderline**), which are defined in the **Control** class as they are attributes of the control.

In the **JadeRichText** class, each character can have its own font because the font is an attribute of the actual text. The **getCharacterFormat** and **setCharacterFormat** methods and the **selFontName**, **selFontBold**, **selFontItalic**, **selFontUnderline**, and **selFontStrikethru** properties are used to get and set the font style in the **JadeRichText** control.

The font properties defined in the **Control** class define the initial font of the rich text control and should be used only to set the font at design time (that is, by using the JADE Painter Properties dialog).

The **setCharacterFormat** method sets only the attributes of the currently selected text or the next inserted text at that point. It does not become the default for subsequent inserted or appended text.

The **zoom** property requires a scalable font; for example, the Microsoft Sans Serif font is not a True Type font and therefore it cannot be scaled.

Formatting and Selecting Text

You can apply formatting attributes to and retrieve formatting attributes from both characters and paragraphs in **JadeRichText** controls.

Although individual properties enable you to get and set both paragraph and character formatting attributes, you should consider the number of requests made to the control, particularly when running the JADE application in thin client mode. For example, calling the **getCharacterFormat** method involves one request from the application server but making individual calls from the application server to the presentation client for specific formatting information within the control requires seven requests.

For an example of a method that dynamically sets character and paragraph attributes at run time, see "JadeRichText Control Method Example", later in this section. See also "Formatting Selected Characters" and "Formatting Paragraphs", in the following subsections.

Formatting Selected Characters

The character formatting **getCharacterFormat** and **setCharacterFormat** methods retrieve attributes from and apply attributes to the currently selected text. If no text is selected, these methods apply to the insertion point. The character formatting of the insertion point is applied to newly inserted text if the current selection is empty. When the selection changes, the default formatting changes to match the first character in the new selection.

The **text** and **selText** properties contain plain text for the control. The **textRTF** and **selTextRTF** properties contain all rich text, including all RTF codes. Use the **append** or the **load** method to add additional plain text or RTF text to the current contents of the control. When accessing formatting attributes of selected text and the attributes have mixed values (for example, you have different font sizes in a block of selected text), the appropriate property has a value of **CharacterFormat_Undefined**. You cannot assign an undefined value to an attribute.

Formatting Paragraphs

Paragraph formatting attributes include alignment, tabs, indents, and numbering.

Use the **getParagraphFormat** and **setParagraphFormat** methods to retrieve and set the formatting attributes of the current paragraph. (The *current* paragraph is the paragraph that contains the insertion point.)

Applying a Bullet to a Paragraph

Use the **JadeRichText** class **bulletIndent** property to specify the indentation used when a bullet is applied to the current paragraph and the **bulletStyle** property to specify the style of bullet to apply.

By default, a bullet is not applied to a paragraph.

URL Detection

You can enable automatic detection of a Uniform Resource Locator (URL), by setting the **JadeRichText** class **autoURLDetect** property to **true** in the Properties dialog of the JADE Painter at design time or dynamically at run time. (By default, automatic detection is disabled.)

For an example of a method that dynamically sets the automatic detection of a URL at run time, see "JadeRichText Control Method Example", later in this section.

When enabled:

- The control scans any modified text to determine whether the text matches the format of a URL.
- The control highlights the URL string by underlining it and setting the text color.
- Clicking on a URL generates a **linkClicked** event and double-clicking on a URL activates the link.

Initializing the JadeRichText Control

When you insert or paste the **JadeRichText** control onto a form in the JADE Painter, it is not running in a mode that allows text to be entered in the control.

Use the **initialContent** property, accessed from the Properties dialog in the JADE Painter, to initialize the control. If the text of this property starts with a valid RTF header sequence (for example, **"{\rtf"**), the RTF reader loads that text when the form is run. All other text is displayed as plain text. Although you would normally type the text with which the control is initialized into the Properties dialog **initialContent** property in the JADE Painter, this requires you to have some knowledge of RTF codes (that is, control words and symbols).

You could use the **JadeRichText** control itself to supply the RTF string, by performing the following actions.

1. Create another form (for example, **TempRTF**) containing a **JadeRichText** control and a **Button** control.

2. Add the following code to the **click** event of the **Button** control of the **TempRTF** form.

```
begin
    // rtfControl is value of the name property for the RTF control
    app.copyStringToClipboard(rtfControl.selTextRTF);
end;
```

3. In the JADE Painter, run the second form (**TempRTF**, in this example) and then enter the text, formatting it to meet your requirements.

4. When the appearance of the control meets your requirements, press Ctrl+A to select all text and then click the form button.

   The **click** event then copies the selection to the clipboard as encoded text that contains RTF control words and symbols.

5. In the Properties dialog for the RTF control on your original form, press Ctrl+V in the **initialContent** text box so that the RTF text string, including its control words and symbols, is then displayed.

Clipboard Operations

The **JadeRichText** control supports standard clipboard operations and multiple levels of undo and redo operations, to a maximum of 100 undo or redo operations.

The output parameters of the **getRedoAndUndoState** method indicate whether a paste, redo, or an undo operation can be performed. The **getRedoAndUndoState** method also indicates the type of undo or redo action that can be performed.

You can call the **undo** or **redo** method to undo an action or reapply the last edit operation dynamically at run time.

From release 2020.0.01, you can programmatically paste from the Windows clipboard into a **JadeRichText** control.

- The **canPaste_** method returns whether there is content such as text or an image in the Windows clipboard that can be pasted into the **JadeRichText** control. (You can also obtain this status by calling the **JadeRichText** class **getRedoAndUndoState** method.)

- If there is suitable content such as text or an image in the Windows clipboard, the **paste_** method pastes that content into the **JadeRichText** control at the current cursor position. If the clipboard does not contain suitable content, the method does not result in any change. (This method is equivalent to selecting the **Paste** command in the context menu of the **JadeRichText** control at run time.)

Before you call the **paste_** method, call the **canPaste_** method to confirm there is suitable content available.

File Operations

The **JadeRichText** class **loadFromFile** and **saveInFile** methods enable you to load text from and save text to a file outside of JADE. You should consider the location of these files, especially in thin client mode.

As the RTF specification is not particularly compact, moving large amounts of RTF data between the presentation client and the application server can sometimes be time-consuming.

Finding and Replacing Text

Use the **JadeRichText** class **find** and **replace** methods to search for and replace text dynamically at run time. Alternatively use the **JadeRichText** class **findReplaceDialog** to display a dialog allowing the user to enter search and replacement text.

Use the **getLine** method to return a string containing the specified line as rich text or plain text.

Printing Rich Text Control Contents

Use the **JadeRichText** class **pageMargins** and **print** methods to specify the margins around a printed page and to output the contents of the control to the printer.

You can specify the output name used in the print queue and whether the whole control or only the selected portion of the control is printed. By default, the whole control is printed.

For a caveat on the use of SVG files when printing **JadeRichText** controls, see "Portable Printing" under "Printer Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*.

Scrolling JadeRichText Controls

The **JadeRichText** control automatically supports scrolling when the text to be displayed exceeds the dimensions of the control. The actual display of scroll bars is dependent on the value of the **scrollBars** property.

No horizontal scrolling takes place by default, as the text is wrapped horizontally within the bounds of the control. Use the **lineWidth** property to specify the width of a line. If the line width exceeds the client area width of the control, horizontal scrolling is enabled and horizontal scroll bars are displayed, if required.

Inserting Objects

You can insert any COM object (for example, an Excel spreadsheet, bitmap, or Word document) into the **JadeRichText** control, either by embedding or linking to it.

An embedded object is edited within the **JadeRichText** control itself and a linked object is edited in the source file location. The COM object can be displayed as a view of the object or as an icon. Double-clicking an inserted object activates its server. (In-place activation is not supported.)

If you want to dynamically insert a COM object at run time, call the **insertObject** method to insert an existing object or the **insertObjectDialog** method to bring up the OLE Insert Object dialog that enables you to create or insert COM objects.

You cannot insert text or objects to the right of a table in a **JadeRichText** control.

Inserting Tables

If you want to dynamically insert a table into the rich text control at run time, call the **JadeRichText** class **insertTable** method. (Although JADE enables you to insert a table containing 99 columns or fewer, Microsoft currently restricts you to 32 columns.)

Context Menu

A context (popup) menu built into the **JadeRichText** control gives application users access to edit operations and it supports basic paragraph formatting; for example, setting bulleting, fonts, and indents.

You can use the **contextMenuOptions** property to specify the menu options that are visible to application users. You can also use this property to suppress the popup menu or to indicate that the **JadeRichText** control class **contextMenu** event should be fired instead of displaying the popup menu. Options that are not available are automatically disabled (grayed out).

You can control the menu items that are visible when users right-click in the **JadeRichText** control, by using the **contextMenuOptions** property to store the menu items that are displayed.

If the context menu does not provide you with the functionality that you require, you could use the JADE Painter to create a standard JADE context menu and then use the **contextMenu** event method to display the menu. Menu items that are followed by the points of ellipsis symbol (**...**) access a dialog relevant to that option.

Unsupported RTF Specification Features

Although the **JadeRichText** control can load any document that conforms to the RTF specification, you will not be able to fully edit some documents or they will not be displayed correctly.

The features contained in the following subsections are not supported.

Headers and Footers

JADE does not support headers and footers. Any header or footer contained in a document loaded into the **JadeRichText** control is preserved but is not visible and cannot be updated.

Tables

The display of a table inside a **JadeRichText** control is limited and table dimensions cannot be edited.

Users can insert and delete table rows using the keyboard but they cannot insert or delete columns.

As text does not wrap within a cell, text can overflow into neighboring cells.

Rulers and Toolbars

Most rich text applications (including Word and WordPad) provide a ruler and toolbar to assist in formatting text within the work area. These are generally other controls that work with the rich text control and are not part of it. JADE does not provide ruler and toolbar controls.

Miscellaneous

There are some minor restrictions when displaying rich text inside the **JadeRichText** control. Anything that conforms to the RTF specification is preserved.

Tip To determine the abilities of the **JadeRichText** control, use the Microsoft WordPad application, which displays an RTF document with similar functionality to that provided by the **JadeRichText** control. (If WordPad can do it, the **JadeRichText** control can do it.)

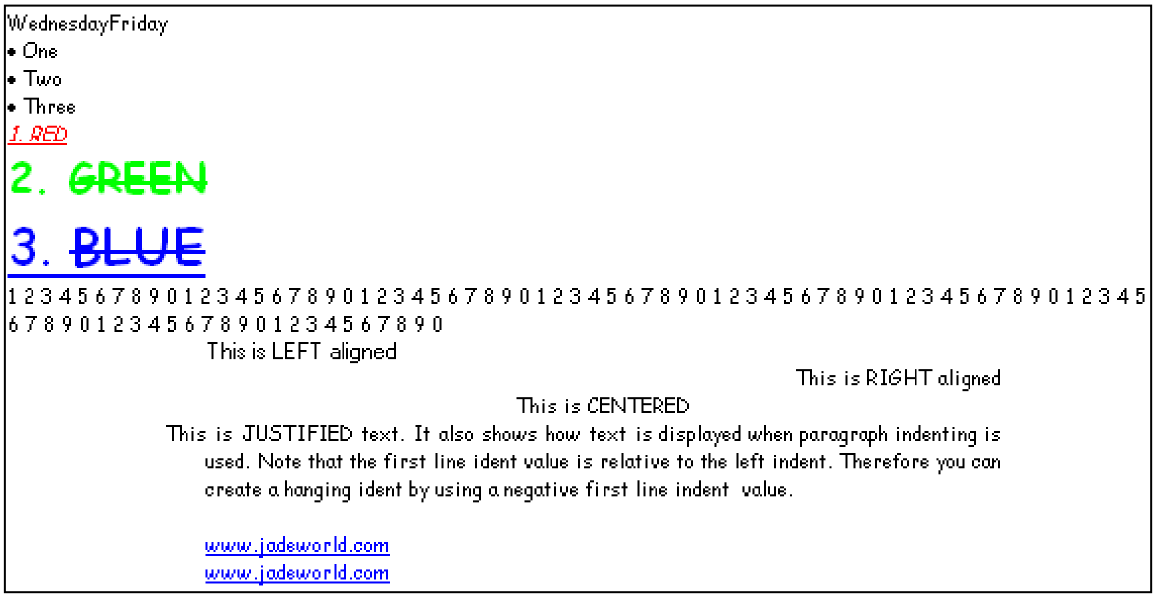
JadeRichText Control Method Example

The following example shows a **JadeRichText** class **click** event method that dynamically creates and manipulates rich text in a control when the **btnRTFByLogic** button is clicked on the running form.

```
btnRTFByLogic_click(btn: Button input) updating;
vars
    str : String;
begin
    // Clear text box
    rtfRichText.text := "";

    // Change some text and end up with one
    // line reading "WednesdayFriday"
    rtfRichText.load("Monday", JadeRichText.Load_Append);
    rtfRichText.load("Tuesday", JadeRichText.Load_Append);
    rtfRichText.load("Wednesday",JadeRichText.Load_ReplaceAll);
    rtfRichText.load("Thursday", JadeRichText.Load_Append);
    rtfRichText.selStart := 9;
    rtfRichText.selLength := 5;
    rtfRichText.load("Fri", JadeRichText.Load_ReplaceSelection);

    // Change the font of ALL text so far, leaving font size,
    // color, and other attributes unchanged
    rtfRichText.setCharacterFormat(false,
                               "Comic Sans MS",
                               JadeRichText.CharacterFormat_Undefined,
                               JadeRichText.CharacterFormat_Undefined,
                               JadeRichText.CharacterFormat_Undefined,
                               JadeRichText.CharacterFormat_Undefined,
                               JadeRichText.CharacterFormat_Undefined,
                               JadeRichText.CharacterFormat_Undefined);
    // Try some plain bullets
    rtfRichText.load(Cr, JadeRichText.Load_Append);
    rtfRichText.bulletStyle := JadeRichText.BulletStyle_Dot;
    rtfRichText.load("One" & Cr, JadeRichText.Load_Append);
    rtfRichText.load("Two" & Cr, JadeRichText.Load_Append);
    rtfRichText.load("Three" & Cr, JadeRichText.Load_Append);

    // Do numbered bullets, with text in its matching color
    rtfRichText.bulletStyle := JadeRichText.BulletStyle_Number;
    rtfRichText.setCharacterFormat(true,        // apply to selection
                null,                           // use current font name
                JadeRichText.CharacterFormat_Undefined,  // font size
                #000000FF,                       // text color = Red
                JadeRichText.CharacterFormat_NotSet,     // not bold
                JadeRichText.CharacterFormat_Set,        // italic
                JadeRichText.CharacterFormat_NotSet,     // no strikethru
                JadeRichText.CharacterFormat_Set);       // underline
    rtfRichText.load("RED" & Cr, JadeRichText.Load_Append);
    rtfRichText.setCharacterFormat(true,
                               null,
                               16,
                               #0000FF00,
                               JadeRichText.CharacterFormat_Set,
                               JadeRichText.CharacterFormat_NotSet,
                               JadeRichText.CharacterFormat_Set,
                               JadeRichText.CharacterFormat_NotSet);
rtfRichText.load("GREEN" & Cr, JadeRichText.Load_Append);
rtfRichText.selTextColor := 256;
rtfRichText.setCharacterFormat(true,
                               null,
                               20,
                               #00FF0000,
                               JadeRichText.CharacterFormat_Undefined,
                               JadeRichText.CharacterFormat_Undefined,
                               JadeRichText.CharacterFormat_Set,
                               JadeRichText.CharacterFormat_Set);
rtfRichText.load("BLUE" & Cr, JadeRichText.Load_Append);

// Create some text that exceeds the control width
rtfRichText.bulletStyle := JadeRichText.BulletStyle_None;
str := "1 2 3 4 5 6 7 8 9 0 ";
str := str & str & str & str & str & str & str & str & str;
rtfRichText.load(str & Cr, JadeRichText.Load_Append);

// Try a different font
rtfRichText.setCharacterFormat(true,
           "MS Sans Serif",  // Not a True Type font so cannot be scaled
           8.25,
           JadeRichText.CharacterFormat_AutoColor,   // System color
           JadeRichText.CharacterFormat_NotSet,
           JadeRichText.CharacterFormat_NotSet,
           JadeRichText.CharacterFormat_NotSet,
           JadeRichText.CharacterFormat_NotSet);

// Different alignments, with left and right indents
rtfRichText.setParagraphFormat(100, 75, 0, JadeRichText.Alignment_Left);
rtfRichText.load("This is LEFT aligned" & Cr, JadeRichText.Load_Append);

rtfRichText.setParagraphFormat(JadeRichText.ParagraphFormat_Undefined,
                               JadeRichText.ParagraphFormat_Undefined,
                               JadeRichText.ParagraphFormat_Undefined,
                               JadeRichText.Alignment_Right);
rtfRichText.load("This is RIGHT aligned" & Cr,JadeRichText.Load_Append);

rtfRichText.setParagraphFormat(JadeRichText.ParagraphFormat_Undefined,
                               JadeRichText.ParagraphFormat_Undefined,
                               JadeRichText.ParagraphFormat_Undefined,
                               JadeRichText.Alignment_Center);
rtfRichText.load("This is CENTERED" & Cr, JadeRichText.Load_Append);

// Justified text with hanging indent
rtfRichText.setParagraphFormat(JadeRichText.ParagraphFormat_Undefined,
                               JadeRichText.ParagraphFormat_Undefined,
                               -20,
                               JadeRichText.Alignment_Justify);
rtfRichText.load("This is JUSTIFIED text. It also shows how text is "
                 & " displayed when paragraph indenting is used. Note "
                 & " that the first line indent value is relative to "
```

```
                         & " the left indent. Therefore you can create a "
                         & " hanging indent by using a negative first line "
                         & " indent value." & Cr,
                         JadeRichText.Load_Append);

        // Turn off hanging indent (or use the setParagraphFormat method)
        rtfRichText.firstLineIndent := 0;
        rtfRichText.alignment := JadeRichText.Alignment_Left;

        // Add a URL that is displayed as a link
        rtfRichText.autoURLDetect := true;
        rtfRichText.append(Cr & "www.jadeworld.com");

        // Replace all 'www.microsoft.com' with 'www.jadeworld.com'
        rtfRichText.replace("www.microsoft.com",
                            "www.jadeworld.com",
                            JadeRichText.Find_BeginningOfText,
                            JadeRichText.Find_EndOfText,
                            JadeRichText.Replace_ReplaceAll);

        // Insert a picture
        rtfRichText.append(Cr); // Make on a new line
        rtfRichText.insertObject("c:\schema\images\jadelogo.bmp", false, false);

        // Scroll the whole control up by 20 pixels
        rtfRichText.scrollVertPos := 20;

    end;
```

The following image shows the result of this method in a **JadeRichText** control on a form when the **btnRTFByLogic** button is clicked.



WednesdayFriday
• One
• Two
• Three
1. RED
2. ~~GREEN~~
3. ~~BLUE~~
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0

This is LEFT aligned

This is RIGHT aligned

This is CENTERED

This is JUSTIFIED text. It also shows how text is displayed when paragraph indenting is used. Note that the first line ident value is relative to the left indent. Therefore you can create a hanging ident by using a negative first line indent value.

www.jadeworld.com
www.jadeworld.com

JadeTextEdit Class

The **JadeTextEdit** control is a type of multiple-line **TextBox** control that allows the display and editing of text such as program source code. The underlying editor engine of the **JadeTextEdit** control is based on an Open Source project called *Scintilla*. For details about Scintilla and the Scintilla-based text editor *SciTE*, see http://scintilla.sourceforge.net/SciTEDoc.html or http://scintilla.sourceforge.net/ScintillaDoc.html.

As the **JadeTextEdit** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** and **Window** classes (for example, the **fontBold**, **fontItalic**, **fontName**, **fontSize**, **fontUnderline**, **foreColor**, **backColor**, **borderStyle**, **enabled**, **height**, **left**, **top**, **visible**, and **width** properties and the **getRegisteredKeys** and **registerKeys** methods defined in the **Control** and **Window** classes).

The **JadeTextEdit** control class provides the **JadeEditor** subclass.

For a summary of the constants, properties, methods, and events defined in the **JadeTextEdit** class, see "JadeTextEdit Class Constants", "JadeTextEdit Properties", "JadeTextEdit Methods", and "JadeTextEdit Events", in the following subsections. See also "Using the JadeTextEdit Control", later in this section.

The **JadeTextEdit** class provides you with:

- Programming language-sensitive text coloring and styling
- Programming language-sensitive text block folding
- Linemarks, to bookmark specific lines in a text edit control as go-to points
- Keystroke binding to editor actions, text insertion, or notification generation
- Load and save text buffer contents in local files
- Support for sharing the text buffer between multiple controls
- Visible indent guides, whitespace, and ends of lines
- Rectangular selection, by pressing the:
  - Alt key when dragging the mouse
  - Shift+Alt+ arrow keys

  You can copy the rectangle selection area to the Windows or the JADE editor clipboard. Pasting the text in the clipboard adds the text at the selected position and starts a new line for lines 2 and greater.

  You can also use this functionality to add tab characters into multiple lines, by pressing Shift+Alt+↓ to select the lines into which the tab characters are added (without selecting any characters) and then pressing Tab to enter tabs on each of the selected lines. Similarly, to remove tabs, press Shift+Alt+↓ to select the lines from which the tabs are removed (without selecting any characters) and then press Backspace, to move text back to the prior tab position.
- If no text is selected and the cursor is over an identifier, all matching occurrences of that identifier (full-word and case-sensitive) are highlighted.

  When text is selected, the result is determined by the value of the **Only highlight whole words matching selection** check box on the **Editor Options** sheet of the Preferences dialog. If the check box is:

  - Unchecked (the default value), any other text matching the case-sensitive selection is also highlighted using the **Additional Selections** color specified on the **Editor** sheet of the Preferences dialog (lime green, by default), unless the selection contains only *space-type* characters; for example, selecting the word **to** highlights any occurrence of **to** in the editor pane.

    - Checked, any selection causes other text matching text in the editor pane to be highlighted if the selection is a full-word identifier and there are other occurrences of that word in the editor pane. This match is case-sensitive and full-word; for example, selecting the word **caption** highlights any other occurrence of the full word **caption**.

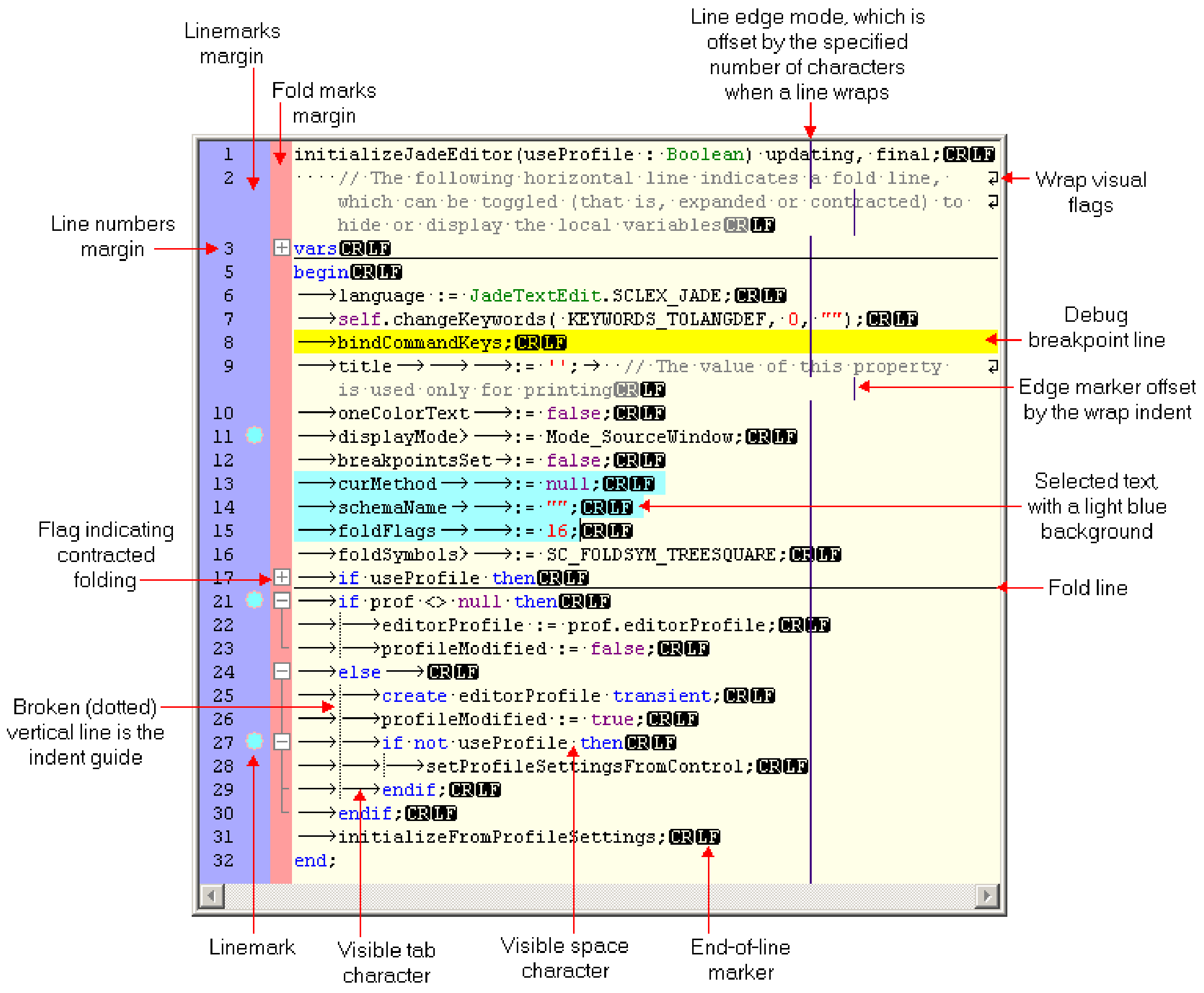
- Parentheses (**()**) and bracket (**[]**) pairs are highlighted when editing a JADE method or schema file when the cursor is positioned before the starting or closing **()** parenthesis or **[]** bracket character, making it quicker to resolve the omission of the closing parenthesis or bracket.

  The back ground of the two parentheses (**()**) and bracket (**[]**) bracket characters is colored using the **Additional Selections** value specified on the **Editor** sheet of the Preferences dialog. The default value is bright green. If the cursor is positioned before a parenthesis or bracket character that does not have a matching starting or closing bracket, the background of the character is highlighted in red. (There is no user preference available for this.)

Notes The **JadeTextEdit** control uses between four and five times the size of the text contained in the control for memory buffer and tables. The maximum size of a file that you can load into the control by using the **loadTextFromFile** method is the smaller of 50M bytes or a tenth of the physical memory.

Although there is no limit on the maximum number of characters in one line of text, only the first 8190 characters are displayed. This limit also applies when wrapping is enabled. The remainder of the line can be edited and the *caret* (insertion point) can be moved into those areas so that new line characters can be inserted.

The following image shows some of the functionality available in text editor controls.


Linemarks margin
Fold marks margin
Line edge mode, which is offset by the specified number of characters when a line wraps
Line numbers margin
Wrap visual flags
Debug breakpoint line
Edge marker offset by the wrap indent
Selected text with a light blue background
Flag indicating contracted folding
Fold line
Broken (dotted) vertical line is the indent guide
Linemark
Visible tab character
Visible space character
End-of-line marker

```
initializeJadeEditor(useProfile : Boolean) updating, final;
    // The following horizontal line indicates a fold line,
    which can be toggled (that is, expanded or contracted) to
    hide or display the local variables
vars
begin
	language := JadeTextEdit.SCLEX_JADE;
	self.changeKeywords( KEYWORDS_TOLANGDEF, 0, "");
	bindCommandKeys;
	title			:= '';	  // The value of this property
	is used only for printing
	oneColorText	:= false;
	displayMode	:= Mode_SourceWindow;
	breakpointsSet	:= false;
	curMethod		:= null;
	schemaName		:= "";
	foldFlags		:= 16;
	foldSymbols	:= SC_FOLDSYM_TREESQUARE;
	if useProfile then
	if prof <> null then
		editorProfile := prof.editorProfile;
		profileModified := false;
	else	
		create editorProfile transient;
		profileModified := true;
		if not useProfile then
			setProfileSettingsFromControl;
		endif;
	endif;
	initializeFromProfileSettings;
end;
```

This image illustrates the following settings that have been defined in an application.

- The current language is JADE. As text is styled using the JADE language settings, the color of words is determined by the type (for example, comments, keywords, numeric literals, global constants, and so on).

  The example in this image uses the following code fragment.

  ```
  language := SCLEX_JADE;
  applySettings;
  ```

- A wrapped line has a line number displayed in margin zero (**0**) only in the first line. Subsequent wrapped lines of the same line are indicated by optional indentation and visual markers, whose positioning you can specify (for example, before or after the wrapped text).

  The example in this image uses the following code fragment.

  ```
  wrapMode := SC_WRAP_WORD;
  wrapIndent := 4;
  wrapVisualFlags := SC_WRAPVISUALFLAG_END;
  ```

- The linemarks in margin **1** serve as bookmarks in the current text editor.

  You can bind editor commands to keystrokes that allow a user to toggle (set or reset) a linemark in the current line (for example, Ctrl+F7) or to move to the next linemark (for example, F7) or to the previous linemark (for example, Shift+F7).

  The example in this image uses the following code fragment.

  ```
  markerMargin := true;
  setLinemarkAttributes(MARKER_JAD_LINEMARK, SC_MARK_ROUNDRECT,
                        Black, LightGreen);
  bindKeyToCommand(J_key_F7+KEYMOD_CTRL, SCI_TOGGLE_JADE_LINEMARK);
  bindKeyToCommand(J_key_F7, SCI_GONEXT_JADE_LINEMARK);
  bindKeyToCommand(J_key_F7+KEYMOD_SHIFT,SCI_GOPRIOR_JADE_LINEMARK);
  ```

  Note Going to a linemark that contains folding on or following that line and before the next marked line expands any folded lines that were hidden (that is, folded).

- Fold marks optionally displayed in margin **2** of the margin enable you to suppress the display of specific text editor lines (for example, if you want to hide the details of an **if** instruction), by:

  - Clicking margin 2 within a fold mark to contract an expanded fold mark (that is, hide the display of lines)
  - Clicking on a collapsed fold symbol to display all lines within that fold
  - Going to a linemark whose following lines contain folds, to expand the folded lines.

    The example in this image uses the following code fragment.

    ```
    folding := true;
    foldFlags := 16;
    foldSymbols := SC_FOLDSYM_TREESQUARE;
    ```

- Whitespace is visible, spaces are indicated by the centered dots, and tabs by right-pointing arrows.

  The example in this image uses the following code fragment.

  ```
  viewWhitespace := SCWS_VISIBLEALWAYS;
  viewEndOfLine := true;
  ```

- The background color of margins, the edge mode line, selected text, background color, and the text edit control window can all be specified to meet your requirements.

  The example in this image uses the following code fragment.

  ```
  self.setStyleAttributes(STYLE_LINENUMBER, "", ATTRIB_NOCHANGE,
                          ATTRIB_NOCHANGE, Purple, ATTRIB_NOCHANGE,
                          ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
  setNamedAttribute("fold.margin.color", Pink);
  selBackColor := LightBlue;
  ```

- You can specify that lines that exceed a specified number of characters are visibly marked.

The example in this image, which uses the following code fragment, uses line mode where a vertical line is drawn after the specified column. The line is offset by the amount of wrap indentation when wrapping is enabled.

```
edgeMode  := SC_EDGE_LINE;
edgeColumn  := 65;
edgeColor  := Black;
```

- The debug breakpoint line is displayed using a linemark. The example in this image uses the following code fragment.

```
setLinemarkAttributes(MARKER_JAD_BREAKPOINT, SC_MARK_BACKGROUND,
                      Yellow, Yellow);
```

For details about the function key and shortcut keys in the JADE editor pane, see "Using Function Keys and Shortcut Keys" under "Using the JADE Development Environment", in Chapter 2 of the *JADE Development Environment User's Guide*.

JadeTextEdit Class Constants

The constants provided by the **JadeTextEdit** class are listed in the following table.

Constant	Value
ADDTXT_ADD	2
ADDTXT_APPEND	3
ADDTXT_INSERT	1
ADDTXT_INSERTREPLACESEL	4
ATTRIB_DEFAULT	-1
ATTRIB_FALSE	0
ATTRIB_NOCHANGE	-2
ATTRIB_TRUE	1
CLIPBUFFER_MAX	9
EVENTTYPE_ALTERREADONLY	#80001002
EVENTTYPE_BOUNDKEY	#80001001
EVENTTYPE_CANCEL	#80001003
EVENTTYPE_CLIPBUFFCHG	#80001005
EVENTTYPE_CLIPBUFFETC	#80001006
EVENTTYPE_OPENFINDREPLACE	#80001007
EVENTTYPE_OPENMACROLIBRARY	#80001008
EVENTTYPE_PLAYSAVEDMACRO	#80001009
EVENTTYPE_SELECTIONSTATE	#80001004
FIND_INTERP_NONE	0
FIND_INTERP_POSIXREGEXPR	3

Constant	Value
FIND_INTERP_REGEXPR	2
FIND_INTERP_UNSLASH	1
FIND_RANGE_ALL	0
FIND_RANGE_CARET	1
FIND_RANGE_SELECTION	2
KEYMOD_ALT	#40000
KEYMOD_CTRL	#20000
KEYMOD_SHIFT	#10000
KEYWORDSET_MAX	9
KEYWORDS_ADD	2
KEYWORDS_DELETE	3
KEYWORDS_SET	1
KEYWORDS_TOLANGDEF	4
KMACRO_GETCOMMANDS	6
KMACRO_GETISRECORDING	8
KMACRO_GETTEMPORARY	4
KMACRO_PLAYTEMPORARY	1
KMACRO_PLAYTEXT	7
KMACRO_RECORDSTART	2
KMACRO_RECORDSTOP	3
KMACRO_SETTEMPORARY	5
KWL_JADE_GLOBALCONSTANTS	5
KWL_JADE_IMPORTEDCLASSES	8
KWL_JADE_INTERFACES	9
KWL_JADE_KEYWORDS	1
KWL_JADE_METHODWORDS	2
KWL_JADE_PACKAGES	7
KWL_JADE_SYSTEMCLASSES	4
KWL_JADE_SYSTEMVARS	3
KWL_JADE_USERCLASSES	6
LMACT_ADD	1
LMACT_ADDATPOSITION	14
LMACT_DELETE	2
LMACT_DELETEALL	3

Constant	Value
LMACT_DELETEBYHANDLE	4
LMACT_GETBITMASK	5
LMACT_GETLINEFROMHANDLE	6
LMACT_GOHANDLE	11
LMACT_GONEXTBYNUMBER	7
LMACT_GONEXTINBITMASK	9
LMACT_GOPRIORBYNUMBER	8
LMACT_GOPRIORINBITMASK	10
LMACT_MOVESINGLETOLINE	12
LMACT_MOVESINGLETOPOSITION	13
LOCAT_CARET	1
LOCAT_MOUSEPOINTER	0
MARKER_JAD_BREAKPOINT	20
MARKER_JAD_DEBUGCURRENT	22
MARKER_JAD_LINEMARK	1
MARKER_MAXMAX	31
MARKER_USERMAX	24
MVCRT_VIEWCARET	1
MVCRT_VIEWSELECTION	2
MVCRT_WORDEND	3
MVCRT_WORDSTART	4
SCE_JAD_BINARYLITERAL	22
SCE_JAD_COMMENT	6
SCE_JAD_COMMENTLINE	7
SCE_JAD_DEFAULT	4
SCE_JAD_DOCTEXT	21
SCE_JAD_DOLLARIDENT	23
SCE_JAD_GLOBALCONST	17
SCE_JAD_IDENTIFIER	11
SCE_JAD_INTERFACE	20
SCE_JAD_KEYWORD	12
SCE_JAD_METHODWORD	13
SCE_JAD_NUMBER	10
SCE_JAD_PACKAGE	18

Constant	Value
SCE_JAD_PACKAGECLASS	19
SCE_JAD_PUNCTUATION	5
SCE_JAD_SINGLECOLOR	0
SCE_JAD_STRING1	8
SCE_JAD_STRING2	9
SCE_JAD_SYSTEMCLASS	15
SCE_JAD_SYSTEMVAR	14
SCE_JAD_USERCLASS	16
SCI_BACKTAB	2328
SCI_COPYTOCLIPBUFF	2961
SCI_FINDAGAIN	2953
SCI_FINDNEXT	2954
SCI_FINDPREV	2955
SCI_GONEXT_JADE_BRKPNT	SCI_MARKERNEXT20
SCI_GONEXT_JADE_LINEMARK	SCI_MARKERNEXT01
SCI_GOPRIOR_JADE_BRKPNT	SCI_MARKERPRIOR20
SCI_GOPRIOR_JADE_LINEMARK	SCI_MARKERPRIOR01
SCI_INSERTLINEABOVE	2956
SCI_LINEDELETE	2338
SCI_MARKERNEXT00	2800
SCI_MARKERNEXT01	2801
SCI_MARKERNEXT20	2820
SCI_MARKERPRIOR00	2850
SCI_MARKERPRIOR01	2851
SCI_MARKERPRIOR20	2870
SCI_NEWLINE	2329
SCI_NULL_COMMAND	0
SCI_OPENFINDREPLACE	2963
SCI_PASTEFROMCLIPBUFF	2962
SCI_TAB	2327
SCI_TOGGLEFOLDERHERE	2951
SCI_TOGGLEMARKER00	2900
SCI_TOGGLEMARKER01	2901
SCI_TOGGLEMARKER20	2920

Constant	Value
SCI_TOGGLEMARKER22	2922
SCI_TOGGLEMARKER24	2924
SCI_TOGGLE_JADE_BREAKPOINT	SCI_TOGGLEMARKER20
SCI_TOGGLE_JADE_DEBUG	SCI_TOGGLEMARKER22
SCI_TOGGLE_JADE_LINEMARK	SCI_TOGGLEMARKER01
SCI_XLATEHEXTOUNICODE	2959
SCI_XLATEUNICODETOHEX	2960
SCI_ZOOMIN	2333
SCI_ZOOMOUT	2334
SCLEX_BASH	62
SCLEX_BATCH	12
SCLEX_CONF	17
SCLEX_CPP	3
SCLEX_CSS	38
SCLEX_DIFF	16
SCLEX_HTML	4
SCLEX_JADE	65
SCLEX_JAVA	65539
SCLEX_JAVASCRIPT	131075
SCLEX_MAKEFILE	11
SCLEX_PERL	6
SCLEX_PROPERTIES	9
SCLEX_PS	42
SCLEX_PYTHON	2
SCLEX_TEXT	1
SCLEX_VB	8
SCLEX_VBSCRIPT	28
SCLEX_XML	5
SCWS_INVISIBLE	0
SCWS_VISIBLEAFTERINDENT	2
SCWS_VISIBLEALWAYS	1
SC_EDGE_BACKGROUND	2
SC_EDGE_LINE	1
SC_EDGE_NONE	0

Constant	Value
SC_EOL_CR	1
SC_EOL_CRLF	0
SC_EOL_LF	2
SC_FOLDSYM_ARROWS	0
SC_FOLDSYM_PLUSMINUS	1
SC_FOLDSYM_TREEROUND	2
SC_FOLDSYM_TREESQUARE	3
SC_INDIC0_MASK	#20
SC_INDIC1_MASK	#40
SC_INDIC2_MASK	#80
SC_INDICS_MASK	#E0
SC_INDIC_BOX	6
SC_INDIC_DIAGONAL	3
SC_INDIC_HIDDEN	5
SC_INDIC_PLAIN	0
SC_INDIC_SQUIGGLE	1
SC_INDIC_STRIKE	4
SC_INDIC_TT	2
SC_MARK_ARROW	2
SC_MARK_ARROWDOWN	6
SC_MARK_ARROWS	24
SC_MARK_BACKGROUND	22
SC_MARK_BOXMINUS	14
SC_MARK_BOXMINUSCONNECTED	15
SC_MARK_BOXPLUS	12
SC_MARK_BOXPLUSCONNECTED	13
SC_MARK_CHARACTER	10000
SC_MARK_CIRCLE	0
SC_MARK_CIRCLEMINUS	20
SC_MARK_CIRCLEMINUSCONNECTED	21
SC_MARK_CIRCLEPLUS	18
SC_MARK_CIRCLEPLUSCONNECTED	19
SC_MARK_DOTDOTDOT	23
SC_MARK_EMPTY	5

Constant	Value
SC_MARK_LCORNER	10
SC_MARK_LCORNERCURVE	16
SC_MARK_MINUS	7
SC_MARK_PLUS	8
SC_MARK_ROUNDRECT	1
SC_MARK_SHORTARROW	4
SC_MARK_SMALLRECT	3
SC_MARK_TCORNER	11
SC_MARK_TCORNERCURVE	17
SC_MARK_VLINE	9
SC_STYLES_MASK	#1F
SC_WRAPVISUALFLAG_END	1
SC_WRAPVISUALFLAG_END_BY_TXT	3
SC_WRAPVISUALFLAG_NONE	0
SC_WRAPVISUALFLAG_START	2
SC_WRAPVISUALFLAG_START_BY_TXT	4
SC_WRAP_NONE	0
SC_WRAP_WORD	1
STYLE_BRACEBAD	35
STYLE_BRACELIGHT	34
STYLE_CONTROLCHAR	36
STYLE_DEFAULT	32
STYLE_INDENTGUIDE	37
STYLE_LINENUMBER	33
STYLE_MAX	127

JadeTextEdit Properties

The properties defined in the **JadeTextEdit** class are summarized in the following table.

Property	Description
canPaste	Specifies whether the clipboard contains text and it can be pasted into the buffer
canRedo	Specifies whether editor actions exist that can be redone
canUndo	Specifies whether editor actions exist that can be undone
currentColumn	Contains the column in which the caret is positioned

Property	Description
currentLine	Contains the line in which the caret is positioned
currentPosition	Contains the character offset at which the caret is positioned
edgeColor	Contains the color of the marker used to show a line has exceeded the **edgeColumn** length
edgeColumn	Contains the number of the column at which the long linemark indicator is displayed
edgeMode	Contains the mode that is used to display long lines in the text editor
endOfLineMode	Contains the type of end-of-line sequence that is used when new lines are inserted
firstVisibleLine	Contains the number of the first line that is visible in the text editor
foldFlags	Contains the flags that influence text folding behavior
foldSymbols	Contains the symbol set used in the fold margin when line folding is enabled
folding	Specifies whether text block folding is enabled in the editor
indentGuides	Specifies whether vertical indentation guidelines are displayed in the editor
indentWidth	Contains the width in characters of the text editor indent
language	Contains the programming language used in the text editor
markerMargin	Specifies whether the marker margin containing linemarks is displayed
modified	Specifies whether the text has been modified
readOnly	Specifies whether the text is read-only for user input
selBackColor	Contains the background color of text selected in the text editor
selForeColor	Contains the color of text selected in the text editor
selLength	Contains the number of characters selected in the text editor
selStart	Contains the starting character offset of selected text in the text editor
selText	Contains the string of the currently selected text in the text editor
tabWidth	Contains the width of a tab in the text editor
text	Contains the text in the text editor
useTabs	Specifies whether tabs are used in the text editor to indent lines to the next indent position
viewEndOfLine	Specifies whether end-of-line characters are displayed in the text editor
viewLineNumbers	Specifies whether line numbers are displayed in the first margin of text editor lines
viewWhitespace	Specifies whether space and tab characters are visible in the text editor
wrapIndent	Contains the number of spaces by which continuation lines of wrapped lines are indented in the text editor
wrapMode	Contains the way in which lines of text that exceed the text editor control width are wrapped
wrapVisualFlags	Contains the way in which visual flags indicating wrapped text are displayed in the text editor
zoom	Contains the zoom factor of the text editor

For details, see "Window, Form, and Control Properties", later in this document.

JadeTextEdit Methods

The methods defined in the **JadeTextEdit** class are summarized in the following table.

Method	Description
addText	Adds the specified text to the contents in the specified manner
applySettings	Applies the application settings to the text editor
bindKeyToCommand	Assigns a key combination to a text editor command
bindKeyToNotification	Assigns a key combination to a notification message
bindKeyToText	Assigns a key combination to the specified text string
changeKeywords	Modifies one of the current keyword lists
clearAllStyles	Clears all text styles defined for the control
colorAs6Hex	Returns a six-character hexadecimal string in the RGB format of the specified color
configureFor_Jade	Performs basic configuration for JADE methods
configureFor_Text	Performs basic configuration for plain text
convertEndOfLines	Replaces all end-of-line characters in the contents with the specified characters
convertIndentWhitespace	Changes the indentation whitespace as specified
copyDefaultToAllStyles	Copies the default text style and all of its attributes to all other text styles
copyToClipboard	Copies the selected text to the clipboard
cutToClipboard	Cuts the selected text from the contents and moves it to the clipboard
doLinemarker	Performs linemark actions (for example, set, go-to, and so on)
emptyUndoBuffer	Clears the undo and redo action list
find	Searches for the specified text in the contents
findAgain	Searches for the text and parameter values reused from the most recent **find**, **findMarkAll**, or **replaceAll** method
findMarkAll	Searches for the specified text and places linemarks where that text is found
getClipBuffer	Returns the contents of the specified editor text buffer
getCoordinates	Returns the coordinates of the requested location (in pixels) relative to the client area
getGlobalSettings	Returns a copy of the global settings table
getLanguageName	Returns the name of the specified programming language number
getLineHeight	Returns the height in pixels of the specified line
getLineStartPosition	Returns the zero-based character offset of the first character in the specified line
getLineText	Returns the text contained in the specified line
getLinemarkLines	Populates an array with each occurrence (position) of the specified linemark

Method	Description
getNamedAttribute	Returns the value of the specified named attribute
getTextLength	Returns the number of characters of text in the contents
getTextRange	Returns the text in the specified range
getToggleKeyStates	Returns the on or off state of the Insert, CAPS LOCK, NUM LOCK, and SCROLL LOCK keys
getWordAt	Returns the word at the specified location (that is, the mouse pointer or the caret position)
initializeAppSettings	Removes all entries from the application settings table
lineCount	Returns the number of lines of text in the contents
lines	Returns the height of the client area in lines
loadTextFromFile	Replaces the contents of the text editor with the contents of the specified file
moveCaret	Moves the caret to the specified position
pasteFromClipboard	Pastes the contents of the clipboard into the text editor buffer at the caret position
recordReplay	Reapplies the last operation that was undone
redo	Reapplies the last operation that was undone
replace	Replaces the most recent find match with the specified replacement text
replaceAll	Searches for all occurrences of specified text in the contents and replaces it with the specified replacement text
restyleText	Recalculates all of the text styling information for the contents using the current language setting
rgb	Returns the red, green, and blue color values as an Integer value
saveTextToFile	Saves the contents to the specified file
selectAll	Selects all of the text in the text editor
setClipBuffer	Sets the specified editor text buffer with specified text
setIndicatorAttributes	Sets type and color attributes for an text editor indicator
setLinemarkAttributes	Sets type and color attributes for text editor linemarks
setLinemarkLines	Adds a linemark to each line in the specified list
setNamedAttribute	Sets the specified named attribute to the specified value
setStyleAttributes	Sets the attributes of a text style
setTextRangeToStyle	Updates the current text styling for each character in the specified range with the specified value
setWordCharactersets	Sets the characters that are treated as words, white space, and punctuation in the text editor
shareDocumentFrom	Specifies the text editor control whose content is shared by this text editor control
undo	Undoes the last edit operation
updateAppSettings	Updates the application **JadeTextEdit** settings table

For details, see "Window, Form, and Control Methods", later in this document.

JadeTextEdit Events

The event methods defined in the **JadeTextEdit** class are summarized in the following table.

Event	Occurs when…
change	Text within the control changes
click	The user presses and then releases the left mouse button
contextMenu	The user right-clicks within the control
dblClick	The user presses and releases the left mouse button and then presses and releases it again
firstChange	The contents of the control change for the first time
gotFocus	The control receives the focus
keyDown	The user presses a key while the control has the focus
keyPress	The user presses and releases an ANSI key
keyUp	The user releases a key while the control has the focus
lostFocus	A control loses the focus
paint	Part or all of a control is exposed
sysNotify	A specified JADE system event occurs
userNotify	A specified user-defined event or bound key notification occurs

For details, see "Window, Form, and Control Events", later in this document.

Using the JadeTextEdit Control

This section contains the following topics.

- Contents of Text Edit Controls
- Navigating Around the Text Editor
  - Using the Mouse within the Editor Text Area
  - Using the Mouse within the Text Editor Margin
  - Using the Mouse within the Fold Margin
  - Using the Keyboard in the Text Editor
- Coloring and Text Styling
- Folding
- Linemarks
- Settings
- Supported Languages
- Unicode and ANSI Considerations

Contents of Text Edit Controls

As illustrated in the image under "**JadeTextEdit** Class", earlier in this section, text edit controls can contain the following.

- Zero to three margins, as follows.
    a. Line number (margin **0**)
    b. Linemark (margin **1**)
    c. Fold mark (margin **2**)
- Text area
- Horizontal and vertical scroll bars, if required

Navigating Around the Text Editor

The caret marks the current position.

The *caret* is a vertical bar displayed before the current character when the insert mode is **true**. Conversely, it is an underline displayed under the current character when the insert mode is **false**.

This section contains the following topics.

- Using the Mouse within the Editor Text Area
- Using the Mouse within the Text Editor Margin
- Using the Mouse within the Fold Margin
- Using the Keyboard in the Text Editor

Using the Mouse within the Editor Text Area

The mouse actions that you can perform in the text area of the **JadeTextEdit** control text editor are listed in the following table.

Mouse Action	Result
Left-click	Moves the caret (insertion point) to the cursor location and cancels the selection
Double-click	Selects the word at the cursor location
Triple-click	Selects the line at the cursor location
Left down and move	Anchors the selection at the down position and extends to follow the cursor location
Right-click	Opens the popup (context) menu
Shift+left-click	Moves the caret (insertion point) to the cursor location and extends the selection to the new location
Alt+left down and move	Anchors a rectangular selection at the down position and extends to follow the cursor location

Using the Mouse within the Line Number Margin

The mouse actions that you can perform within the line number margin of the **JadeTextEdit** control text editor are listed in the following table.

Mouse Action	Result
Left-click	Selects the line at the cursor location
Ctrl+left-click	Selects all text in the text editor
Shift+left-click	Extends the line-based selection from the current anchor point to the line at the cursor location

Using the Mouse within the Fold Margin

The mouse actions that you can perform within the fold margin of the **JadeTextEdit** control text editor are listed in the following table.

Mouse Action	Result
Ctrl+Shift+left-click	Toggles (expands or contracts) the fold point and all lower-level (child) fold points to match
Shift+left-click	Toggles (expands or contracts) all outer (parent) fold points
Left-click	Toggles (expands or contracts) the nearest fold point

Using the Keyboard in the Text Editor

The keyboard and shortcut commands in the **JadeTextEdit** control follow standard conventions. All move keys (that is, arrows, Page Up, Page Down, Home, and End keys) enable the extension or reduction of the stream selection when holding down the Shift key and of the rectangular selection when holding the Shift and Alt keys.

Note The setting of the **wrapMode** property changes the behavior of the Home key. When the value of the **wrapMode** property is **SC_WRAP_WORD**, the Home key moves the caret to the start of the text line rather than to the start of the display line.

In addition to the keyboard zoom functionality commands listed in the following table, you can use the Ctrl key and the mouse to increase, decrease, or restore text size; that is, Ctrl+ scrolling the mouse wheel forward increases the zoomed size, Ctrl+ scrolling the mouse wheel backward decreases the zoomed size, and Ctrl+ clicking the middle mouse button (pressing the mouse wheel) restores the editor pane to normal.

The keyboard commands are listed in the following table.

Action	Key
Magnify text size (zoom functionality)	Ctrl+numeric + symbol
Reduce text size (zoom functionality)	Ctrl+numeric - symbol
Restore text size to normal (zoom functionality)	Ctrl+numeric / symbol
Indent block	Tab
Remove indent block	Shift+Tab
Delete to start of word	Ctrl+Backspace

Action	Key
Delete to end of word	Ctrl+Delete
Delete to start of line	Ctrl+Shift+Backspace
Delete to end of line	Ctrl+Shift+Delete
Go to start of document	Ctrl+Home
Go to the start of a newly inserted line without performing automatic indenting	Ctrl+Enter
Extend selection to start of document	Ctrl+Shift+Home
Go to start of display line	Alt+Home
Extend selection to start of text within a line	Alt+Shift+Home
Extend selection to start of line	Alt+Shift+Home pressed twice
Go to end of document	Ctrl+End
Extend selection to end of document	Ctrl+Shift+End
Go to end of display line	Alt+End
Scroll down	Ctrl+down arrow
Scroll up	Ctrl+up arrow
Line cut	Ctrl+L
Line copy	Ctrl+Shift+T
Line delete	Ctrl+Shift+L
Line transpose with previous	Ctrl+T
Line duplicate	Ctrl+D
Previous paragraph (press Shift to extend the selection)	Ctrl+[
Next paragraph (press Shift to extend the selection)	Ctrl+]
Previous word (press Shift to extend the selection)	Ctrl+left arrow
Next word (press Shift to extend the selection)	Ctrl+right arrow
Previous word part (press Shift to extend the selection)	Ctrl+/
Next word part (press Shift to extend the selection)	Ctrl+\
Toggle fold point	Ctrl+numeric keyboard ***** (asterisk)
Magnifies text size (zoom functionality)	Ctrl+numeric **+** (plus) symbol
Reduces text size (zoom functionality)	Ctrl+numeric **–** (minus) symbol
Restores text size to normal (zoom functionality)	Ctrl+numeric / (divide) symbol
Toggles all outer fold points	Ctrl+Shift+numeric keyboard ***** (asterisk)

Coloring and Text Styling

Each language supported by Scintilla has an associated lexical analyzer, which understands that language. It is responsible for examining the text contained in the control and determining, with the help of up to nine keyword tables, which characters in the text make up a keyword, numeric literal, string literal, or a comment, and setting appropriate style information about that text.

Each character of text has an associated byte of style information, which includes a style number and up to three indicator bits. Each language specifies its own table of style numbers. For example, the JADE lexer uses style **SCE_JAD_KEYWORD** (**12**) for keywords such as **if**, **while**, and **endif**, and style **SCE_JAD_COMMENTLINE** (**7**) for line-based comments beginning with a double slash (**//**).

Each character in a line of text is displayed using the attributes associated with its style number. The style attributes include font, font size, bold, italic, foreground color, and background color. The indicator bits provide an additional way to highlight text independent of the basic styling. Indicator bits can highlight a section of text with a red wavy underline symbol, to indicate a syntax error without overriding keyword coloring.

Style numbers in the range 32 (**STYLE_DEFAULT**) through 39 provide style attributes for non-text-related items such as line numbers. You can use style 32 (**STYLE_DEFAULT**) to initialize all possible styles (in the range zero through 127) to a common setting before language-specific style settings are applied.

The **clearAllStyles** method initializes the default style to the attributes specified on the control (that is, the **fontName**, **fontSize**, **fontBold**, **fontItalic**, **fontUnderline**, **foreColor**, and **backColor** property values) and then copies the default style to all other styles. The **setStyleAttributes** method alters one or more of the attributes of a specific style. The **copyDefaultToAllStyles** method copies the default style to all other styles.

The **changeKeywords** method can set one of the nine keyword lists to a specified list of words (separated by whitespace). It can also add or delete one or more words from a specified list and it can set all of the keyword lists to language-specific entries taken from the application and global setting tables.

The **applySettings** method loads the text styles table and the keywords lists with entries taken from the application and global settings tables, which are specific to the current language.

The minimum code required to have the text styled according to a fully supported language (for example, C++) is as follows.

```
control-name.language := SCLEX_CPP;
control-name.applySettings();
control-name.restyleText();
```

Folding

Folding is primarily used with block structured languages. In addition to colorization, the language lexical analyzer (lex analyzer) performs fold point calculations. It scans the text for keywords or characters that indicate statement blocks.

The line introducing a statement block is marked as a fold point. The following lines through to, and sometimes including, the line where the block ends are linked to the fold point header line. Statement blocks can have other statement blocks embedded inside them.

Programming languages like C and C++ use brace characters (**{ }**) to mark the extent of statement blocks. The JADE language uses matched pairs of keywords (for example, the **if** and **endif** instruction pair) to mark the extent of statement (instruction) blocks. The JADE language lex analyzer also recognizes **//{** and **//}** as fold block markers.

Some languages (including JADE) support the following styles of fold calculation (selected by using the named attribute **fold.compact**). See also the **setNamedAttribute** method.

- NORMAL

  The fold point is placed on the line containing the end of the header statement. For an **if** instruction, this is the line containing the **then** clause.

  The lines that can be hidden by using this fold point extend to, but do not include, the line that contains the matching block terminator word (for example, **endif)**.

- COMPACT

  The fold point is placed on the line containing the first word of the instruction (for example, **if**). The lines that can be hidden by using this fold point extend to and include the line that contains the matching block terminator word and any following blank lines.

A fold point can be in two states, as follows.

- Expanded, in which the text lines of the statement block are visible. Blocks embedded inside this block can be independently expanded or contracted.

- Contracted, in which the text lines of the statement block are all hidden, including the lines in embedded blocks.

A fold point can optionally have a horizontal line drawn above or below it, to indicate its current state. The suggested setting for a fold point (used by the **JadeEditor** class) is a line drawn below a contracted fold point (**foldFlags** := 16).

The fold point symbols drawn in the fold margin can be completely customized, by using the **setLinemarkAttributes** method to modify the attributes of linemarks in the range 25 through 31.

Use the **foldSymbols** property to select one of the four available sets of fold point markers. The symbol set represented by the **SC_FOLDSYM_TREESQUARE** constant is recommended.

To contract and expand fold points:

- Use the mouse in the fold margin

- Press Ctrl+* (the asterisk on the numeric keypad).

Linemarks

Linemarks are indicators on specific lines in a text edit control and are normally used as go-to points. You can assign style and color attributes to any of the 32 text editor linemarks, numbered zero (**0**) through **31**, which you can assign in any combination to each text editor line. (See also the **JadeTextEdit** class **setLinemarkAttributes** method.)

Linemarks are displayed in the selection margin at the left of text in a line. If the selection margin is hidden, the background color of the whole line is changed instead. Linemarks numbered **25** through **31** are used by the text folding feature and are bound to margin 2 by default. You should not use the line folding linemarks in this range for other purposes.

The remaining linemark numbers in the range zero (**0**) through **24** are bound to margin 1 and have no predefined function, so you can use them as bookmarks for indicating syntax errors or breakpoints, for example.

You can bind editor commands to key combinations, which allow a linemark to be toggled on the current line (for example, Ctrl+F7), move the caret to the next line with a specified linemark (for example, F7), or move the caret to the previous line with a specified linemark (for example, Shift+F7). Note that when the caret moves to a linemark on a line that is hidden in a contracted fold block, that fold block and any embedded fold blocks are expanded to show that line.

Each linemark number can have a different symbol, foreground color, and background color. If a line has more than one linemark set on it, they are drawn in order on top of each other, starting with zero (**0**). If low numbers are set to large symbols, they are partially visible when over-drawn by smaller symbols assigned to higher linemark numbers. If the linemark margin (**1**) is not visible and lines have linemarks set on them, each of those lines is displayed with the background color overridden with the background color of the highest linemark set on that line. A linemark with the type **SC_MARK_BACKGROUND** (**22**) is always displayed this way, even when the linemark margin is visible.

Linemarks are also known as bookmarks in other languages; for example, in MS Visual Studio.

Settings

The settings is a textual table that contains information for customizing the text editor for a specific programming language (for example, JADE, C, Perl, HTML, and so on). Each language has its own set of text styles and keyword lists, as well as some other language-specific items.

A built-in table of settings contains the information for the directly supported languages. You can retrieve this for display by calling the **JadeTextEdit** class **getGlobalSettings** method or the **Application** class **getJadeTextEditGlobalSettings** method. There are approximately 80K bytes of global settings. As the settings table is read-only, you cannot modify it.

In addition to the global table, each JADE application has a separate settings table, which is initially empty. A setting in the application table with the same key value overrides a matching setting in the global table. The application table is modified by using the **Application** class **updateJadeTextEditAppSettings** method or the **JadeTextEdit** class **updateAppSettings** method. The application settings table is shared by all instances of the **JadeTextEdit** control (or its subclasses) within the same JADE application. Use the **JadeTextEdit** class **initializeAppSettings** method to clear all entries from the application settings table.

The settings text consists of a series of *key*=*value* entries, separated by line delimiters (for example, the **CrLf** end-of-line sequence). Each entry consists of a *key/value* pair, separated by an equals (**=**) character. The value text can be split over multiple lines, using a backslash (\) character as the last character in a line to indicate continuation.

Both the key and value text can include text substitution, which is indicated by another key enclosed in parentheses preceded by a dollar character (for example, **$(*another.key*)**).

Comments can also be embedded, by using a crosshatch (**#**) character as the first non-blank character in the line.

The global setting table is generated from a selection of SciTE properties files.

You can add settings for other languages not included in the global properties table, by extracting the contents of the appropriate SciTE properties file.

Use the **Application** class **getJadeTextEditOneSetting** method to retrieve the value associated with a specific key.

Note Text substitution is performed on the value text but not on the key text.

Supported Languages

All Scintilla lexical analyzers are built-in, but keyword list and style settings are available only for a subset of the 65 languages available to Scintilla. The fully supported languages are:

- JADE
- C and C++ family
- Java
- JavaScript
- Perl
- Python
- Shell script (for example, Bash)
- Windows command files (.**bat**, .**cmd**)
- HTML, XML, SVG, PHP
- CCS
- Properties, INI, cfg, cnf
- Diff
- PostScript
- Visual Basic
- VBScript

Use the **JadeEditor** subclass of the **JadeTextEdit** control to edit JADE method source, as it has facilities that help with the building and real-time maintenance of the keyword lists.

Keyword sets and text styles for other languages are contained in the appropriate properties files included with the SciTE program that the reference application built around the Scintilla editor engine.

Unicode and ANSI Considerations

When the text edit control is used in an ANSI system, the text is limited to single 8-bit characters. Characters inserted from the keyboard with a decimal value greater than 254 are ignored.

Pasting Unicode characters from the clipboard has no defined behavior, as it is operating system-dependent. The **loadTextFromFile** method reads from ANSI text files only.

When the control is used in a Unicode system, the text is held and manipulated internally in UTF8 format. Conversion to and from wide characters is performed as text enters and leaves the control.

In a Unicode system, an exception (**15645**) is raised if an invalid wide character is found in the text. Valid characters are limited to the range **U+0000** through **U+10FFFF** (that is, the UTF-16-accessible range) as specified in *RFC 3629 - UTF-8, a transformation format of ISO 10646*.

JadeXamlControl Class

A **JadeXamlControl** control enables JADE systems to utilize the facilities offered by .NET 3 Foundation. The content of this control type is defined entirely by a Windows Presentation Foundation (WPF) Extensible Application Markup Language (XAML) definition. The control is essentially an empty canvas on which the XAML definition is displayed. It has a WPF dock panel that causes the XAML element or elements to be sized to the control size.

A **JadeXamlControl** control can have child controls (including in the JADE Painter). A child of a XAML control cannot be transparent (that is, the **transparent** property is ignored).

Transparent sibling controls are always painted before a **JadeXamlControl** control, regardless of their **zOrder** settings. It is not possible to handle the painting of transparent controls in the correct **zOrder** when some controls are directly painted by JADE and others are painted by Windows separately.

As the **JadeXamlControl** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** class and **Window** class. However, the following JADE control properties are ignored, as they are not relevant or they are ignored in favor of the defined XAML setup.

Control	Description
automaticCellControl	Use of a XAML control as a **Table** cell control is not supported
autoRedraw	Dynamic drawing on the control is not supported
backColor	Used only if the XAML content is not set
backBrush	Not supported
borderStyle	Ignored in favor of the XAML definition
draw-related properties	Dynamic drawing on the control is not supported
font-related properties	Ignored in favor of the XAML definition
ignoreSkin	Skinning a XAML control is not supported
show3D	Ignored in favor of the XAML definition

For a summary of the property, methods, and events defined in the **JadeXamlControl** class, see "JadeXamlControl Property", "JadeXamlControl Methods", and "JadeXamlControl Events", in the following subsections.

JadeXamlControl Property

The property defined in the **JadeXamlControl** class is summarized in the following table.

Property	Description
xaml	Defines the content of the control in the WPF Extensible Application Markup Language (XAML)

For details, see "Window, Form, and Control Properties", later in this document.

JadeXamlControl Methods

The methods defined in the **JadeXamlControl** class are summarized in the following table.

Method	Description
callMethod	Executes a WPF method on a specified entity of the XAML control
eventItemName	Determines the element of the XAML contents that issued the JADE event
getValue	Returns the value of a WPF method or property for the specified XAML entity
setValue	Sets the value of a WPF property for the specified XAML entity
setXamlEventMethod	Registers for other WPF control events apart from the standard JADE events

For details, see "Window, Form, and Control Methods", later in this document.

JadeXamlControl Events

The event methods defined in the **JadeXamlControl** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
gotFocus	When a control receives the focus
keyDown	When the user presses a key while the control has the focus
keyPress	When the user presses and releases an ANSI key
keyUp	When the user releases a key while the control has the focus
lostFocus	When a control loses the focus
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control (only called at the **JadeXamlControl** level)
mouseLeave	When the user moves the mouse off a control (only called at the **JadeXamlControl** level)
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call
windowCreated	Called for all controls when the window for the control is created, so that a control can be initialized when the window for the control is present

For details, see "Window, Form, and Control Events", later in this document.

Note The **paint** method is never called.

Label Class

A label is a control that you can use to display text that cannot be directly changed by the user.

As the **Label** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** class and **Window** class.

You can write logic that changes a label in response to events at run time. For example, if your application takes a while to commit a change, you could display a processing-status message in a label. You can also use a label to identify a control (for example, a **TextBox** control) that does not have its own **caption** property.

A label can be set to automatically resize to the caption size.

The **Label** class provides the following subclasses.

- ProgressBar
- WebHotSpot
- WebInsert
- WebJavaApplet

For a summary of the constants, properties, method, and events defined in the **Label** class, see "Label Class Constants", "Label Properties", "Label Method", and "Label Events", in the following subsections.

Label Class Constants

The constants provided by the **Label** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Alignment_Center_Bottom	8	Alignment_Center_Middle	7
Alignment_Center_Top	6	Alignment_Left_Bottom	2
Alignment_Left_Middle	1	Alignment_Left_Top	0
Alignment_Right_Bottom	5	Alignment_Right_Middle	4
Alignment_Right_Top	3		

Label Properties

The properties defined in the **Label** class are summarized in the following table.

Property	Description
alignment	Contains the placement of the text in the control
autoSize	Specifies whether a control is automatically resized to fit its contents
caption	Contains the control caption
formatOut	Contains system-defined formats of data when printing labels

Property	Description
hyperlink	Contains a hyperlink string that is programmatically attached to the label control
noPrefix	Specifies whether the character following a single ampersand (**&**) is underlined in the caption of a label
transparent	Causes the control to be placed above all other sibling controls and the controls or form underneath to be visible
wordWrap	Specifies whether text displayed in a caption advances to the next line of the control when the current line is filled

For details, see "Window, Form, and Control Properties", later in this document.

Label Method

The method defined in the **Label** class is summarized in the following table.

Method	Description
isSizeable	Returns **false** to specify that the control cannot be resized in the JADE Painter

For details, see "Window, Form, and Control Methods", later in this document.

Label Events

The event methods defined in the **Label** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

ListBox Class

A list box control displays a list of items from which the user can select one or more items. If the number of items exceeds the number that can be displayed, a vertical scroll bar is automatically added to the list box.

To add an item to a list

- Use the **addItem** or **addItemAt** method.

To delete an item from a list

- Use the **removeItem** method.

If no item is selected, the **listIndex** property value is **-1**. The first item in the list has a **listIndex** of **1**, and the **listCount** method returns the number of items in the list. The **text** property returns the text for the currently selected item.

When multiple items are currently selected, the value of the **listIndex** property is the last of the items selected. One or more items can be selected, with the value of the **listIndex** property being none of those items (for example, when you select an item, press the Shift key and select another item, then press the Ctrl key and remove the selection of one of the previously selected items).

Tip It is much more efficient to copy a GUI value into a local variable for reuse rather than request the value again. For example, **listBox.listCount** requires the calling of a list box method to retrieve the value. Storing the value in a local property for reuse avoids a significant overhead for the second and subsequent requests for that value when it will not change. The first of the following examples is much more expensive than the second of these examples.

```
while count <= listBox.listCount do  // inefficient use of the variable

vars                                 // recommended use of the variable
    listBoxCount : Integer;
begin
    listBoxCount := listBox.listCount;
    while count <= listBoxCount do
        ...
    endwhile;
end;
```

If the width of any displayed item exceeds the width of the list box and the **scrollHorizontal** property is set to **1** (automatic), a horizontal scroll bar is added to the list box. The scroll ranges for the list box are set automatically and cannot be changed. For details about the support of mouse wheel requests to scroll up, down, or across a list box control, see "Window Class", earlier in this document. A list box can have a maximum of 32,000 items.

You can implement filtering of combo box and list box entries to enable users to locate a required list item more quickly, using standard combo box and list box facilities to achieve this filtering. (For details, see "**Filtering Entries in Combo Box and List Box Text**", in Chapter 2 of the *JADE Development Environment User's Guide*.) You can achieve the filtering of entries in your own JADE systems, as follows.

- Set the value of the **style** property for the **ComboBox** control class to **ComboBox.Style_DropDown** (0).
- Set the value of the **ListBox** control **hasPictures**, **hasTreeLines**, and **hasPlusMinus** properties to **false**.
- The value of the **sorted** property must be set to **false** after the list is loaded.
- A disabled **description** list item is added as the first entry in the list (after sorting is turned off).

- The **change** event on the combo box or associated text box calls a filtering routine to hide or show the entries based on the new text entered. Setting the level of the item to **2** hides the entries, which is why there has to be an entry at item position **1** that is always at level **1**. (A child item must have a parent.)

Press Ctrl+Home to move a list box with an associated collection to the first entry in the collection. Conversely, press Ctrl+End to move the display to the last entry in the collection.

Note Pressing the Home key on a list box moves the display to the first entry that has been loaded in the control. Pressing the End key moves the display to the last entry currently loaded in the control. Using the Ctrl key for a list box that does not have a collection attached has the same result as pressing the Home or End key without the Ctrl key. For details about dragging the scroll bar thumb of a list box when an **Array**, **Dictionary**, or **Set** collection is attached to the list box, see the **displayCollection** method.

If the list item entry in a list box is too long to fit in the list, bubble help showing the complete text is automatically displayed over the text portion of the entry when the mouse is moved over that entry. Bubble help is no longer displayed if the mouse is moved off that entry or after approximately three seconds.

Notes Clicking on the bubble help generates a **click** event for that list entry.

The automatic display of bubble help does not occur if the list box already has bubble help defined (by using the **Window**::**bubbleHelp** property).

To disable the automatic bubble help for a list box, set the **bubbleHelp** property of that control to a space.

Note For the arrays associated with list boxes (for example, **itemBackColor**), the only methods that are implemented are **at**, **atPut** (which enables you to use the square brackets notation to access the elements), **createIterator** (which allows logic to do a **foreach** over the array), **size**, and **size64**.

For a summary of the constants, properties, methods, and events defined in the **ListBox** class, see "ListBox Class Constants", "ListBox Properties", "ListBox Methods", and "ListBox Events", later in this section. See also "Setting Properties for Individual Items in a List Box", "Using a List Box to Display a Hierarchy or Tree", and "Copying Text from a List Box", in the following subsections.

Setting Properties for Individual Items in a List Box

The properties summarized in the following table can be set for an individual item in a **ListBox** control.

Property	Description
itemBackColor	Contains the background color of each item
itemData	Contains a specific number for each item
itemEnabled	Specifies whether individual items are disabled or enabled
itemExpanded	Contains the expansion (or collapse) status of each item
itemForeColor	Contains the text color of each item
itemLevel	Contains the hierarchical level of each item
itemObject	Contains an object for each entry
itemPicture	Contains a picture for individual items
itemPictureType	Contains the type of picture of each item
itemSelected	Contains the selection status of each item
itemText	Contains the text of an item

Each of these properties is an array of values, with the same number of entries as items in the list box. To set a property for an item in a list box, you must also specify the index of the item; for example:

```
    myListBox.itemText[1]
```

The previous example returns the text for the first item in a list box called **myListBox**.

The code fragment in the following example sets the **itemText** property for the tenth item in this list box.

```
    myListBox.itemText[10] := "Fred";
```

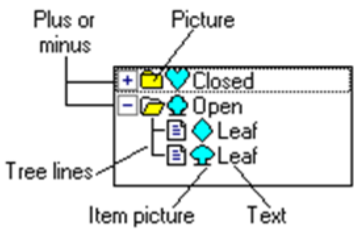
See also "Using a List Box to Display a Hierarchy or Tree", in the following subsection.

Using a List Box to Display a Hierarchy or Tree

The **ListBox** class also provides features to display items in a hierarchy. Each item in the list can have subordinate items visually represented by indentation levels. When an item is expanded, its subordinate items are visible. When an item is collapsed, its subordinate items are hidden.

Items in the **ListBox** control can also display graphical elements to provide visual cues about the state of the item. Picture images in a list box are transparent only if the image type in JADE handles transparency; for example, icons or Graphics Interchange Format (GIF) files. If the image is a bitmap, the list box treats any white pixels on the outside of the image as transparent.

The following image shows an example of the types of the extended feature display.



The display is made up of the parts listed in the following table.

Display Part	Description
Tree lines	The tree lines are displayed if the **hasTreeLines** property is set to **true**. Clicking on this image causes a **pictureClick** event, in which logic can expand or collapse the item.
Plus or minus	An icon or bitmap is displayed if the **hasPlusMinus** property is set to **true** and the list item has subitems. A different image is normally displayed, depending on whether the item is expanded or collapsed. The default images are shown in the above image. These images can be changed by using the **picturePlus** and the **pictureMinus** properties. Clicking this portion of an item has the same impact as tree lines.
Picture	The picture image is present if the **hasPictures** property is set to **true**. The type of image that is displayed depends on whether the item has subitems, or is closed or expanded. The default images are shown in the above image. These images can be changed by using the **pictureClosed**, **pictureOpen**, and **pictureLeaf** properties. Clicking this portion of an item has the same impact as tree lines.
Item picture	If the list item has a picture (set by the **itemPicture** property), that picture is placed after all other images, just before the text. The picture is scaled to fit the list item height.

Display Part	Description
Text	The text portion of the list item is always present. An item is selected only (setting the **listIndex** property) when the text is clicked, or when arrow keys, the Page Up, Page Down, Home, or End key selects a new item. Clicking the text area causes a **click** event and selects that item as the current item.

If the **sorted** property is set to **true**, the hierarchical properties (**hasPictures**, **hasPlusMinus**, and **hasTreeLines**) are set to **false**. See also "Setting Properties for Individual Items in a List Box", in the previous subsection.

Copying Text from a List Box

When a **ListBox** control has focus, you can copy the contents of the list box to the clipboard, by using the Ctrl+C or Ctrl+Insert shortcut key combination. The copy action starts with the first entry currently displayed in the list box and ends with the last entry in the list box.

A carriage return / line feed character (**CrLf**) is added to the end of each entry's text.

Note In the JADE development environment, the Edit menu includes a Ctrl+C menu accelerator for the **Copy** command. The list boxes displaying classes, properties, and methods will therefore not receive a Ctrl+C copy request because such a request is sent to the editor pane. To copy the contents of the classes, properties, or methods list box, use the Ctrl+Insert key combination.

Entering Characters to Find an Entry in a List Box

When a list box has focus, entered characters are accumulated as a string and are used to select the current entry.

If the user delays for more than a half a second between keystrokes, the current accumulation is discarded and restarted. For example, to find an element beginning **SP**, enter **P** within half a second of entering **S**. If you delay longer than this, you are effectively starting a new selection for an element beginning with **P**.

ListBox Class Constants

The constants provided by the **ListBox** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
DisplayCollection_Forward	0	DisplayCollection_NoPrior	0
DisplayCollection_Prior	2	DisplayCollection_Reversed	1
ItemNotFound	-1	ItemPictureType_Closed	0
ItemPictureType_Leaf	2	ItemPictureType_Open	1
MultiSelect_Extended	2	MultiSelect_None	0
MultiSelect_Simple	1	PictureClick_ItemPicture	4
PictureClick_KeyBoard	5	PictureClick_Picture	3
PictureClick_PlusMinus	1	PictureClick_TreeLine	2
ScrollHorizontal_Auto	1	ScrollHorizontal_None	0

ListBox Properties

The properties defined in the **ListBox** class are summarized in the following table.

Property	Description
alternatingRowBackColor	Specifies an alternate row background color
alternatingRowBackColorCount	Specifies the number of list entry or table rows at which the alternating background color of each visible list entry, non-fixed row, and non-fixed cell is displayed
defaultLineHeight	Specifies the default height of lines in a list box independent of the font size
hasPictures	Specifies whether the picture images are displayed
hasPlusMinus	Specifies whether plus/minus images are displayed
hasTreeLines	Specifies whether the tree lines are drawn
integralHeight	Specifies whether the list box height shows partial lines
itemBackColor	Contains the background color of each item
itemData	Contains a specific number for each item
itemEnabled	Specifies whether individual items are disabled or enabled
itemExpanded	Contains the expansion (or collapse) status of each item
itemForeColor	Contains the text color of each item
itemLevel	Contains the hierarchical level of each item
itemObject	Contains an object for each entry
itemPicture	Contains individual items to assign to a picture
itemPictureType	Contains the type of picture of each item
itemSelected	Contains the selection status of each item
itemText	Contains the text of an item
listIndex	Contains the index of the currently selected item
listObject	Contains the associated object of the currently selected item
multiSelect	Specifies whether a user can make multiple selections
nameSeparator	Contains the item delimiter string used when accessing the **itemFullName** method
pictureClosed	Contains the qualifying picture image displayed for an entry
pictureLeaf	Contains the qualifying picture image displayed for an entry
pictureMinus	Contains the qualifying picture image displayed for an entry
pictureOpen	Contains the qualifying picture image displayed for an entry
picturePlus	Contains the qualifying picture image displayed for an entry
scrollBars	Contains the scroll bars that can be displayed when text extends beyond the client window co-ordinates of the control

Property	Description
scrollHorizontal	Specifies whether a horizontal scroll bar is added when a line item does not fit horizontally
scrollHorzPos	Contains the position of the horizontal scroll bar
sortAsc	Specifies whether the sorting is ascending or descending
sortCased	Specifies whether the sorting is case-sensitive
sorted	Specifies whether the elements are automatically sorted alphabetically
text	Contains the text of an item
topIndex	Contains the item that is the first item displayed in the list

For details, see "Window, Form, and Control Properties", later in this document.

ListBox Methods

The methods defined in the **ListBox** class are summarized in the following table.

Method	Description
addItem	Adds a new item
addItemAt	Adds a new item at a specified item index
clear	Clears the contents
clearAllSelected	Clears all selected items in the list box
displayCollection	Attaches the specified collection to the list box
dragListIndex	Provides the **listIndex** property of an entry during drag and drop actions
findObject	Searches the **itemObject** property values of the list entries for the specified object
findString	Searches the entries for an entry that matches the specified string
findStringCaseSensitive	Searches the entries for an entry that matches the specified case-sensitive string
findStringExact	Searches the entries for an entry with an exact match to the specified string
findStringExactCaseSensitive	Searches the entries for an entry with an exact match to the specified case-sensitive string
getCollection	Returns the collection attached to the list box by the **displayCollection** or **listCollection** method
getLineHeight	Returns the height in pixels of each list box entry
getListIndex	Returns the index of the displayed list entry corresponding to the position x, y
getListIndexText	Returns the list index that corresponds to the specified position within the text portion of a list entry
getScrollRange	Returns the scroll range information for the window
itemFullName	Returns the fully qualified name of an item

Method	Description
itemHasSubItems	Returns a **Boolean** value that indicates whether an item has subitems
itemVisible	Returns a **Boolean** value that indicates whether an item is visible
lines	Returns the number of lines available for display in the list box
listCollection	Enables controls to have a collection attached to them
listCount	Returns the number of items in the list
newIndex	Returns the index of the item most recently added
positionCollection	Positions the collection attached to the **ListBox** control to an object in that collection and to a position within the list box
positionLeft	Returns the displayed left position in pixels of the start of text in the specified list entry
positionTop	Returns the displayed top position in pixels of the specified list entry
refreshEntries	Refreshes the displayed list of entries in the list box
removeItem	Removes an item (and its subitems) from a list box
selectedCount	Returns the number of entries selected in the list box

For details, see "Window, Form, and Control Methods", later in this document.

ListBox Events

The event methods defined in the **ListBox** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
displayEntry	When a control is attached to a collection object and the text to be displayed for an entry in the collection is required
displayRow	For each entry in the collection of the current list, to display the contents of the row
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
gotFocus	When a control receives the focus
keyDown	When the user presses a key while the control has the focus
keyPress	When the user presses and releases an ANSI key
keyUp	When the user releases a key while the control has the focus
lostFocus	When a control loses the focus
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control

Event	Occurs…
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
pictureClick	When the picture area before the text of an item is clicked with the mouse
pictureDblClick	When the picture area before the text of an item is double-clicked with the mouse
scrolled	When the user scrolls
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

MultiMedia Class

The **MultiMedia** control provides the ability to play sound and video files. It also allows the control of devices such as audio compact discs (CDs), image scanners, and MIDI sequencers. The **MultiMedia** control is a wrapper for the standard Microsoft Video for Windows **MCIWnd** window class.

The **MultiMedia** control provides options that enable the file to be played in its window with or without a playbar, title bar, and controlling popup menus. The playbar enables a user to start or stop the playback. (For details about the file types that can be handled by this control, see "MultiMedia File Types", later in this section.) The control also provides the ability to:

- Play portions of the medium
- Position and step the playback
- Pause and continue the playback
- Record data when the device is capable

To control the device in a more complex fashion, the **sendString** method enables commands to be issued. For examples of the commands that are available in a Windows GUI environment, see the "Multimedia Command Strings" section under "Reference", in the *Microsoft Developer Network* product documentation.

Caution Not all commands issued by the **sendString** method are valid and available for all device types; for example, changing the volume is not valid for a device that has no sound. Check the device documentation for compliance. (For details about the device types that can be accessed from the **MultiMedia** class, see "MultiMedia Device Types", later in this section.)

If the medium name is set at development time and involves a file, that file is copied into the database. As JADE handles only binary data that has a length less than the maximum database cache size, an attempt to store large files in the database may fail. To cover situations where the data is copied by logic, ensure that only files of a size less than half the cache size are stored.

Transparent sibling controls are always painted before a **MultiMedia** control, regardless of their **zOrder** settings. It is not possible to handle the painting of transparent controls in the correct **zOrder** when some controls are directly painted by JADE and others are painted by Windows separately.

The **MultiMedia** control provides notify events that inform JADE logic of changes of media, operating mode, and playback position. For details about using a **MultiMedia** control, see "Using MultiMedia Controls", in the following subsection.

For a summary of the constants, properties, methods, and events defined in the **MultiMedia** class, see "MultiMedia Class Constants", "MultiMedia Properties", "MultiMedia Methods", and "MultiMedia Events", later in this section.

Using MultiMedia Controls

The steps that you perform to use the **MultiMedia** control are:

1. Add a **MultiMedia** control to your form, and then set the required properties.
2. Set the **mediaName** property, by performing one of the following actions.
   - Specify the file or device in the **Specific** sheet of the Properties dialog in Painter.
   - Call the **openDialog** method to enable the user to select the file or device to be loaded.
   - Explicitly assign a value to the property in your logic.
3. Reimplement event methods.
4. Use the **MultiMedia** methods to control the behavior of the file or device, or use the **sendString** method to control the file or device in a more complex fashion.

Notes When the playbar is visible on the control, no JADE events are issued for mouse actions over that playbar area. In addition, if the playbar is visible and the **useDotNetVersion** property is set to **true**, it is drawn using WPF entities and it has a different appearance from the playbar drawn using an MP3 file type.

Setting the **zoom** property to zero (**0**) when the value of the **autoSize** property is **false** stretches the media image to the current size of the client area of the control.

The **MultiMedia** control is not supported on forms defined as Web pages and is ignored when HTML is generated.

To play a video file, you need only perform the following actions.

1. Set the **mediaName** property to the file name that is to be played
2. Call the **play** method, as shown in the code fragment in the following example.

```
mmcontrol.mediaName := "c:\image.avi";
mmcontrol.play;
```

If the **useDotNetVersion** property is set to **true**, media attribute values position (obtained by calling the **getEndPosition** or **getStartPosition** method) and length (obtained by calling the **getLength** method) are not available until the medium has been opened and the **notifyMedia** or **notifyMode** event has been received. For details about using the MP4 version of a control, see the **useDotNetVersion** property.

MultiMedia Device Types

The device types that can be accessed by the **MultiMedia** class are listed in the following table.

Device Type	Description
animation	Animation device
cdaudio	Audio CD player
dat	Digital audio tape player
digitalvideo	Digital video in a window
overlay	Overlay device (analog video in a window)
scanner	Image scanner
sequencer	MIDI sequencer
vcr	Videotape recorder or player
videodisc	Videodisc player
waveaudio	Audio device that plays digitized waveform files

MultiMedia File Types

The file types that can be handled by the **MultiMedia** class depend on the software that is installed on the workstation, and can include the types of files listed in the following table.

File Type	Description
wav	Sound files
mid	MIDI sequence
avi	Video with or without sound
mp3	MPEG-1 Audio Layer-3
mp4	MPEG Layer-4 Audio
mpg	Mpeg

For details about using the MP4 version of a control, see the **useDotNetVersion** property.

MultiMedia Class Constants

The constants provided by the **MultiMedia** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Mode_Not_Ready	1	Mode_Open	7
Mode_Paused	6	Mode_Playing	3
Mode_Recording	4	Mode_Seeking	5
Mode_Stopped	2		

MultiMedia Properties

The properties defined in the **MultiMedia** class are summarized in the following table.

Property	Description
autoSize	Specifies whether the control is automatically resized to fit its contents
mediaData	Contains the data associated with the current device
mediaName	Contains the name of the currently installed device
position	Contains the current position of the medium with the content of the device
repeat	Specifies whether continuous playback mode is set
showMenu	Specifies whether a menu button is displayed on the playbar
showMode	Specifies whether the control includes a caption including the device status
showName	Specifies whether the control includes a caption including the medium name
showOpenMenu	Specifies whether a menu item enabling the opening of a file is displayed
showPlayBar	Specifies whether a playbar is included
showPosition	Specifies whether the control includes a caption including the current position
showRecord	Specifies whether a record button is displayed on the playbar
speed	Contains the playback speed of the device
timeFormat	Contains the time format of the device
timerPeriod	Contains the timer period used for the notify events
useDotNetVersion	Specifies whether the control uses .NET, providing access to new style media files such as an MPEG Layer-4 Audio (MP4) file
volume	Contains the playback volume of the device
zoom	Contains the zoom factor of the video image

For details, see "Window, Form, and Control Properties", later in this document.

MultiMedia Methods

The methods defined in the **MultiMedia** class are summarized in the following table.

Method	Description
canEject	Returns whether the device can eject its media
canPlay	Returns whether the device can play its media
canRecord	Returns whether the device supports recording
canSave	Returns whether the device supports saving data
close	Closes the associated device or file
eject	Requests the device to eject its media
getEndPosition	Returns the end position of the content of the medium in the device

Method	Description
getLength	Returns the length of the content of the medium in the device
getMode	Returns the current operating mode of the device
getStartPosition	Returns the start position of the content of the medium in the device
hasAudio	Returns whether the current device type supports audio.
hasVideo	Returns whether the current device type supports video
isSizeable	Returns **false** to specify that the control cannot be resized in the JADE Painter
newFile	Creates a new file for the current device
openDialog	Displays a version of the common Open File dialog
pause	Pauses the playing or recording of the device or file
play	Starts the device playing from the current position in the content
playFromTo	Plays the content of the current device or file from and to specified positions
playReverse	Starts the device playing for the current position in the content in the reverse direction
record	Begins recording content at the current position of the content of the device
resume	Resumes playback or recording content from the paused mode
save	Saves the content currently used by a device
sendString	Enables commands to be issued directly to the device driver
stepRelative	Moves the current position in the content forward or backwards
stop	Stops the playing or recording if the device
usesFiles	Returns whether the data storage used by the device is a file

For details, see "Window, Form, and Control Methods", later in this document.

MultiMedia Events

The event methods defined in the **MultiMedia** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a control belonging to the same application
dragOver	For each form or control of the application over which a window is dragged
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button

Event	Occurs…
notifyMedia	When there is a change in the media
notifyMode	When there is a change in the status of the **MultiMedia** control
notifyPosition	When there is a change in the position of the content
paint	When part or all of a control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Ocx Class

The **Ocx** system class provided by JADE handles any external OLE objects that are registered on your workstation and added to JADE by the interface to external ActiveX controls (formerly known as external Object Linking and Editing, or OCX, controls).

Caution To register and run an ActiveX control, all libraries used by the control must be available.

An ActiveX control is contained in a library written in a language like C++ by an external developer. This library defines the behavior of a **Window** class OCX control.

When an ActiveX control is imported using the earlier import option, a JADE subclass of the **Ocx** class is created that describes the properties, methods, and events defined and exposed by the ActiveX control. This information is obtained from the control itself. At run time, JADE translates the property and method requirements into ActiveX control equivalents and then calls the control to perform the function.

Notes As the ActiveX control is external to JADE, no documentation about the control is contained within JADE, and any problems in the functionality of the control should be taken up with the suppliers of that control. If an error occurs, JADE is dependent on the control returning error information that can be displayed to the user.

This class has now been superseded by the **ActiveXControl** class, but it is still valid.

Transparent sibling controls are always painted before an **Ocx** control, regardless of their **zOrder** settings. It is not possible to handle the painting of transparent controls in the correct **zOrder** when some controls are directly painted by JADE and others are painted by Windows separately.

All ActiveX controls are defined as subclasses of the JADE abstract **Ocx** class. OCX subclasses inherit all of the standard properties and methods of the **Window**, **Control**, and **Ocx** classes, although not all of these properties and methods have meaning to the control.

To refresh an ActiveX control that has changed but is already imported into JADE, simply import the ActiveX again and give it the same name that it had previously.

The following OCX standard properties are handled automatically by JADE, using the **Window** class or **Control** class properties summarized in the following table. The control may not have implemented these standard ActiveX properties, and changing them by using JADE logic may then have no effect.

OCX Property	JADE Property
DISPID_BACKCOLOR	backColor
DISPID_BORDERSTYLE	borderStyle

OCX Property	JADE Property
DISPID_FORECOLOR	foreColor
DISPID_ENABLED	enabled
DISPID_FONT	fontBold, fontName, fontSize, fontUnderline, fontStrikethru, fontItalic
DISPID_TABSTOP	tabStop

The following ActiveX properties are assumed to be the equivalent JADE **Window** properties when the property types match.

- enabled
- mousePointer
- borderStyle
- fontBold
- fontSize
- fontName
- fontItalic
- fontStrikethru
- fontUnderline

For a summary of the constants, property, method, and events defined in the **Ocx** class, see "Ocx Class Constants", "Ocx Property", "Ocx Method", and "Ocx Events", in the following subsections.

Ocx Class Constants

The constants provided by the **Ocx** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
AutoSize_Control	0	AutoSize_Object	1

Ocx Property

The property defined in the **Ocx** class is summarized in the following table.

Property	Description
autoSize	Specifies whether a control is automatically resized to fit its contents

For details, see "Window, Form, and Control Properties", later in this document.

Ocx Methods

The methods defined in the **Ocx** class are summarized in the following table.

Method	Description
ocxClassName	Returns the Windows Registry name of the control
processInputFromWeb	When reimplemented, processes ActiveX controls used on Web pages

For details, see "Window, Form, and Control Methods", later in this document.

Ocx Events

The event methods defined in the **Ocx** class are summarized in the following table.

Event	Occurs…
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
gotFocus	When a control receives the focus
lostFocus	When a control loses the focus
paint	When part or all of a control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

OleControl Class

The **OleControl** class allows attachment of objects that are actually edited or controlled by other applications. For example, a Word document, an Excel spread sheet, a video clip, or a Paintbrush picture can be stored in JADE, but editing or playing the object invokes the controlling application.

The **OleControl** class handles any objects for which an Object Linking and Editing (OLE) Server is registered in the database of your operating system registry. An application that is capable of being an OLE Server has a **.reg** file associated with it. To load the registration information, double-click on the **.reg** file in File Manager or Explorer.

The following objects can be handled by the **OleControl** class.

- Word for Windows
- Excel
- Media Player (for sound and videos)
- Sound Recorder
- Paintbrush
- Word Art 2

- Equation 2
- MS Graph

Notes The **OleControl** class is not supported on forms defined as Web pages and is ignored when HTML is generated. For details about interfacing to OLE, see "Interfacing to OLE 2.0", in Chapter 2 of the *JADE External Interface Developer's Reference.*

The JADE extract process extracts a definition of the **OleControl** object (as it does with other controls painted at design time when the form definitions are extracted) but it does not extract the user data associated with the control. It is your responsibility to extract the user data and load it into a deployed database, if required.

For a caveat on the use of SVG files when printing **OleControl** controls, see "Portable Printing" under "Printer Class", in Volume 1 of Chapter 1 of the *JADE Encyclopaedia of Classes*.

Transparent sibling controls are always painted before an **OleControl** control, regardless of their **zOrder** settings. It is not possible to handle the painting of transparent controls in the correct **zOrder** when some controls are directly painted by JADE and others are painted by Windows separately.

For a summary of the constants, properties, methods, and events defined in the **OleControl** class, see "OleControl Class Constants", "OleControl Properties", "OleControl Methods", and "OleControl Events", in the following subsections.

OleControl Class Constants

The constants provided by the **OleControl** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Activation_DblClick	2	Activation_Manual	0
Activation_SetFocus	1	GetUsername_App	3
GetUsername_Full	1	GetUsername_Short	2
ObjectType_Embedded	1	ObjectType_Linked	2
ObjectType_None	0	SizeMode_AutoSizeControl	2
SizeMode_ClipToControl	0	SizeMode_Proportional	3
SizeMode_StretchToControl	1		

OleControl Properties

The properties defined in the **OleControl** class are summarized in the following table.

Property	Description
activation	Specifies whether the application of the object can be activated by its primary verb
allowInPlace	Specifies whether in-place editing can occur
displayAsIcon	Specifies whether only the icon of the application is displayed
fullName	Contains the full name of the OLE object in the control
oleObject	Contains an OLE object

Property	Description
shortName	Contains the short name of the OLE object in the control
showMenu	Specifies whether the control displays a popup menu when the right mouse button is clicked over the inactive control
sizeMode	Contains the automatic sizing of the control or image

For details, see "Window, Form, and Control Properties", later in this document.

OleControl Methods

The methods defined in the **OleControl** class are summarized in the following table.

Method	Description
applyVerb	Calls the application with the specified verb
cloneSelf	Creates a new instance of the same type as the receiver
close	Closes the presentation of the object
discard	Closes and discards the OLE object presentation
embedFromClass	Initiates the specified server application to create or attach a new OLE object
embedFromFile	Embeds an OLE object from the specified file into a control
getUserName	Returns the OLE internal names for the object
isObjectOpen	Determines whether the OLE server application has the object open for editing
linkFromFile	Links to an OLE object based on the specified file
loadFromDB	Loads the control from its **oleObject** property
objectType	Returns the loaded object type
showInsertForm	Enables the user to select the OLE object to be loaded into the control
update	Updates the OLE object

For details, see "Window, Form, and Control Methods", later in this document.

OleControl Events

Although the standard **Window** event methods summarized in the following table are defined in the **OleControl** class, when the OLE object is active, the server application rather than the OLE control receives the Windows messages, and the JADE events do not occur.

Event	Occurs…
click	When the user presses and then releases the left mouse button
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged

Event	Occurs…
gotFocus	When a control receives the focus
lostFocus	When a control loses the focus
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
sysNotify	When a specified JADE system event occurs
updated	When an OLE object is saved in the JADE database
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

OptionButton Class

An option button (radio button) control displays an option that can be turned on or off. Only one of a group of option buttons can be turned on. As the **OptionButton** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** and **Window** classes.

Usually, option buttons are used as part of a group to display multiple options from which the user can select only one. You can group option buttons by drawing them inside another control that accepts children (for example, a **Frame** control or **GroupBox** control), or you can add them directly to a form.

To group option buttons in a parent control, draw the parent first and then draw the option buttons inside it. All option buttons within the same parent are treated as a group. Option buttons on a form are also treated as a separate group from any option buttons in a control.

While option buttons and check boxes may appear to function similarly, there is an important difference: when a user selects an option button, the other option buttons in the same group are automatically cancelled. In contrast, any number of check boxes can be selected.

The height of the option button is automatically set to fit the caption.

When focus is on an option button, the right and down arrow keys move the focus to the next option button in the tab order in the group and set the option button on, generating a **click** event. Similarly, the left and up arrow keys move the focus to the prior option button in the tab order in the group and set the value on, generating a **click** event.

Option buttons (radio buttons) are drawn using the current Windows theme that is in use, unless the currently active JADE skin defines the option button look and feel.

For a summary of the constants, properties, and events defined in the **OptionButton** class, see "OptionButton Class Constants", "OptionButton Properties", and "OptionButton Events", in the following subsections.

OptionButton Class Constants

The constants provided by the **OptionButton** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Alignment_Left	0	Alignment_Right	1

OptionButton Properties

The properties defined in the **OptionButton** class are summarized in the following table.

Property	Description
alignment	Specifies whether the text is placed before or after the button bitmap image
autoSize	Specifies whether an option button control is automatically resized to fit its contents
caption	Contains the caption of an option button control
transparent	Causes the option button control to be placed above all other sibling controls and the controls underneath to be visible
value	Specifies whether the state of the button is up or down

For details, see "Window, Form, and Control Properties", later in this document.

OptionButton Events

The events defined in the **OptionButton** class are summarized in the following table.

Event	Description
change	Indicates that the contents of the control have changed
click	Occurs when the user presses and then releases the left mouse button
contextMenu	Occurs after the right **mouseUp** event and after the **keyUp** event
dragDrop	Occurs when a dragged window is dropped over a window belonging to the same application
dragOver	Occurs for each window of the application over which a window is dragged
gotFocus	Occurs when a control receives the focus
keyDown	Occurs when the user presses a key while the control has the focus
keyPress	Occurs when the user presses and releases an ANSI key
keyUp	Occurs when the user releases a key while the control has the focus
lostFocus	Occurs when a control loses the focus
mouseDown	Occurs when the user presses a mouse button
mouseEnter	Occurs when the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer

Event	Description
mouseLeave	Occurs when the user moves the mouse off a control
mouseMove	Occurs when the user moves the mouse
mouseUp	Occurs when the user releases a mouse button
paint	Occurs when part or all of a control is exposed
sysNotify	Occurs when a specified JADE system event occurs
userNotify	Occurs when triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Picture Class

A picture box control can display a graphic from a bitmap, icon, cursor, or metafile. It displays only as much of the graphic image as fits into the rectangle that you have drawn with the Painter **Picture box** tool, unless the **stretch** property is set.

In JADE thin client mode, all picture properties set by logic are cached on both the application server and in the form file on the presentation client. When a picture has been transmitted to the client, subsequent use of that picture is achieved by use of an identifier to that picture, eliminating the need to transmit the picture data.

When an image containing transparency is displayed in a browser as part of a Web-enabled application, the following conditions must be true for the transparency to be displayed:

- The drawing methods of the **Window** class, such as **drawLine**, are not used on the control
- The value of the **caption** property of the control is null
- The client area of the control is the same size as the image, which is true when the **stretch** property of the control is set to **Stretch_ToControl** (**1**)

As the **Picture** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** and **Window** classes. The **Picture** control class provides the **JadeMask** subclass.

For a summary of the constants, properties, methods, and events defined in the **Picture** class, see "Picture Class Constants", "Picture Properties", "Picture Methods", and "Picture Events", in the following subsections.

For details about the support of mouse wheel requests to scroll up, down, or across a picture control, see "Window Class", earlier in this document.

Picture Class Constants

The constants provided by the **Picture** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Stretch_CenterPicture	5	Stretch_ControlTo	2
Stretch_None	0	Stretch_Pic_Proportional	4
Stretch_Proportional	3	Stretch_ToControl	1

Picture Properties

The properties defined in the **Picture** class are summarized in the following table.

Property	Description
cachePictures	Specifies whether pictures are stored in a cache file when running in JADE thin client mode
clipControls	Specifies whether the Windows environment creates a clipping region that excludes controls contained by the object
hyperlink	Contains a hyperlink string that is programmatically attached to the picture control
picture	Contains the graphic to be displayed
pictureCount	Contains the maximum index of the picture array of the control
pictureDisabled	Contains a graphic to be displayed when the control is disabled
pictureDown	Contains a graphic to be displayed when the left mouse button is down over the control
pictureIndex	Contains the picture that is displayed in a picture box
rotation	Provides ability to rotate the current picture in the control
scrollBars	Specifies whether an object has horizontal or vertical scroll bars
scrollHorzPos	Contains the position of the horizontal scroll bar
scrollVertPos	Contains the position of the vertical scroll bar
stretch	Determines how a picture stretches to fit the size of the control
transparent	Causes the control to be placed above all other sibling controls and the controls underneath to be visible
transparentColor	Contains the color to be made transparent in a bitmap
webFileName	Contains the name of the image that is to be displayed on the Web page

For details, see "Window, Form, and Control Properties", later in this document.

Picture Methods

The methods defined in the **Picture** class are summarized in the following table.

Method	Description
getScrollRange	Returns the scroll range information for the window
pictureHeight	Returns the height (in pixels) of the current image in the **Picture** control
pictureType	Returns the type of picture loaded into a picture control
pictureWidth	Returns the width (in pixels) of the current image in the **Picture** control
play	Starts cycling through the pictures in an array with a pause between the display of each picture in the array
setPicture	Sets the specified index entry of an array of pictures for the control
setScrollRange	Enables control of the scroll ranges

Method	Description
startDrawingCapture	Enables drawing on a **Picture** control with the mouse
stop	Stops the current animation process started by the **play** method
stopDrawingCapture	Disables drawing on a **Picture** control with the mouse
useImage	Sets the image to be displayed without creating a copy of the image in transient cache

For details, see "Window, Form, and Control Methods", later in this document.

Picture Events

The event methods defined in the **Picture** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
mouseDown	When hen the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
scrolled	When the user scrolls the picture control
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

ProgressBar Class

The **ProgressBar** class enables you to display a progress bar in your label controls. As the **ProgressBar** class is a subclass of the **Label** class, it inherits all of the properties and methods defined in the **Label**, **Control**, and **Window** classes.

The **ProgressBar** control is not supported on forms defined as Web pages and is ignored when HTML is generated.

For a summary of the properties, method, and event defined in the **ProgressBar** class, see "ProgressBar Properties", "ProgressBar Method", and "ProgressBar Event", in the following subsections.

ProgressBar Properties

The properties defined in the **ProgressBar** class are summarized in the following table.

Property	Description
partsDone	Contains information about the percentage of the task that is completed
partsInJob	Contains information about the number of parts in the job
showTaskBarProgress	Specifies whether the progress bar state is shown on the taskbar icon of the application as well as in the progress bar
thinClientUpdateInterval	Specifies how often the progress bar is redrawn when the percentage changes when running the application in thin client mode

For details, see "Window, Form, and Control Properties", later in this document.

ProgressBar Method

The method defined in the **ProgressBar** class is summarized in the following table.

Method	Description
create	Creates the progress bar

For details, see "Window, Form, and Control Methods", later in this document.

ProgressBar Event

The event method defined in the **ProgressBar** class is summarized in the following table.

Event	Description
paint	Occurs when part or all of the form or control is exposed

For details, see "Window, Form, and Control Events", later in this document.

ScrollBar Class

The scroll bar control provides easy navigation through a long list of items or a large amount of information. It can also provide an analog representation of current position.

You can use a scroll bar as an input device or as an indicator of speed or quantity.

The **HScroll** and **VScroll** classes (the horizontal and vertical scroll bar control classes) are subclasses of the **ScrollBar** class (a subclass of the **Control** class). These classes therefore inherit all of the properties and methods defined in the **ScrollBar**, **Control**, and **Window** classes.

The horizontal and vertical scroll bar controls are not supported on forms defined as Web pages and they are ignored when HTML is generated.

When you use a scroll bar as an indicator of quantity or speed or as an input device, use the **max** property and the **min** property to set the appropriate range of the control.

As scroll bars use the Windows scroll bar API calls that set the thumb size to a size that reflects the size of the scroll bar range and the value of the **largeChange** property, the larger the value of the **largeChange** property, the larger the thumb size. To specify the amount of change to report in a scroll bar, use the **largeChange** property for clicking in the scroll bar and the **smallChange** property for clicking the arrows at the ends of the scroll bar. The **value** property increases or decreases by the amounts set for the **largeChange** and **smallChange** properties.

You can position the scroll bar at run time, by setting the **value** property.

For details about the support of mouse wheel requests to scroll up, down, or across a form or control, see "Window Class", earlier in this document. For details about dragging the scroll bar thumb of **ListBox** and **Table** controls when an **Array**, **Dictionary**, or **Set** collection is attached to the list box or table, see the **displayCollection** method.

For a summary of the properties and events defined in the **ScrollBar** class, see "ScrollBar Properties" and "ScrollBar Events", in the following subsections.

ScrollBar Properties

The properties defined in the **ScrollBar** class are summarized in the following table.

Property	Description
largeChange	Contains the amount of change when the user clicks the area between the scroll box and scroll arrow
max	Contains a scroll bar position when the scroll box is in its lowest position or farthest right position
min	Contains a scroll bar position when the scroll box is in its highest position or farthest left position
smallChange	Contains the amount of change when the user clicks a scroll arrow
value	Contains the current position of the scroll bar

For details, see "Window, Form, and Control Properties", later in this document.

ScrollBar Events

The event methods defined in the **ScrollBar** class are summarized in the following table.

Event	Occurs…
change	At the end of the scrolling process to indicates that the scroll box portion of the scroll bar has moved
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
gotFocus	When a control receives the focus
keyDown	When the user presses a key while the control has the focus
keyPress	When the user presses and releases an ANSI key

Event	Occurs…
keyUp	When the user releases a key while the control has the focus
lostFocus	When a control loses the focus
paint	When part or all of a form or control is exposed
scrolled	As the scroll bar control is scrolled
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Sheet Class

The **Sheet** class enables you to create a sheet control for a folder.

A sheet is the same as a **GroupBox** control, in that it holds a series of painted controls. Only one sheet is visible at any time, but tabs for each sheet are displayed at the top of the folder or at a user-specified edge. These tabs can be clicked to make the selected sheet visible. When the user clicks on a tab, the corresponding sheet is displayed (that is, both the **click** and the **sheetChg** event methods occur).

A specific sheet can also be selected, by using the keyboard if the caption of the sheet includes an accelerator sequence (for example, pressing Alt+C selects the sheet labeled **&Customers**), or you can use the Ctrl+Page Up or Ctrl+Page Down shortcut keys to move the focus of a folder to the prior or next enabled visible sheet when that folder or a child of the folder has focus.

As the **Sheet** class is a subclass of the **GroupBox** control class, it inherits all of the properties and methods defined in the **GroupBox**, **Control**, and **Window** classes.

For a summary of the property and methods defined in the **Sheet** class, see "Sheet Property" and "Sheet Methods", in the following subsections.

Sheet Property

The property defined in the **Sheet** class is summarized in the following table.

Property	Description
icon	Contains the icon displayed for a sheet at run time

For details, see "Window, Form, and Control Properties", later in this document.

Sheet Methods

The methods defined in the **Sheet** class are summarized in the following table.

Method	Description
canBeChildOf	Returns whether the control can be placed on the specified form or control
isMoveable	Returns **false** to specify that the control cannot be moved in the JADE Painter
isSizeable	Returns **false** to specify that the control cannot be resized in the JADE Painter

For details, see "Window, Form, and Control Methods", later in this document.

StatusLine Class

A status line control is a special frame control that aligns itself to the bottom and width of its container by default, providing a status line for the container.

Use the status line to display text that the user cannot directly change.

The status line has 3D effects by default. The status line text is always displayed within the area bounded by the 3D borders.

As the **StatusLine** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** and **Window** classes.

The status line is in effect a **Frame** control with the **alignContainer** property set to align to the bottom of the container. When the container is resized, the status line control is also resized. A **StatusLine** control that is aligned to their parent by using the **alignContainer** property will not scroll and it will remains in place in its parent when the scroll bar of the parent is shifted. It therefore remains visible and unchanged when the scroll bar of the parent is adjusted.

The status line has additional auto-size features beyond those of a label. Because the status line can also be a container, when controls are placed inside the status line, the control resizes in height to fit the controls placed inside it when the **autoSize** property is set to **true**. For example, multiple text boxes or labels could be inserted into the status line to display discrete pieces of information on the status line.

When the value of the **StatusLine** control **autoSize** property is **true**:

- If a child control **left** property value position is less than zero, the control is moved to be zero (**0**) when the **parentAspect**, **relativeLeft**, and **relativeWidth** property values of the child do not affect the horizontal position (not stretch horizontal, anchor right, and centered horizontal, and the **relativeLeft** and **relativeWidth** property values are **false**).
- If a child control is not fully visible horizontally, the child is right-aligned in the status line control if it can be fully displayed or positioned at zero (**0**) if it cannot when the **parentAspect**, **relativeLeft**, and **relativeWidth** property values of the child do not affect the horizontal position (not stretch horizontal, anchor right, and centered horizontal, and the **relativeLeft** and **relativeWidth** property values are **false**).
- If a child control top position is less than zero (**0**), the control is moved to be zero when the **parentAspect** property value of the child does not affect the vertical position (not stretch vertical, anchor bottom, or centered vertical).
- If a child control is not fully visible vertically, the child is bottom-aligned in the **StatusLine** control when the **parentAspect** property value of the child does not affect the vertical position (not stretch vertical, anchor bottom and centered vertically).
- The values of the **relativeTop** and **relativeHeight** properties of child controls are always set to **false**, as their functionality is not compatible with auto-sizing the height of the **StatusLine** control (as has always been the case).
- All **parentAspect** flag values and **relativeLeft** and **relativeWidth** values are applied.

The height of the **StatusLine** control is then determined by analyzing the child control as follows, to determine the maximum height required. For a child control that does not have a fixed height and has the **parentAspect** property with the:

- **ParentAspect_StretchBottom** flag set, the height required is the **top** position, **height**, and **parentBottomOffset** property values of the child.
- **ParentAspect_AnchorBottom** flag set, the height required is the **height** and **parentBottomOffset** property values of the child; otherwise, the **height** of the child control.

Note For a frame control or status line control, the position (**0,0**) is the top left of the area inside the 3D frame. Controls with the **StatusLine** class as its parent are not painted in the border area.

For a summary of the constants, properties, methods, and events defined in the **StatusLine** class, see "StatusLine Class Constants", "StatusLine Properties", "StatusLine Methods", and "StatusLine Events", in the following subsections.

For details about docking status line controls, see the **JadeDockBar**, **JadeDockBase**, and **JadeDockContainer** classes, earlier in this document.

StatusLine Class Constants

The constants provided by the **StatusLine** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
Alignment_Center_Bottom	8	Alignment_Center_Middle	7
Alignment_Center_Top	6	Alignment_Left_Bottom	2
Alignment_Left_Middle	1	Alignment_Left_Top	0
Alignment_Right_Bottom	5	Alignment_Right_Middle	4
Alignment_Right_Top	3	Bevel_Inset	1
Bevel_None	0	Bevel_Raised	2
BoundaryBrush_Dotted	1	BoundaryBrush_Solid	0

StatusLine Properties

The properties defined in the **StatusLine** class are summarized in the following table.

Property	Description
alignment	Contains the placement of text in the control
autoSize	Specifies whether a control is automatically resized to fit its contents
bevelColor	Contains the color used to paint the bevel areas of a 3D control
bevelInner	Contains the style of the inner bevel of the control
bevelInnerWidth	Contains the width of the bevel along the four sides of the control to determine the height of the three-dimensional shadow effect
bevelOuter	Contains the style of the outer bevel of the control
bevelOuterWidth	Contains the width of the bevel along the four sides of the control to determine the height of the three-dimensional shadow effect
bevelShadowColor	Contains the color used to paint the bevel areas of a 3D control
boundaryBrush	Specifies whether the boundary area is of a plain color or is painted with a dotted brush
boundaryColor	Contains the color of the boundary area
boundaryWidth	Contains the width of the boundary area

Property	Description
caption	Contains the caption for the control
clipControls	Specifies whether the Windows environment creates a clipping region that excludes controls contained by the object
wordWrap	Specifies whether text displayed in a caption advances to the next line of the control when the current line is filled

For details, see "Window, Form, and Control Properties", later in this document.

StatusLine Methods

The methods defined in the **StatusLine** class are summarized in the following table.

Method	Description
isMoveable	Returns **false** to specify that the control cannot be moved in the JADE Painter
isSizeable	Returns **false** to specify that the control cannot be resized in the JADE Painter

For details, see "Window, Form, and Control Methods", later in this document.

StatusLine Events

The event methods defined in the **StatusLine** class are summarized in the following table.

Event	Occurs…
click	When the user presses and then releases the left mouse button
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a form or control is exposed
sysNotify	When a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Table Class

A **Table** control enables you to display entries in a table using rows and columns. As the **Table** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** and **Window** classes.

The table control also enables you to have many different sheets in the table, of which only one is visible at any time. The heading for each sheet is displayed in a selectable tab at the top of the control. When the user clicks on that tab, the corresponding sheet of the table is displayed (that is, both the **click** and the **sheetChg** events occur).

The number of rows and columns on a sheet is controlled by the **rows** and **columns** properties. Changing the value of the **column** or **row** property does not cause the table to be repainted. **Table** control subclasses do not reference **mousePointer** in a **mouseDown** event to determine if the mouse is in the resize position (that is, positioned over the dividing lines between fixed columns or rows).

Use the **allowResize** property to specify whether users can resize the rows and columns of a table. (The **resizeColumn** and **resizeRow** event methods are called when a resize operation has been performed using the mouse on a table column or row, respectively.)

The **Table** control class has the following behavior.

- Clicking on an enabled fixed row or column cell draws the cell in a slightly darker background color. When the mouse is released, the original background color is restored.

  This gives the user a visual indication that the cell was clicked.

- Clicking on a disabled cell generates table **mouseDown** and **mouseUp** events, but does not generate a **click** event after the mouse is released.

- Resizing a row or column generates table **mouseDown**, **mouseUp**, and **resizeRow** or **resizeColumn** events, but does not generate a **click** event after the mouse is released.

  Resizing a row or column does not resort the row or column.

- Moving a row or column generates table **mouseDown**, **mouseUp**, and **queryRowMove** or **queryColumnMove** events, but does not generate a **click** event after the mouse is released.

  Moving a row or column does not resort the row or column.

Table cells on a form defined as a Web page can contain pictures and labels only.

The table enables you to control the background color, text color, alignment, and picture displayed for each cell. Each cell can also be input-capable.

Although the table control can display a maximum of 63 sheets, the resulting size of the tabs prevents this limit being a reasonable option. Each sheet can have a maximum of 16,000 columns and 32,000 rows. These maximum limits cannot be achieved, however, as too much memory is required to be able to be run the application effectively.

The table can also be defined with fixed rows and columns. Fixed columns are usually the headings for each row and column. These rows and columns are always displayed, and do not scroll with the rest of the table.

Set the **dropDown** property to enable the table to fold up when focus is lost, showing only the current row.

Each sheet can be sorted by using the contents of one through six columns. Rows and columns on a sheet can be hidden.

By default, a table has one defined sheet. Use the **sheets** property to control the number of sheets in the table. The current sheet that is being accessed by logic is defined by the **sheet** property. The current sheet that is displayed is defined by the **topSheet** property.

The **rows** and **columns** properties control the number of rows and columns in each sheet. Rows can also be added to the table by using the **addItem** and **addItemAt** methods. If you add rows by using either of these methods, the text for each cell should be concatenated into one string, with the text of each cell separated by a tab character.

Use the **text** property to access the text for each cell. The current sheet, row, and column are then used to access the text in that cell. When setting the **text** property, the current cell in the current row is filled with the text that is passed. If the text contains tab characters, this character is assumed to be the end of the text for a cell, and the next cell is then filled with the remainder of the text, and so on. If all cells in the row are filled and there is more text, the remainder is ignored.

As table controls do not support a row height larger than the table, only the visible part of the row is displayed.

Move around the cells of a table by using the mouse, arrow keys, or Home, End, Page Up, or Page Down keys. Tables are scrolled by rows, not by pixels. When an input-capable cell has the focus, it may be necessary to also press the Alt key, as navigation keys may also have meaning to the text or combo boxes in these cells.

Press Ctrl+Home to move a table with an associated collection to the first entry in the collection. Conversely, press Ctrl+End to move the display to the last entry in the collection.

Note Pressing the Home key on a table moves the display to the first entry that has been loaded in the control. Pressing the End key moves the display to the last entry currently loaded in the control. Using the Ctrl key for a list box that does not have a collection attached has the same result as pressing the Home or End key without the Ctrl key.

You can set the width of a column to a percentage of the client width of a table. For details, see the **widthPercent** property of the **JadeTableColumn** class. To control how any columns percentages set on the table sheet using the **widthPercent** property are interpreted, use the **widthPercentStyle** property of the **JadeTableSheet** class.

When writing your logic to handle **Table** controls, you can:

- Set object references on sheets, rows, columns, and cells by using the **accessSheet**, **accessRow**, **accessColumn**, and **accessCell** methods to set the respective **accessedSheet**, **accessedRow**, **accessedColumn**, and **accessedCell** properties.

Note The effect of logic changing a font property for a sheet, row, column, or cell that has no font settings of its own is that all the other font properties are also set from the default font that applies.

The **accessMode** property provides multiple meanings for the **alignment**, **backColor**, **comboIndex**, **foreColor**, **fontName**, **fontSize**, **fontBold**, **fontItalic**, **fontUnderline**, **fontStrikethru**, **inputType**, **maxLength**, **partialTextIndication**, and **wordWrap** properties of **Table** controls.

- Search the **itemObject** property values of rows, columns, and cells for an object reference.
- Select or deselect an entire sheet, row, or column by using the **selectMode** property.
- Sort columns based on the type of cell data (that is, alphanumeric, numeric, date, time, or timestamp), by using the **sortType** property.
- Control automatic row and column sizing based on the text content of cells, by using the **autoSize** property.

  Note Text in table cells wraps only when the value of the **autoSize** property is set to **AutoSize_Row** (**1**).

- Control vertical or horizontal alignment within a cell, by using **Table** class constant values for the **alignment** property; for example, **Alignment_Center_Bottom**.
- Control tabbing within a table, by using the **tabKey** property.

- Specify a numeric input type allowing decimals and negative values, by using the **decimals** property and **Table** class constant values for the **inputType** property; for example, **InputType_SignedNumeric**.
- Handle user-created controls for input, by using the **cellControl** and **userInputEnabled** properties and the **cellInputReady** event method. Alternatively, you can use the **Control** class **automaticCellControl** property to control table cells (for example, when running in JADE thin client mode over a slow link).
- Optionally allow automatic dragging and dropping of a column or row, by using the **allowDrag** property.
- Provide the **displayCollection** method to invoke the **displayRow** event that automatically attaches a collection to the current sheet of a table and displays it.

  For details about dragging the scroll bar thumb of tables when an **Array**, **Dictionary**, or **Set** collection is attached to the table, see the **displayCollection** method.
- Retrieve the cell that corresponds to a position on the table, by using the **positionLeft** and **positionTop** methods.
- Control the optional display of the points of ellipsis symbol (**...**) on the end of the visible text when the column is not wide enough to show all the text, by using the **partialTextIndication** property.
- Use the **foreColor** property of a sheet to display the caption of a sheet in a table.
- Enable or disable all set **inputType** or **cellControl** properties, by using the **userInputEnabled** property.
- Control what automatic selections are made when the user clicks the table or uses the Shift key or the Ctrl key, by using the **selectMode** property.
- Specify whether the picture or text is truncated when there is insufficient room to display both in a cell, by using **Table** class constant values for the **stretch** property; for example, **Stretch_Cell_Picture_First**.

Note When the text of a sorted column changes, the automatic sorting of rows occurs only when the **Table** class **addItem** method adds a new row or the **Table** class **resort** method is used.

When a collection is attached to a table and the current row is scrolled so that it will be discarded (outside of the displayed virtual window over the collection), the **queryRowColChg** and **rowColumnChg** event methods are called. If the **queryRowColChg** event is rejected, the scroll action is discarded. (Note that the **queryRowColChg** event passes the current **row** and **column** values as parameters, because the actual current row number remains unchanged after scrolling.)

For an example and details about merging cells in a table, see the **JadeTableCell** class **mergeCells** property and **getCellWidth** method. For details about the support of mouse wheel requests to scroll up, down, or across a table control, see "Window Class", earlier in this document.

Note For the arrays associated with tables (for example, **Table.columnVisible**), the only methods that are implemented are **at**, **atPut** (which enables you to use the square brackets notation to access the elements), **createIterator** (which allows logic to do a **foreach** over the array), **size**, and **size64**.

For an overview of directly accessing properties and methods in a table element (for example, a cell or a row) without using the **accessMode** property, see "Directly Accessing Table Elements". For a summary of the constants, properties, methods, and events defined in the **Table** class, see "Table Class Constants", "Table Properties", "Table Methods", and "Table Events", in the following subsections.

Directly Accessing Table Elements

The **Table** control provides access to an internally created sheet, row, column, and cell object; that is, the **JadeTableElement** class and its **JadeTableCell**, **JadeTableColumn**, **JadeTableRow**, and **JadeTableSheet** subclasses. You can use these objects to directly access the properties and methods of the object without using the **accessMode** property to access the tables (and therefore reducing the amount of JADE logic required to handle tables).

Using instances of these subclasses is equivalent to setting the **accessMode** property of the **Table** control, summarized in the following table.

JadeTableElement Subclass	Equivalent to the accessMode value for the Table class…
JadeTableCell	Table.AccessMode_Cell
JadeTableColumn	Table.AccessMode_Column
JadeTableRow	Table.AccessMode_Row
JadeTableSheet	Table.AccessMode_Sheet

For details, see the appropriate class in Chapter 1.

To eliminate the overhead of creating an object for each cell, column, row, and sheet of the table, only one object of each type is created, which is essentially a proxy object that holds the last reference to the cell, column, row, or sheet that was last accessed.

Accessing a cell, column, row, or sheet sets a corresponding property in the **Table** class that you can then use to subsequently access that table element, as follows.

- **accessCell** method sets the **accessedCell** property to the returned cell
- **accessColumn** method sets **accessedColumn** property to the returned column
- **accessRow** method sets **accessedRow** property to the returned row
- **accessSheet** method sets **accessedSheet** property to the returned sheet

The following code fragments show examples of accessing the last table elements that were accessed.

```
table1.accessCell(2, 3).inputType := Table.InputType_TextBox;
table1.accessedCell.foreColor := Red;

table1.accessSheet(2).accessCell(1, 4).text := "Company";
table1.accessedCell.alignment := Table.Alignment_Right_Middle;
```

Storing a reference to a returned cell causes problems unless you take a copy of that cell, as shown in the following example in which both **cell1** and **cell2** refer to the same object, which is referencing **cell(3, 4)**.

```
cell1 := table1.accessCell(2, 3);
cell2 := table1.accessCell(3, 4);
cell1.text := "abc";
```

In the following example, **cell1** has been cloned and still refers to **cell(2, 3)**.

```
cell1 := table1.accessCell(2, 3).cloneSelf(true);
// the cloned cell must be deleted by your logic
cell2 := table1.accessCell(3, 4);
cell1.text := "abc";
```

Table Class Constants

The constants provided by the **Table** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
AccessMode_Cell	1	AccessMode_Column	2
AccessMode_Row	3	AccessMode_Sheet	0
Alignment_Center	3	Alignment_Center_Bottom	9
Alignment_Center_Middle	3	Alignment_Center_Top	6
Alignment_Default	0	Alignment_Left	1
Alignment_Left_Bottom	7	Alignment_Left_Middle	1
Alignment_Left_Top	4	Alignment_Right	2
Alignment_Right_Bottom	8	Alignment_Right_Middle	2
Alignment_Right_Top	5	AllowDrag_Both	3
AllowDrag_Columns	2	AllowDrag_None	0
AllowDrag_Rows	1	AllowResize_Any	0
AllowResize_Column	1	AllowResize_None	3
AllowResize_Row	2	AutoSize_Both	3
AutoSize_BothColumnMinimum	5	AutoSize_Column	2
AutoSize_ColumnMinimum	4	AutoSize_None	0
AutoSize_Row	1	DisplayCollection_Forward	0
DisplayCollection_NoPrior	0	DisplayCollection_Prior	2
DisplayCollection_Reversed	1	DisplaySorting_AllColumns	2
DisplaySorting_First	1	DisplaySorting_None	0
DisplaySorting_Numbers	3	DropDown_Auto	2
DropDown_Click	1	DropDown_None	0
InputType_CheckBox	1	InputType_ComboBox	3
InputType_Default	5	InputType_EditMask	7
InputType_None	0	InputType_SignedNumeric	6
InputType_TextBox	2	InputType_TextNumeric	4
MergeCells_Available	0	MergeCells_Merge	1
MergeCells_MergeSelectable	2	MergeCells_NotAvailable	3
ScrollMode_Both_Pixel	3	ScrollMode_Cell	0
ScrollMode_HorzPixel_VertCell	1	ScrollMode_VertPixel_HorzCell	2
SelectMode_Column	5	SelectMode_CurrentColumn	10
SelectMode_CurrentRow	8	SelectMode_Default	0

Constant	Integer Value	Constant	Integer Value
SelectMode_FixedColumn	2	SelectMode_FixedRow	1
SelectMode_Multiple	3	SelectMode_None	7
SelectMode_Row	4	SelectMode_Single	6
SelectMode_WholeColumns	11	SelectMode_WholeRows	9
SortType_Alpha	0	SortType_Date	2
SortType_Numeric	1	SortType_Time	3
SortType_TimeStamp	4	Stretch_Cell	1
Stretch_Cell_Picture_First	3	Stretch_None	0
Stretch_None_Picture_First	2	TabInitialPosition_First	1
TabInitialPosition_First_Last	3	TabInitialPosition_Last	2
TabInitialPosition_None	0	WidthPercent_Style_ClientWidth	0
WidthPercent_Style_NoSetWidths	1		

Table Properties

The properties defined in the **Table** class are summarized in the following table.

Property	Description
accessedCell	Contains a reference to the cell returned by the **accessCell** method
accessedColumn	Contains a reference to the column returned by the **accessColumn** method
accessedRow	Contains a reference to the row returned by the **accessRow** method
accessedSheet	Contains a reference to the sheet returned by the **accessSheet** method
accessMode	Enables table cells to have different colors, fonts, alignments, and input modes
alignment	Contains the placement of text in a cell
allowDrag	Enables a row or column to be dragged to a new position within the table
allowResize	Specifies whether users can resize the rows and columns of a table
autoSize	Controls whether the table automatically sets the width of columns and the height of rows to fit the size of the contents of the cells
cellControl	Controls the input and display within a table by defining a user-specified control that is placed over the cell when that cell becomes current
column	Contains the current column on the current sheet
columns	Contains the number of columns on the current sheet
columnVisible	Specifies whether a column is displayed or hidden, or the visibility status
columnWidth	Contains the size of a column to be accessed
comboIndex	Contains the index of a combo box in a cell
comboList	Contains the list entries displayed in a combo box in the sheet, column, row, or cell

Property	Description
decimals	Specifies that a cell controlled by the **accessMode** property can accept decimal input for cells with a numeric text or signed numeric input type
defaultRowHeight	Specifies the default height of rows in a table independent of the font size
displayHotKey	Specifies whether the character following an ampersand (&) character is underlined
dropDown	Specifies that a table need only occupy the space required for one row and yet still provide all the features of an expanded display
editMask	Sets the mask used for edit mask input for a cell, row, column, or sheet
expandedHeight	Contains the height of the table in pixels when it is expanded
fixed3D	Specifies whether a 3D button image is painted on the cells in the fixed area
fixedColumns	Contains the number of fixed columns in a table
fixedRows	Contains the number of fixed rows in a table
gridColor	Contains the color of grid lines
gridLines	Specifies whether lines are drawn between the rows and columns of the current sheet
hyperlinkColumn	Contains an array of integers that represent a column in each row of the table
inputType	Contains the type of input (if any) that is accepted by a cell, row, column, or sheet
itemObject	Contains an object for each cell
leftColumn	Contains the column that is displayed at the left edge of the non-fixed area of the current sheet
maxLength	Contains the amount of text that can be entered into a cell
partialTextIndication	Specifies if an indication is displayed when there is insufficient room to show all text of a cell
picture	Contains a graphic to be displayed in a cell
readOnly	Specifies whether a control is read-only for user input
row	Contains the current row on the current sheet
rowHeight	Contains the size of a row of a table control
rows	Contains the number of rows on the current sheet
rowVisible	Specifies whether a row is displayed or hidden
scrollBars	Specifies whether the table has horizontal or vertical scroll bars when required
selected	Contains the selected status of the current cell on the current sheet
selectMode	Specifies the selections that are made automatically when the user selects a cell in the table
sheet	Contains the index of the current sheet
sheetCaption	Contains the caption for the current sheet
sheets	Contains the number of sheets
sheetVisible	Specifies whether a sheet is visible

Property	Description
showFocus	Specifies whether the focus rectangle is shown on the current cell of a table when it has focus
sortAsc	Specifies whether the sorting is ascending or descending
sortCased	Specifies whether the sorting is case-sensitive
sortColumn	Contains the column number for which the text is to be sorted
sortType	Specifies the type of data the cell text represents
stretch	Specifies whether pictures placed in the cells are drawn to fit the cell
tabActiveColor	Contains the color drawn for the active tab of multiple sheet table
tabInactiveColor	Contains the color drawn for the inactive tabs of multiple sheet tables
tabKey	Specifies the key that is used to tab within cells of the table
text	Contains the text in the current cell on the current sheet
topRow	Contains the row that is displayed at the top edge of the non-fixed area of the current sheet
topSheet	Contains the sheet that is currently visible
userInputEnabled	Specifies whether input (**inputType** and **cellControl** property actions) for the table cells is enabled or disabled
wordWrap	Specifies whether text displayed in a cell is displayed using word wrap when the width of the cell is less than the length of the text

For details, see "Window, Form, and Control Properties", later in this document.

Table Methods

The methods defined in the **Table** class are summarized in the following table.

Method	Description
accessCell	Returns a reference to the cell that is currently accessed at run time
accessColumn	Returns a reference to the column that is currently accessed at run time
accessRow	Returns a reference to the row that is currently accessed at run time
accessSheet	Returns a reference to the sheet that is currently accessed at run time
addItem	Adds a new row to a table control
addItemAt	Adds a specified new row to a table control
clear	Clears the contents of the current sheet
clearAllSelected	Clears all the **selected** properties of all cells for the current sheet
create	Creates the table
deleteColumn	Deletes the specified column from the current sheet
deleteRow	Deletes the specified row from the current sheet
deleteSheet	Deletes the specified sheet

Method	Description
displayCollection	Attaches the specified collection to the current sheet of the table
dragColumn	Provides table-specific location information of the drag and drop processes
dragRow	Provides table-specific location information of the drag and drop processes
dragSheet	Provides table-specific location information of the drag and drop processes
getCellFromPosition	Returns the cell at the specified position and the row and column of that cell
getCellSelected	Returns the selected status of the cell
getCellText	Returns the text of the cell
getCollection	Returns the collection attached to the current sheet of the table by the **displayCollection** method or **ListBox** class **listCollection** method
hasPicture	Returns **true** if the current cell indicated by the **sheet**, **column**, and **row** properties has a picture displayed
insertColumn	Enables a single column to be inserted into the current sheet
isHyperlinkSet	Specifies whether a HyperText link is set for the specified row and column
loadCollectionEntries	Accesses any collection entries required to fill the display size of the table sheet
moveColumn	Moves a column of the current sheet
moveRow	Moves a row of the current sheet
positionLeft	Returns the displayed position of the current cell in pixels
positionTop	Returns the displayed position of the current cell in pixels
removeItem	Removes a row from the current sheet at run time
refreshEntries	Refreshes the displayed list of entries on the current sheet of the table
resetAllHyperlinks	Clears all HyperText links that were set by using the **setHyperlinkCell** method
resetHyperlinkCell	Clears the HyperText link from the specified row and column
resort	Resorts the contents of the current sheet
selectedCount	Returns the number of cells with the selected status set
selectedNext	Returns the next selected cell
setCellSelected	Sets the selected status of a cell
setCellText	Sets the text of the cell
setCollectionObject	Sets the object in the collection attached to the table, ensuring that the object referenced is in the displayed list of collection entries for the table
setHyperlinkCell	Sets up a HyperText link for the cell in the specified row and column

For details, see "Window, Form, and Control Methods", later in this document.

Table Events

The event methods defined in the **Table** class are summarized in the following table.

Event	Description
cellInputReady	Occurs when the cell is selected and after it has been resized to overlay the cell and made visible, to allow the control specified by the **cellControl** property to be initialized
change	Occurs when the contents of a cell have been changed by user input
click	Occurs when the user presses and then releases the left mouse button
closeup	Occurs when the table folds up
contextMenu	Occurs after the right **mouseUp** event and after the **keyUp** event
dblClick	Occurs when the user presses and releases the left mouse button and then presses and releases it again
displayRow	Occurs for each entry in the collection of the current sheet, to display the contents of the row
dragDrop	Occurs when a dragged window is dropped over a window belonging to the same application
dragOver	Occurs for each window of the application over which a window is dragged
gotFocus	Occurs when a control receives the focus
keyDown	Occurs when the user presses a key while the control has the focus
keyPress	Occurs when the user presses and releases an ANSI key
keyUp	Occurs when the user releases a key while the control has the focus
lostFocus	Occurs when a control loses the focus
mouseDown	Occurs when the user presses a mouse button
mouseEnter	Occurs when the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	Occurs when the user moves the mouse off a control
mouseMove	Occurs when the user moves the mouse
mouseUp	Occurs when the user releases a mouse button
openup	Occurs when the table opens up to its full size
paint	Occurs when part or all of a control is exposed
queryColumnMove	If defined, occurs when the user releases the mouse after dragging a fixed column to a new position when set to **true**
queryRowColChg	Occurs when the user selects a new cell or sheet using the keyboard or the mouse
queryRowMove	If defined, occurs when the user releases the mouse after dragging a fixed row to a new position when set to **true**
resizeColumn	Occurs after the user has resized a table column using the mouse
resizeRow	Occurs after the user has resized a table row using the mouse

Event	Description
rowColumnChg	Occurs when the user selects a different cell
scrolled	Occurs when the user scrolls
sheetChg	Occurs when the user clicks on the tab of a sheet that is not currently the top sheet, to enable that sheet to become visible
sysNotify	Occurs when a specified JADE system event occurs
userNotify	Occurs when triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

TextBox Class

A text box control, sometimes called an edit field or edit control, can display information entered in the JADE development environment, entered by the user, or assigned to the control by logic at run time.

As the **TextBox** class is a subclass of the **Control** class, it inherits all of the properties and methods defined in the **Control** and **Window** classes. The **TextBox** class provides the **WebHTML** subclass.

The **TextBox** class validates the setting and entry of text based on the **dataType** property value, using the current locale of the client with regional overrides on both the presentation client and the application server when the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is set to **true**.

By setting appropriate property values, the text box control can:

- Be a single-line data entry text box
- Be a multiple-line data entry text box
- Allow the word wrap feature (implied when the **scrollHorizontal** property is set to **false**)
- Allow scrolling of the text both horizontally and vertically

  For details about the support of mouse wheel requests to scroll up, down, or across a text box control, see "Window Class", earlier in this document.
- Align the text left, right, or centrally
- Be a password entry text box, where the characters entered are displayed as asterisk (*) characters
- Be declared as a numeric text box, preventing the entry of invalid characters or non-numeric characters
- Automatically convert the case of the entered characters
- Be a read-only display text box
- Offset the left and right margins, to improve the appearance of the displayed text

Tip Use the **firstChange** event to move logic from other key events when that logic is relevant only to the first time the text changes. (This reduces the number of events that must be sent and processed for each key that is pressed.)

The text box is always created as a multiple line text box, unless the **passwordField** property is set to **true**. In a text box, **CrLf** and **Lf** are recognized as end-of-line sequences.

The text box functions as a single-line text box when:

- The height of the text box leaves room only for one line of text to be displayed and the **scrollVertical** property is set to **false**
- The **dataType** property is set to that of a numeric text box

Control the functionality of the text box as a multiple-line control by setting the **scrollHorizontal**, **scrollVertical**, and **scrollBars** properties. If the **scrollHorizontal** property is set to **true**, word wrapping does not occur, and each line must be terminated by the Enter key. If the **scrollVertical** property is set to **false**, the entry of text is limited to the number of visible text lines.

The text box does not receive tab characters from Windows unless the text box is the only enabled visible control that has the **tabStop** property set, as the tab functions is interpreted by Windows as a skip to the next text box. If there is no other text box to tab to, the character is passed to the text box. Use Ctrl+Tab to enter a tab when the tab is interpreted as a control skip.

Note By default, the text box does not respond to the Enter key if an enabled and visible default button is defined on the same form. Use the **TextBox** class **wantReturn** property to make the text box respond to the Enter key.

If the text box is defined as numeric data type, the text cannot be set to a value that is an invalid numeric (according to the **dataType**, **decimals**, and **maxLength** properties). Additionally, as each character is entered, the character is rejected with a beep if the resulting text would result in an invalid number (including the deletion of a character).

If the **case** property is set to **Case_UpperFirst** (**3**) or **Case_LowerFirst** (**4**), the text box control accepts only an alphabetic character as the first character entered in the text box and automatically changes the case of the entered character to the required type, if applicable. If the **case** property is set to any value other than the default value of **Case_None** (**0**), the **dataType** property is set to the default value of **DataType_AlphaNumeric** (**0**) and the **decimals** property is set to zero (**0**); that is, no exception is raised.

You can specify hint text and colors for a text box; for example, text that can be displayed when the **text** property is empty, to advise the user what data is required. The **TextBox** control provides the properties listed in the following table.

Property	Value	Default	Description
hintText	String	An empty string	Text that is displayed in an empty text box as a hint
hintBackColor	Integer	#80000000	Background color of hint text
hintForeColor	Integer	#80000000	Color of the hint text

If the value of the **hintText** property is:

- Empty, these properties have no impact on the text box display
- Not empty, the specified hint text is displayed in the text box when the value of the **text** property is empty

When the value of the **hintText** property is displayed, the:

- **text** and **selText** properties return an empty string
- Text cannot be selected or deleted, and the cursor is always positioned at the beginning of the text

As soon as text is entered or pasted into the text box, the hint text is removed and replaced with the specified or pasted text. If the entire text is removed, the hint text is displayed again.

When hint text is displayed, the:

- Back ground of the text box is drawn using the value of the **hintBackColor** property except when it is **#80000000**, in which case the **TextBox** control **backColor** property value is used.
- Text of the text box is drawn using the value of the **hintForeColor** property except when it is **#80000000**, in which case the **TextBox** control **foreColor** property value is used.

The values of the **dataType**, **case**, and **selectionStyle** properties are ignored when the hint text is displayed; for example, hint text can be displayed for a numeric field. The hint text, which is always displayed in the case of its defined string, can never be selected. In addition, for a password text box, the hint text is displayed as clear text (that is, it is not displayed using asterisk (*****) characters).

When the value of the **hintBackColor** or **hintForeColor** property is **#80000000**, the value of the respective **hintBackColor** or **hintForeColor** property in the JADE Painter Properties dialog is displayed as **Use backColor Value** or **Use foreColor Value**, with an image at the left of the text displaying the current value of the respective **backColor** or **foreColor** property. The combo box drop-down list of colors includes an entry for **Use backColor Value** or **Use foreColor Value** that sets the property value to the default.

If the values of the **hintBackColor** and **hintForeColor** properties are both zero (**0**), the default values of **#80000000** (that is, transparent) are used instead.

Note A text box automatically handles the clipboard copy, cut, and paste key sequences (that is, Ctrl+Insert, Ctrl+C, Ctrl+Delete, Ctrl+X, and Shift+Insert).

If text is pasted into or cut from a text box and the resulting text is no longer valid according to the automatic validation rules, the text in the text box is cleared; for example, when pasting non-numeric text into a numeric text box. This ensures that JADE logic can always assume the resulting text complies with the validation rules. This situation occurs only with paste or cut operations that involve numeric text boxes and when the first character must be alphabetic (**case** property). When data is entered from the keyboard, the validation rules are applied, and the entered character is ignored (with a beep) if the resulting text would be invalid.

Note When text with trailing carriage return and line feed characters (the **CrLf** end-of-line sequence) is pasted after the current text of a text box, the **CrLf** characters are removed if the:

- Resulting text would be invalid (for example, pasting **'1234'** followed by a **CrLf** character into a numeric text box results in **'1234'** only being pasted)
- Text box has the **scrollVertical** property set to **false** and the **CrLf** characters would cause the number of lines of text to exceed the height of the available area in the text box (for example, pasting **'No'** followed by a **CrLf** character into a single-line text box with the **scrollVertical** property set to **false** results in **'No'** being pasted)

To implement the handling of a **TextBox** control to achieve the same functionality as the default **Table** class **InputType_TextBox** value, only the following is required.

- In the **change** event for the text box:

```
table1.text := textbox1.text
```

- In the **cellInputReady** event for the table:

```
textbox1.text := table1.text
```

Note You can use the Windows key + period (**.**) function keys combination to paste a selected emoji or selection of emojis from the emoji selection window into **TextBox** controls in a Unicode JADE system. For details, see "Unicode Surrogate Pair Characters, including Emojis", in Chapter 1 of the *JADE Developer's Reference*.

Pastes a selected emoji or selection of emojis from the emoji selection window into **TextBox** controls or the editor pane in a Unicode JADE system. For more details, see "Unicode Surrogate Pair Characters, including Emojis", in Chapter 1 of the *JADE Developer's Reference*.

For more details, see the **Table** class **inputType** and **cellControl** properties. See also the **Control** class **automaticCellControl** property and "Value Round Trips through TextBox Controls", elsewhere in this document. For a summary of the constants, properties, methods, and events defined in the **TextBox** class, see "TextBox Class Constants", "TextBox Properties", "TextBox Methods", and "TextBox Events", in the following subsections.

TextBox Class Constants

The constants provided by the **TextBox** class are listed in the following table.

Constant	Value	Constant	Value
Alignment_Center	2	Alignment_Left	0
Alignment_Right	1	Case_Lower	1
Case_LowerFirst	4	Case_None	0
Case_Upper	2	Case_UpperFirst	3
DataType_AlphaNumeric	0	DataType_Currency	3
DataType_LongDate	5	DataType_Numeric	1
DataType_ShortDate	4	DataType_SignedNumeric	2
DataType_Time	6	SelectionStyle_Hide	1
SelectionStyle_Retain	0	SelectionStyle_SelectAll	2
SelectionStyle_SelectAllAlways	3	Web_InputType_File	'F'
Web_InputType_Hidden	'H'	Web_InputType_Password	'P'
Web_InputType_Text	'T'	Web_InputType_TextArea	'A'

TextBox Properties

The properties defined in the **TextBox** class are summarized in the following table.

Property	Description
alignment	Contains the text position in the control
autoTab	Specifies whether focus is automatically moved to the next control in the tab order of the form, or to the next accessible cell (for a **TextBox** control in a table)
case	Contains the automatic case conversion of entered text
dataType	Contains the type of data that can be entered by the user
decimals	Specifies whether text is a decimal type numeric and the maximum number of decimal places text can have
formatOut	Contains system-defined formats of data in text boxes during printing
hintBackColor	Contains the text box back ground color when hint text is displayed

Property	Description
hintForeColor	Contains the hint text color when hint text is displayed
hintText	Contains the text displayed in an empty text box as a hint
integralHeight	Specifies whether the text box height is an exact number of lines of text
maxLength	Contains how much text can be entered
passwordField	Specifies whether the characters typed by a user or placeholder characters are displayed
readOnly	Specifies whether a control is read-only for user input
scrollBars	Specifies whether an object has horizontal or vertical scroll bars
scrollHorizontal	Specifies whether the text scrolls horizontally
scrollHorzPos	Contains the position of the horizontal scroll bar
scrollVertical	Specifies whether the text scrolls vertically
scrollVertPos	Contains the position of the vertical scroll bar
selectionStyle	Specifies whether selected text is highlighted when a control loses or gains the focus
selLength	Contains the number of selected characters
selStart	Contains the starting point of selected text
selText	Contains the string containing the currently selected text
text	Contains the text contained in the edit area
textOffset	Contains the pixel offset of the left and right margins for text displayed in a text box
wantReturn	Specifies whether carriage returns are passed to the text box when an enabled and visible default button is defined on the same form
webInputType	Contains the type of input that is accepted by a text box control on a Web page

For details, see "Window, Form, and Control Properties", later in this document.

TextBox Methods

The methods defined in the **TextBox** class are summarized in the following table.

Method	Description
firstVisibleLine	Returns the first visible line
getRegisteredKeys	Returns an array of the keys that are in effect for the text box
getScrollRange	Gets the scroll ranges
getTextAsCurrencyDecimal	Returns the value of the **text** property in currency format converted to a **Decimal** value
getTextAsCurrencyReal	Returns the value of the **text** property in currency format converted to a **Real** value
getTextAsDecimal	Returns the value of the **text** property converted to a **Decimal** value
getTextAsInteger	Returns the value of the **text** property converted to an **Integer** value

Method	Description
getTextAsInteger64	Returns the value of the **text** property converted to an **Integer64** value
getTextAsLongDate	Returns the value of the **text** property in long date format converted to a **Date** value
getTextAsReal	Returns the value of the **text** property converted to a **Real** value
getTextAsShortDate	Returns the value of the **text** property in short date format converted to a **Date** value
getTextAsTime	Returns the value of the **text** property converted to a **Time** value
isValid	Returns whether the value of the **text** property is valid according to the value of the **dataType** property
lineCount	Returns the number of lines of text
lines	Returns the number of lines available for display in the text box
registerKeys	Establishes the entire set of key codes in which the key events of a text box are interested
setTextFromCurrencyDecimal	Sets the **text** property value to a **Decimal** value converted to a **String** in the currency format of the locale under which the control is running
setTextFromCurrencyReal	Sets the **text** property value to a **Real** value converted to a **String** in the currency format of the locale under which the control is running
setTextFromDecimal	Sets the **text** property value to a **Decimal** value converted to a **String** in the format of the locale under which the control is running
setTextFromInteger	Sets the **text** property value to a **Integer** value converted to a **String** in the format of the locale under which the control is running
setTextFromInteger64	Sets the **text** property value to a **Integer64** value converted to a **String** in the format of the locale under which the control is running
setTextFromLongDate	Sets the **text** property value to a **Date** value converted to a **String** in the long date format of the locale under which the control is running
setTextFromReal	Sets the **text** property value to a **Real** value converted to a **String** in the format of the locale under which the control is running
setTextFromShortDate	Sets the **text** property value to a **Date** value converted to a **String** in the short date format of the locale under which the control is running
setTextFromTime	Sets the **text** property value to a **Time** value converted to a **String** in the time format of the locale under which the control is running

For details, see "Window, Form, and Control Methods", later in this document.

TextBox Events

The event methods defined in the **TextBox** class are summarized in the following table.

Event	Occurs…
change	When the contents of the control have changed
click	When the user presses and then releases the left mouse button

Event	Occurs…
contextMenu	After the right **mouseUp** event and after the **keyUp** event
dblClick	When the user presses and releases the left mouse button and then presses and releases it again
dragDrop	When a dragged window is dropped over a window belonging to the same application
dragOver	For each window of the application over which a window is dragged
firstChange	When the user performs keyboard or cut and paste actions
gotFocus	When a control receives the focus
keyDown	When the user presses a key while the control has the focus
keyPress	When the user presses and releases an ANSI key
keyUp	When the user releases a key while the control has the focus
lostFocus	When a control loses the focus
mouseDown	When the user presses a mouse button
mouseEnter	When the user moves the mouse onto a control
mouseHover	When the user moves the mouse onto a control and then the mouse remains static for one second or longer
mouseLeave	When the user moves the mouse off a control
mouseMove	When the user moves the mouse
mouseUp	When the user releases a mouse button
paint	When part or all of a control is exposed
scrolled	When the user scrolls
sysNotify	When it is triggered when a specified JADE system event occurs
userNotify	When triggered from the JADE Object Manager by a user call

For details, see "Window, Form, and Control Events", later in this document.

Value Round Trips through TextBox Controls

The following subsections provide code fragment examples of round trips (converting back and forth to and from primitive types in **TextBox** controls).

Number Round Trips

```
val       : Integer, Byte, Integer64, Decimal, Real;
str       : String;
txb       : TextBox;
errOffset : Integer;

//Binary to String
txb.dataType := DataType_Numeric, DataType_SignedNumeric;

txb.decimals := <as-required>;
txb.setTextFromType(val)     // direct
```

```
str := val.numberFormat();   // indirect
txb.text := str;

//String to Binary
val := txb.getTextAsType();  // direct

str := txb.text;             // indirect
if val.parseNumberWithCurrentLocale(str, errOffset) <> 0 then
    // invalid text
endif;
```

Currency Round Trips

```
val       : Decimal, Real;
str       : String;
txb       : TextBox;
errOffset : Integer;

//Binary to String
txb.dataType := DataType_Currency;      // overrides decimals

txb.setTextFromCurrencyDecimal(val);    // direct
txb.setTextFromCurrencyReal(val);

str := val.userCurrencyFmtAndLcid(null, 0);  // indirect
txb.text := str;

//String to Binary
val := txb.getTextAsCurrencyDecimal();  // direct
val := txb.getTextAsCurrencyReal();

str := txb.text;                        // indirect
if val.parseCurrencyWithCurrentLocale(str, errOffset) <> 0 then
    // invalid text
endif;
```

Date Round Trips

```
val : Date;
str : String;
txb : TextBox;

//Binary to String
txb.dataType := DataType_ShortDate, DataTypeLongDate;

txb.setTextFromShortDate(val);
txb.setTextFromLongDate(val);

str := val.shortFormat();         // indirect
str := val.longFormat();
txb.text := str;

//String to Binary
val := txb.getTextAsShortDate();
```

```
    val := txb.getTextAsLongDate();

    str := txb.text;                  // indirect
    if not val.parseForCurrentLocale(str) then
        // invalid text
    endif;
```

Time Round Trips

```
    val       : Time;
    str       : String;
    txb       : TextBox;
    errOffset : Integer;

    //Binary to String
    txb.dataType := TextBox.DataType_Time;

    txb.setTextFromTime(val);

    str := val.userFormatAndLcid(null, 0);  // indirect
    txb.text := str;

    //String to Binary
    val := txb.getTextAsTime();             // direct

    str := txb.text;                        // indirect
    if val.parseWithFmtAndLcid(str, null, 0, errOffset) <> 0 then
        // invalid text
    endif;
```

WebHotSpot Class

The **WebHotSpot** class enables you to insert rectangular hotspots directly on to picture controls.

As the **WebHotSpot** class is a subclass of the **Label** class, it inherits all of the properties and methods defined in the **Label**, **Control**, and **Window** classes.

For a summary of the method defined in the **WebHotSpot** class, see "WebHotSpot Method", in the following subsection.

WebHotSpot Method

The method defined in the **WebHotSpot** class is summarized in the following table.

Method	**Description**
canBeChildOf	Specifies whether the control can be placed on the specified form or control

For details, see "Window, Form, and Control Methods", later in this document.

WebHTML Class

The **WebHTML** class enables you to enter free-form text directly into generated HyperText Markup Language (HTML) text.

The HTML code that is dynamically generated conforms to the HTML 3.2 standard and enables you to generate a single user interface that is compatible with multiple browsers.

Note Your HTML is not syntax-checked.

As the **WebHTML** class is a subclass of the **TextBox** class, it inherits all of the properties and methods defined in the **TextBox**, **Control**, and **Window** classes.

For a summary of the properties defined in the **WebHTML** class, see "WebHTML Properties", in the following subsection.

WebHTML Properties

The properties defined in the **WebHTML** class are summarized in the following table.

Property	Description
ignoreHeight	Specifies whether text is resized to fit the height of HTML on a Web page accessed using Internet Explorer 4.0 (or higher)
ignoreWidth	Specifies whether text is resized to fit the width of HTML on a Web page accessed using Internet Explorer 4.0 (or higher)
transparent	Causes the control to be placed above all other sibling controls and the controls or form underneath to be visible

For details, see "Window, Form, and Control Properties", in the following section.

WebInsert Class

The **WebInsert** class enables you to insert the contents of a file as part of the HTML generation. The file contents are copied as is; that is, they are not edited in any way.

As the **WebInsert** class is a subclass of the **Label** class, it inherits all of the properties and methods defined in the **Label**, **Control**, and **Window** classes.

For a summary of the method defined in the **WebInsert** class, see "WebInsert Method", in the following subsection.

WebInsert Method

The method defined in the **WebInsert** class is summarized in the following table.

Method	Description
loadFile	Dynamically loads the specified text file as part of the HTML generate process

For details, see "Window, Form, and Control Methods", later in this document.

WebJavaApplet Class

The **WebJavaApplet** class enables you to insert a Java applet into your Web page as part of the HTML generation.

As the **WebJavaApplet** class is a subclass of the **Label** class, it inherits all of the properties and methods defined in the **Label**, **Control**, and **Window** classes.

For a summary of the properties defined in the **WebJavaApplet** class, see "WebJavaApplet Properties", in the following subsection.

Note If your Java applet is contained in an archive file, set the **hyperlink** attribute to the name of the **.jar** file.

WebJavaApplet Properties

The properties defined in the **WebJavaApplet** class are summarized in the following table.

Property	Description
appletName	Contains the case-sensitive name of the Java applet
code	Contains the compiled code of the applet
codebase	Contains the base Uniform Resource Locator (URL), or code directory, of the applet
horizontalSpace	Contains the number of pixels on each side of the applet
parameters	Contains applet parameters
verticalSpace	Contains the number of pixels above and below the applet

For details, see "Window, Form, and Control Properties", in the following section.

Window, Form, and Control Properties

This section describes the properties defined in the following classes.

- Window
- Form
- Control class and subclasses

acceptTabs

Type: Boolean

Availability: Read or write at any time

The **acceptTabs** property of the **JadeRichText** control specifies whether the Tab key inserts a tab character in the control instead of moving the focus to the next control in the tab order.

By default, a tab character is inserted (that is, the default value is **false**).

accessedCell

Type: JadeTableCell

Availability: Read or write at run time only

The **accessedCell** property of the **Table** class contains a reference to the cell returned by the **accessCell** method of the **Table** class or the **accessCell** method of the **JadeTableSheet** class.

The code fragments in the following examples show the use of the **accessedCell** property.

```
table1.accessCell(2,3).inputType := Table.InputType_TextBox;
table1.accessedCell.foreColor := Red;

table1.accessSheet(2).accessCell(1,4).text := "Company";
table1.accessedCell.alignment := Table.Alignment_Right_Middle;
```

See also the **Table** class **accessedColumn**, **accessedRow**, and **accessedSheet** properties.

accessedColumn

Type: JadeTableColumn

Availability: Read or write at run time only

The **accessedColumn** property of the **Table** class contains a reference to the column returned by the **accessColumn** method of the **Table** class or the **accessColumn** method of the **JadeTableSheet** class.

The code fragment in the following example shows the use of the **accessedColumn** property.

```
table1.accessedColumn.visible := false;
```

See also the **Table** class **accessedCell**, **accessedRow**, and **accessedSheet** properties.

accessedRow

Type: JadeTableRow

Availability: Read or write at run time only

The **accessedRow** property of the **Table** class contains a reference to the row returned by the **accessRow** method of the **Table** class or the **accessRow** method of the **JadeTableSheet** class.

The code fragment in the following example shows the use of the **accessedRow** property.

```
table1.accessedRow.visible := false;
```

See also the **Table** class **accessedCell**, **accessedColumn**, and **accessedSheet** properties.

accessedSheet

Type: JadeTableSheet

Availability: Read or write at run time only

The **accessedSheet** property of the **Table** class contains a reference to the sheet returned by the **accessSheet** method of the **Table** class.

The following example shows the use of the **accessedSheet** property.

```
begin
    table1.accessSheet(1);
    if chkboxFontBold.value = true  then
        table1.accessedSheet.fontBold := true;
    else
        table1.accessedSheet.fontBold := false;
    endif;
end;
```

See also the **Table** class **accessedCell**, **accessedColumn**, and **accessedRow** properties.

accessMode

Type: Integer

Availability: Read or write at run time

The **accessMode** property provides multiple meanings for the **alignment**, **backColor**, **comboIndex**, **foreColor**, **fontName**, **fontSize**, **fontBold**, **fontItalic**, **fontUnderline**, **fontStrikethru**, **inputType**, **maxLength**, **partialTextIndication**, and **wordWrap** properties of **Table** controls.

The **accessMode** property enables table cells to have different colors, fonts, alignments, and input modes. The values of the **accessMode** property are listed in the following table.

Table Class Constant	Value	Contains the …
AccessMode_Sheet	0	Default value for the current sheet. The default value is that initially set for the table at development time or the current sheet for added additional sheets.
AccessMode_Cell	1	Effective value for the current cell of the current sheet.

Table Class Constant	Value	Contains the …
AccessMode_Column	2	Default value for the current column of the current sheet.
AccessMode_Row	3	Default value for the current row of the current sheet.

The font setting for value **AccessMode_Sheet** (**0**) applies to all sheets in the table. Set the **accessMode** property to **AccessMode_Cell** (**1**), **AccessMode_Column** (**2**), or **AccessMode_Row** (**3**), to cause the values to be returned for the current cell, current column, or current row respectively, reflected by the **row**, **column**, or **sheet** property.

Note You can directly access a table element using an instance of the **JadeTableCell**, **JadeTableColumn**, **JadeTableRow**, or **JadeTableSheet** class or by using the appropriate **Table** class **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property without using the appropriate **accessMode** property value.

When the **Table** control obtains the value of one of the properties affected by **accessMode** for a cell, the following rules apply when retrieving the value of that property.

1. If the cell value is specifically set, that value is returned.
2. If the cell value is not set, the value of the column is returned if that is set.
3. If the column value is not set, the value of the row is returned if that is set.
4. If the row value is not set, the default value for the sheet is returned.

The value returned when the **accessMode** property is set to **AccessMode_Cell** (**1**) is the effective value of that property for the current cell.

Cells in the fixed part of a column have the **backColor** and **inputType** properties set only if those properties are specifically set for a cell or the fixed column.

The default value is **AccessMode_Sheet** (**0**); that is, changing any of these properties affects the default values for the current sheet of the table. Each cell of its owner sheet uses these default values.

The code fragment in the following example shows the use of the **accessMode** property.

```
table1.accessMode := Table.AccessMode_Sheet;
table1.alignment  := Table.Alignment_Right;
table1.column     := 1;
table1.accessMode := Table.AccessMode_Column;
table1.alignment  := Table.Alignment_Left;
```

See also the **Table** class **accessedCell**, **accessedColumn**, **accessedRow**, and **accessedSheet** properties.

activation

Type: Integer

The **activation** property of the **OleControl** class controls how or if the application of the object can be activated by its primary verb (usually **edit** or **play**). The values of the **activation** property are listed in the following table.

OleControl Class Constant	Value	Actioned …
Activation_Manual	0	Programmatic initiation only by the **applyVerb** method.

OleControl Class Constant	Value	Actioned …
Activation_SetFocus	1	When the control gains focus.
Activation_DblClick	2	When the control is double-clicked (the default).

activeColor

Type: Integer

Availability: Read or write at any time

The **activeColor** property of the **JadeMask** class contains the color to be used in the mask picture for the definition of the logical area of the control. Any pixel of this color in the mask picture defines a pixel within the control. A pixel of any other color in the mask is considered not to be on the control, and the control will not respond while the mouse is over that pixel.

The default color value for this property is **black** (**0**).

Note The masking ability is disabled if the values of the **rotation** and **stretch** properties are non-zero or the **style** property value is **Style_Mask_Color** (**3**).

alignChildren

Type: Integer

Availability: Read or write at any time

The **alignChildren** property of the **Frame** class and **JadeDockBar** class enables the frame or dock bar control to align child controls placed inside the frame or dock bar. The controls are automatically resized when the frame or dock bar resizes. If the three-dimensional (3D) effects are turned on for a **Frame** control, the children are placed inside the 3D border area.

For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document. See also the docking examples under the **allowDocking** property, later in this document.

Tip If it is difficult positioning docking controls to meet your requirements in the JADE Painter when the **alignChildren** and **alignContainer** properties are set, use the**Suspend Parent Alignments** command from the JADE Painter Layout menu. The **alignContainer** and **alignChildren** properties of controls are then treated as though the property values are zero (**0**) so that no automatic alignment occurs.

When this command is unchecked (the default), the **alignChildren** and **alignContainer** properties of controls behave as normal.

The settings of the **Frame** class **alignChildren** property are listed in the following table.

Frame Class Constant	Value	Description
AlignChildren_None	0	No alignment is performed (the default). Resizes of the frame have no effect.
AlignChildren_Width	1	Aligns width. When the frame is resized, each child control is sized to exactly fit the width of the client area of the frame (the area inside the 3D border).

Frame Class Constant	Value	Description
AlignChildren_All	2	Aligns all. When the frame is resized, any child controls are also resized to exactly fit the height and width of the client area of the frame. If there is more than one child, they are all aligned.

Use this property to place a **Frame** in a form with the **alignContainer** property set to **AlignChildren_All** (**2**).

When a **ListBox** control is placed inside the frame and the **alignChildren** property set to **AlignChildren_All**, the frame and the list box are resized to fit when the form is resized.

By turning off the 3D effects and the frames border, the entire frame can be hidden and yet still make use of the alignment features.

The **integralHeight** property of **TextBox** and **ListBox** controls is ignored if the parent of those controls has the **alignChildren** property set.

Some controls, such as a check box, do not permit their size (usually the height) to change. The recalculation of the positions of the children occurs only if the size or position of the frame is altered. Changing the size and position of a child does not reposition the set of child controls within the frame, unless that child is also a frame control. To force the realignment, change to the size and position of the frame, or set the **alignChildren** property within logic (even if unchanged).

Dock controls that are aligned to their parent by using the **JadeDockBar** class **alignChildren** or **alignContainer** property and that have scroll bars are not scrolled and remain in place in their parent when the scroll bar of the parent is shifted. They therefore remain visible and unchanged when the scroll bar of the parent is adjusted.

The settings of this property for the **JadeDockBar** class are listed in the following table.

JadeDockBar Class Constant	Integer Value
AlignChildren_None	0
AlignChildren_AllHorizontal	1
AlignChildren_AllVertical	2
AlignChildren_Auto	3

For the **JadeDockBar** class, the **AlignChildren_None** (**0**) constant value has the following rules.

- The children within the dock bar remain in the positions allocated by the developer.
- The dock bar size is also unaffected by its children.
- Because the control has fixed positions for its children, docking the control cannot alter the position of the children or the size of the control itself. The current width is therefore retained and the docking could cause the waste of space in its parent.

If the **alignChildren** property is *not* set to **AlignChildren_Auto** and the width of the control is greater that its height and the control is dragged to a left or right docking position (and it is not currently docked left or right), the values of **width** and **height** properties are exchanged when considering the docking position.

Similarly, if the height of the control is greater than its width and the control is dragged to a top or bottom docking position (and it is not currently docked top or bottom), the values of **width** and **height** properties are exchanged when considering the docking position. This is necessary because the control does not know how to resize itself. For example, if the control were docked at the top and then dragged to the left position and these exchanges were not done, the control would retain its current width, stretch its height, and therefore probably use all of the space in its parent.

For the **JadeDockBar** class, the **AlignChildren_AllHorizontal** (**1**) constant value has the following rules.

- If there is only one child, the child is positioned at the left and top of that area with a width and height equal to the remaining width and height of that area.
- If there is more than one child, the children are positioned at the left of the remaining area with a width equal to the width of the remaining area. The child with the lowest top value is positioned at the top of the area, immediately followed by the next top-most child, and so on.

  The height of each child is the minimum of the gap between the initial top position and the top position of the next child, and the initial height of the child itself.
- If the combined height of the children exceeds the total available height, each child is equally reduced in height so that exactly the entire available area of the parent is used. If the combined height of the children is less than the total available height, each child is equally increased in height so that exactly the entire available area of the parent is used.
- When the children are aligned so that they share all of the remaining area of the parent, they retain their relative heights if the parent is subsequently resized. If the user resizes the form, no resize logic is therefore required when the parent height is affected by the height of the form.

For the **JadeDockBar** class, the **AlignChildren_AllVertical** (**2**) constant value has the following rules.

- If there is one child only, that child is positioned at the left and top of that area with a width and height equal to the remaining width and height of that area.
- If there is more than one child, the children are positioned at the top of the remaining area with a height equal to the height of the remaining area. The child with the lowest left value is positioned at the left of the area, immediately followed by the next left-most child, and so on.

  The width of each child is the minimum of the gap between the initial left position and the left position of the next child, and the initial width of the child itself.
- If the combined width of the children exceeds the total available width, each child is equally reduced in width so that exactly the entire available area of the parent is used. If the combined width of the children is less than the total available width, each child is equally increased in width so that exactly the entire available area of the parent is used.
- When the children have been aligned so that they share all of the remaining area of the parent, they retain their relative widths if the parent is subsequently resized. If the user resizes the form, no resize logic is therefore required when the parent width is affected by the width of the form.

For the **JadeDockBar** class, the **AlignChildren_Auto** (**3**) constant value has the following rules.

- The dock bar automatically positions the child controls in the order of the **tabIndex** property.
- The dock bar automatically resizes itself to the size required for its children.
- If the dock bar is aligned vertically, the controls are positioned in columns.
- If the dock bar is aligned horizontally, the controls are positioned horizontally in rows.
- Additional columns or rows result if the position of the controls would cause the dock bar to exceed the size of the area available within its parent.
- The spacing between the controls is determined by the value of the **JadeDockBar** class **autoSpacingX** and **autoSpacingY** properties.
- The spacing of a control from the edges of the dock bar area is half the value of the **autoSpacingX** property horizontally and half the value of the **autoSpacingY** property vertically.

alignContainer

Type: Integer

Availability: Read or write at any time

The **alignContainer** property of the **Frame** class and **JadeDockBase** class enables the control to align itself to its parent container so that the control automatically resizes with the container. For example, if the container of a frame is the form, any form resize causes the frame to also resize automatically to fit the new form size.

If the frame has children, those children could also be resized automatically (another frame with the **alignContainer** property set or this frame with the **alignChildren** property set).

This property is handled the same way for **Frame** and **JadeDockBar** controls except that the handling is defined by the parent of the **JadeDockBar** control rather than by the control itself.

When aligning controls, you should be aware of the following functionality.

- Sibling controls that have the **alignContainer** property set do not occupy the same area on their parent. If so, use the **visible** property to control the window that is currently displayed.
- Controls that are aligned to their parent by using the **alignContainer** or **alignChildren** property and that have scroll bars are not scrolled and remain in place in their parent when the scroll bar of the parent is shifted. Dock controls and **StatusLine** controls, for example, therefore remain visible and unchanged when the scroll bar of the parent is adjusted.
- The **relativeTop**, **relativeLeft**, **relativeWidth**, and **relativeHeight** properties are ignored when a control is aligned.
- The **integralHeight** property of **TextBox** and **ListBox** controls is ignored if the parent of those controls has the **alignChildren** property set.

Tip If it is difficult positioning docking controls to meet your requirements in the JADE Painter when the **alignContainer** and **alignChildren** properties are set, use the**Suspend Parent Alignments** command from the JADE Painter Layout menu. The **alignContainer** and **alignChildren** properties of controls are then treated as though the property values are zero (**0**) so that no automatic alignment occurs.

When this command is unchecked (the default), the **alignContainer** and **alignChildren** properties of controls behave as normal.

For the **Frame** class, any non-zero alignment setting causes the frame to align to the width of its parent container. The setting then determines the vertical size and position of the frame.

The **alignContainer** property values for **Frame** controls are listed in the following table.

Frame Class Constant	Value	Description
AlignContainer_All	5	Aligns container. The frame aligns itself to its entire container size.
AlignContainer_AllHorizontally	5	Children share the entire remaining client area of the parent horizontally. (For details, see the following table of values for the **JadeDockBase** class **alignContainer** property.)
AlignContainer_AllVertically	6	Children share the entire remaining client area of the parent vertically. (For details, see the following table of values for the **JadeDockBase** class **alignContainer** property.)

Frame Class Constant	Value	Description
AlignContainer_Bottom	2	The frame retains its current height and aligns the bottom of the frame to the bottom of its container.
AlignContainer_None	0	No alignment is performed (the default). Resizes of the frame container have no effect.
AlignContainer_Stretch	4	Stretches to next control below. The frame retains its current top position and stretches downwards until it encounters another control or the bottom of its container.
AlignContainer_Top	1	The frame retains its current height and aligns the top of the frame to the top of its container.
AlignContainer_Width	3	Aligns width. The frame retains its current height and top position.

For the **Frame** control, the **AlignContainer_All** (**5**) constant value for the **alignContainer** property is treated as the **AlignContainer_AllHorizontal** (**5**) value. Use this property to place a frame in a form with the **alignContainer** property set to **AlignContainer_All** (**5**) or **AlignContainer_AllHorizontal** (**5**).

When a **ListBox** control is placed inside the frame and the **alignChildren** property is also set to **AlignChildren_All** (**2**), the frame and the list box are resized to fit when the form is resized. By turning off the three-dimensional (3D) effects and the frames border, the entire frame can be hidden and yet still make use of the alignment features.

Note For the **AlignContainer_Stretch** (**4**) value, vertically adjacent frames and status line controls merge their borders if they both have the **Window** class **borderStyle** property set.

Additionally, if adjacent frames both have the **borderStyle** and **bevelOuter** properties set to **BorderStyle_None** (**0**) and **Bevel_None** (**0**) and the same **boundaryBrush** and **boundaryColor** property settings, the frames or status line controls merge their boundary areas.

For the **JadeDockBase** class, the **alignContainer** property values are listed in the following table.

Class Constant	Value	Description
AlignContainer_AllHorizontally	5	For this value of the **alignContainer** property, the children share the entire remaining client area of the parent horizontally. If there is one child only, the child is positioned at the left and top of that area with a width and height equal to the remaining width and height of that area. If there is more than one child, the children are positioned at the left of the remaining area with a width equal to the width of the remaining area. The child with the lowest top value is positioned at the top of the area, immediately followed by the next top-most child, and so on. The height of each child is the minimum of the gap between the initial top position and the top position of the next child, and the initial height of the child itself. If the combined height of the children exceeds the total available height, each child is equally reduced in height so that exactly the entire available area of the parent is used. If the combined height of the children is less than the total available height, each child is equally increased in height so that exactly the entire available area of the parent is used.

Class Constant	Value	Description
		When the children have been aligned so that they totally share the remaining area of the parent, they retain their relative heights if the parent is subsequently resized. If the user resizes the form, no resize logic is therefore required when the parent height is affected by the height of the form.
AlignContainer_AllVertically	6	For this value of the **alignContainer** property, the children share the entire remaining client area of the parent vertically. If there is one child only, the child is positioned at the left and top of that area with a width and height equal to the remaining width and height of that area. If there is more than one child, the children are positioned at the top of the remaining area with a height equal to the height of the remaining area. The child with the lowest left value is positioned at the left of the area, immediately followed by the next left-most child, and so on. The width of each child is the minimum of the gap between the initial left position and the left position of the next child, and the initial width of the child itself. If the combined width of the children exceeds the total available width, each child is equally reduced in width so that exactly the entire available area of the parent is used. If the combined width of the children is less than the total width available, each child is equally increased in width so that exactly the entire available area of the parent is used. When the children have been aligned so that they share all of the remaining area of the parent, they retain their relative widths if the parent is subsequently resized. If the user resizes the form, no resize logic is therefore required when the parent width is affected by the width of the form.
AlignContainer_Bottom	2	Positioned at the left and bottom of the remaining client area of the parent of the control, with a width equal to the width of the remaining area. The control retains its height unless its children can be automatically positioned and resized.
AlignContainer_Left	3	Positioned at the left and top of the remaining client area of the parent of the control, with a height equal to the height of the remaining area. The control retains its width unless its children can be automatically positioned and resized.
AlignContainer_None	0	There is no alignment to the parent of the control.
AlignContainer_Right	4	Positioned at the right and top of the remaining client area of the parent of the control, with a height equal to the height of the remaining area. The control retains its width unless its children can be automatically positioned and resized.
AlignContainer_Top	1	Positioned at the left and top of the remaining client area of the parent of the control, with a width equal to the width of the remaining area. The control retains its height unless its children can be automatically positioned and resized.

You can dock a **JadeDockBar** or **JadeDockContainer** control into any docking position (with one exception), regardless of the value of its **alignContainer** property. The docking control then adopts the new **alignContainer** property value implied by the docking mode. The exception is that a **JadeDockBar** control that has its **alignContainer** property set to **AlignContainer_None** cannot be docked **AlignContainer_AllHorizontal** or **AlignContainer_AllVertical**.

A **JadeDockBar** control whose **alignContainer** property is set to **AlignContainer_None** is mostly useful as a toolbar sitting on a **JadeDockContainer** control as its parent and when docked somewhere else, it remains at its current size or the size derived from its children if the value of the **alignChildren** property is **AlignChildren_Auto**. Even when dragged and docked left, right, top, or bottom, it requires a **JadeDockContainer** control as its parent. If it is not docked into a **JadeDockContainer** control, it clones its original **JadeDockContainer** parent or creates a new **JadeDockContainer** control if it did not have one.

Setting the **alignContainer** property to **AlignContainer_AllHorizontal** causes all other siblings with the property set to **AlignContainer_AllHorizontal** to also be set to **AlignContainer_AllVertical**.

Similarly, setting the **alignContainer** property to **AlignContainer_AllVertical** causes all other siblings with the **alignContainer** property set to **AlignContainer_AllHorizontal** to also be set to **AlignContainer_AllVertical**. In addition, a **Frame** control could be aligned to the top, bottom, width, or stretched to the next control or to its entire parent container area. If multiple controls were added to the same parent with the same alignment type, they occupy the same positions within the parent.

You can have a number of different controls all aligned in the same or in different ways within the parent, to allow the dock bar containers to co-exist with each other, with **StatusLine** controls, and with aligned **Frame** controls.

In addition, controls with the same alignment type (and therefore the same alignment priority) are ordered by their position. For example, the order of two controls aligned to the top of the parent is based on the current top positions of the controls. If the top positions are the same, the control that is placed above the other is undefined. Similarly, controls aligned to the bottom are ordered in reverse order of their top positions, left-aligned controls by their left positions, and right-aligned controls are ordered in reverse order of their left position.

To enable multiple controls to be aligned in different ways within the same parent, the values of the **alignContainer** property have a priority.

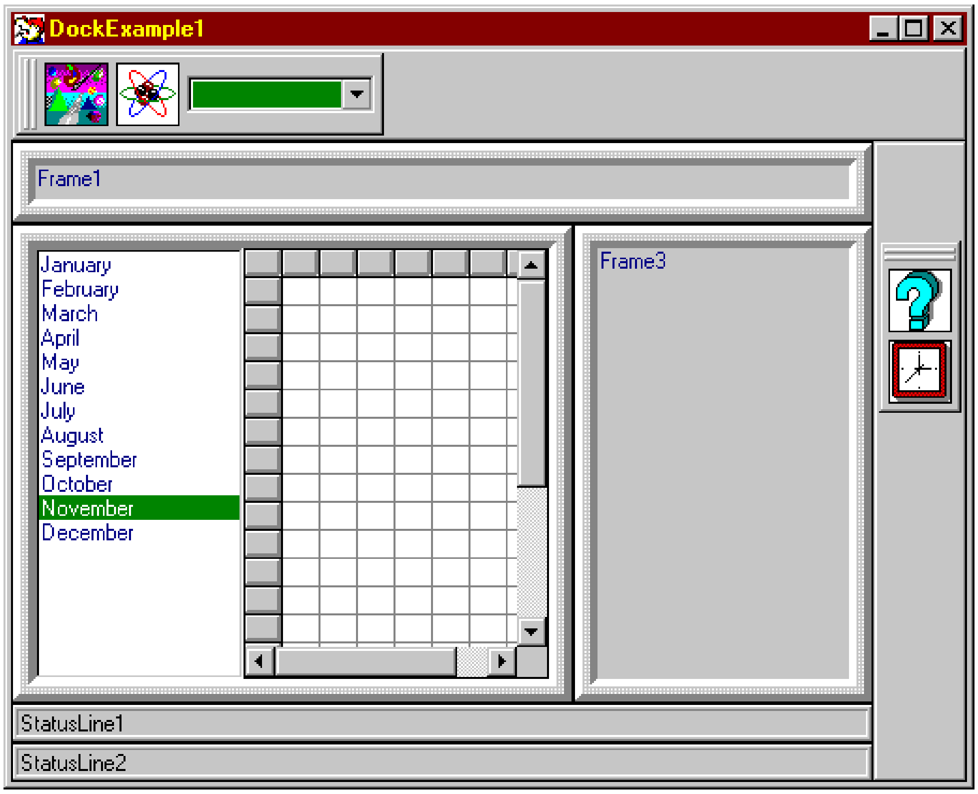
Each control is aligned in the following priority order so that the control is positioned using the remaining space after each previous alignment within that parent.

1. Dock control top-aligned
2. Dock control bottom-aligned
3. Dock control left-aligned
4. Dock control right-aligned
5. Frame control top-aligned
6. Frame or status line bottom-aligned
7. Frame width-aligned
8. Frame stretched to the next control
9. Frame or dock control aligned-all horizontally
10. Frame or dock control aligned-all vertically

Controls with such alignments are automatically re-evaluated and resized, when required, under the following conditions.

- A control is moved
- A control is resized
- A control is added to the form
- A control is deleted from the form
- The **visible** property of the control is changed
- The **alignContainer** property of the control is changed
- The **alignChildren** property of the control is changed
- A docking control is floated
- A docking control is docked
- The parent of a control is changed

The following image is an example of a form that includes docking container controls.



In this image:

- A docking container is aligned to the top (with a dock bar inside, holding pictures and a combo box)
- A docking container is aligned on the right (with a dock bar inside, holding pictures)

- **Frame1** is aligned to the top
- Two status lines are aligned to the bottom
- **Frame2** is aligned all, containing a list box and a table, and the **Frame** control has the **alignChildren** property set to **AlignChildren_All** (**2**)
- **Frame3** is aligned all

If the user resizes the form in this example, all of the aligned containers are readjusted without JADE logic being required. In addition, the **ListBox** and **Table** controls are also resized, because **Frame2** uses the value **AlignChildren_All** (**2**) of the **alignChildren** property.

Frame2 and **Frame3** demonstrate the *align 'all'* concept so that they are stretched vertically and they share the remaining space horizontally in the form. The left position of **Frame3** determines how much space the **Frame2** control is given. Similarly, **Frame2** has the **alignChildren** property set to **AlignChildren_All** (**2**), which aligns the list box and table within the **Frame2** area with the left position of the table, determining how much width the list box is assigned.

Notes Frames aligned to the width and stretched to the next control (that is, those with priority 7, in which the frame width is aligned, and priority 8, in which the frame is stretched to the next control) do not adjust the remaining space used for alignment and therefore should not be mixed with align 'all' types (that is, those with priority 9, in which the frame or dock control are aligned-all horizontally, and priority 10, in which the frame or dock control are aligned-all vertically).

The value of the **alignContainer** property of a control can be defined by using its own **alignContainer** property value and the **alignChildren** property of its parent. If the **alignContainer** property of a control is set, that setting takes precedence over the value of the **alignChildren** property of its parent.

If a container contains controls that are docked both 'all' vertically and horizontally, they are aligned horizontally.

For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document. See also the docking examples under the **allowDocking** property. For details about adding resize bars to docking controls that are all aligned horizontally or vertically, see the **showResizeBar** property.

alignment

Type: Integer

Availability: Read or write at any time

The **alignment** property determines whether the text is placed before or after the button bitmap image for **CheckBox** and **OptionButton** classes. For other control classes, it sets the alignment of the text in a control.

For a check box or option button control, the **alignment** property settings are listed in the following table.

Class Constant	Value	Description
Alignment_Left	0	Control aligns to the left with text on the right (the default)
Alignment_Right	1	Control aligns to the right with text on the left

For the **Table** class, the **alignment** property settings, which affect the text and picture components of table cells, are listed in the following table. (In this table, the constants with two integer values in the range **1** through **3** are compatible and are the values for vertical or horizontal alignment in each case.)

Table Class Constant	Value	Description
Alignment_Default	0	Uses default alignment (left for a sheet, or the alignment value of the cell)
Alignment_Left	1	Text aligns to the left of the cell after any cell picture
Alignment_Left_Middle	1	Text is left-justified in the vertical middle of the cell
Alignment_Right	2	Text aligns to the right with any cell picture on its left
Alignment_Right_Middle	2	Text is right-justified in the vertical middle of the cell
Alignment_Center	3	Text and any picture are centered within the cell
Alignment_Center_Middle	3	Text is centered horizontally at the middle of the cell
Alignment_Left_Top	4	Text is left-justified at the top of the cell
Alignment_Right_Top	5	Text is right-justified at the top of the cell
Alignment_Center_Top	6	Text is centered horizontally at the top of the cell
Alignment_Left_Bottom	7	Text is left-justified at the bottom of the cell
Alignment_Right_Bottom	8	Text is right-justified at the bottom of the cell
Alignment_Center_Bottom	9	Text is centered horizontally at the bottom of the cell

The **Table** control **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the alignment for the current sheet, cell, column, or row is being accessed. If the cell is too small to fit both the text and the picture, the text takes precedence over the picture unless the **stretch** property is set to **Stretch_None_Picture_First** (**2**) or **Stretch_Cell_Picture_First** (**3**), where the picture takes precedence.

The size of the text is determined by taking the cell size and calculating the text size required using the word wrap option. The space left over is used to scale the image proportionally so that the whole image is displayed. (See also the **partialTextIndication** and **stretch** properties.)

For the **TextBox** class, the **alignment** property settings are listed in the following table.

TextBox Class Constant	Value	Description
Alignment_Left	0	Text aligns to the left (the default).
Alignment_Right	1	Text aligns to the right (setting this value resets the **scrollBars** and **scrollHorizontal** properties to exclude horizontal scrolling)
Alignment_Center	2	Text aligns to the center (setting this value resets the **scrollBars** and **scrollHorizontal** properties to exclude horizontal scrolling)

Note If the value of the alignment property of a text box is **Alignment_Right** or **Alignment_Center**, setting the value of the **scrollHorizontal** property to **true** has no effect.

For the **Frame**, **JadeMask**, **Label**, or **StatusLine** class, the **alignment** property settings are listed in the following table.

Class Constant	Value	Description
Alignment_Left_Top	0	Left justify, top of the control (the default)
Alignment_Left_Middle	1	Left justify, middle (vertically) of the control
Alignment_Left_Bottom	2	Left justify, bottom of the control
Alignment_Right_Top	3	Right justify, top of the control
Alignment_Right_Middle	4	Right justify, middle (vertically) of the control
Alignment_Right_Bottom	5	Right justify, bottom of the control
Alignment_Center_Top	6	Center horizontally at the top of the control
Alignment_Center_Middle	7	Center horizontally at the middle (vertically) of the control
Alignment_Center_Bottom	8	Center horizontally at the bottom of the control

For **JadeMask** controls, the alignment occurs within the rectangle defined by the **captionLeft**, **captionTop**, **captionHeight**, and **captionWidth** properties. If the value of the **captionWidth** property is zero (**0**), the width of the caption region is the value of the **clientWidth** property less the value of the **captionWidth** property. If the value of the **captionHeight** property is zero (**0**), the height is the value of the **clientHeight** property less the value of the **captionTop** property.

For frame and status line controls, the text is always displayed inside the area bounded by the 3D borders.

The code fragment in the following example shows the use of the **alignment** property.

```
    tblPortfolio.alignment := Table.Alignment_Right;
```

For the **JadeRichText** control, the **alignment** property contains the alignment of the current paragraph. (For an example of the use of this property, see "JadeRichText Control Method Example", earlier in this document.)

The **alignment** property values for the **JadeRichText** control are listed in the following table.

JadeRichText Class Constant	Value	Paragraph…
Alignment_Center	2	Is centered in the control
Alignment_Justify	3	Is aligned relative to the left and right margins
Alignment_Left	0	Aligns to the left with text on the right (the default)
Alignment_Right	1	Aligns to the right with text on the left
ParagraphFormat_Undefined	#80000000	Alignment cannot be assigned but is returned to indicate that the selected text contains multiple paragraphs with different alignment values

Although you can define values other than those listed in this table, if you do so, the paragraph is displayed as left-aligned.

allowClose

Type: Boolean

Availability: Read or write at run time only

The **allowClose** property of the **Form** class enables access to the status of the Control-Menu **Close** command. By default, a form can be closed by selecting the **Close** command from the Control Menu. To disable the ability to close a form by using the **Close** command, set the **allowClose** property to **false**. The user then cannot specifically close that form by using the Control-Menu, and the form can only be closed programmatically by using the **unloadForm** method.

If the form is an MDI child, closing the MDI frame of the form also closes the child form.

Setting the **allowClose** property to **false** does not affect the calling of the **unloadForm** method when the form is unloaded.

If the **controlBox** property of the form is set to **false**, the **allowClose** property is always **false**. If the **controlBox** property is changed from **false** to **true**, the **allowClose** property is reset to **true**.

allControlChildren

Type: ControlArray

Availability: Read at run time only

The **allControlChildren** property of the **Window** class contains a reference to an array of all of the controls that are contained in the window (form or control), including children of children. The collection is in no particular order except that children come after parents.

The array is changed if the z-order of a control is changed by logic or if the parent of a control is changed.

Applies to Version: 2016.0.01 and higher

allMenuItems

Type: MenuItemArray

Availability: Read at run time only

The **allMenuItems** property of the **Form** class contains a reference to an array of all of the menu items on the form. The collection is ordered according to the defined menu item list. This is the equivalent of using the **Form** class **menuItems** method.

Applies to Version: 2016.0.01 and higher

allowDocking

Type: Integer

Availability: Read or write at any time

The **allowDocking** property of the **Form**, **Frame**, and **JadeDockContainer** classes controls the type of docking allowed by the window.

The value of the **allowDocking** property is a bit mask of the allowable docking types that are accepted by the window, and it is used when the user drags a **JadeDockBar** or **JadeDockContainer** control into a potential docking position within the **Form**, **Frame**, or **JadeDockContainer** control.

Docking is automatically rejected if the appropriate bit is not set within the value of the **allowDocking** property.

The **allowDocking** property allows any combinations of the **Window** class constants listed in the following table.

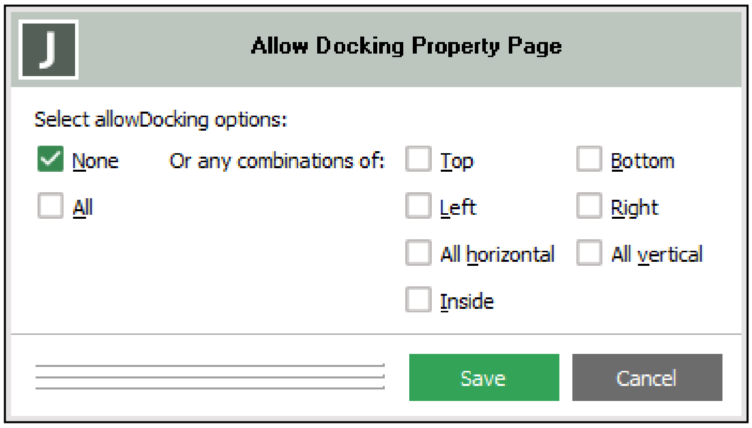
Window Class Constant	Value	Description
AllowDocking_None	0	No docking allowed
AllowDocking_Top	#1	Allow top docking
AllowDocking_Bottom	#2	Allow bottom docking
AllowDocking_Left	#4	Allow left docking
AllowDocking_Right	#8	Allow right docking
AllowDocking_AllHorizontal	#10	Allow all horizontal docking
AllowDocking_AllVertical	#20	Allow all vertical docking
AllowDocking_Inside	#40	Allow docking inside
AllowDocking_All	#7f	All of the above
AllowDocking_AnyEdge	#f	Allow top, bottom, left, and right docking

Note The **AllowDocking_Inside** value, which means that the control is docked without any special alignment applying, is intended for use with a **JadeDockContainer** that has multiple **JadeDockBar** controls as children (for example, a toolbar that has several **JadeDockBar** controls, each holding a logical grouping of controls).

To simplify the setting of the allowDocking property during painting

1. Select a **Form**, **Frame**, or **JadeDockContainer** control on a form in the JADE Painter.
2. Click the **allowDocking** property on the **Specific** sheet of the Properties dialog.
3. Select the **Change** value from the drop-down list in the right column of the **allowDocking** property row.

   The Allow Docking Property Page dialog, shown in the following image, is then displayed.



Allow Docking Property Page

Select allowDocking options:

[✓] None   Or any combinations of: [ ] Top   [ ] Bottom
[ ] All   [ ] Left   [ ] Right
[ ] All horizontal   [ ] All vertical
[ ] Inside

4. To specify the docking that you require for the control, clear all docking options by checking the **None** check box, set all docking options by checking the **All** check box, or specify individual options by checking or unchecking the values that you require or do not want to apply, respectively.

Although the **allowDocking** property indicates that a docking type is permitted, the **queryDock** event method is still called for the **JadeDockBar** or **JadeDockContainer** control. The **queryDock** event method passes the window into which the control will be docked and the type of docking as parameters and returns whether that type of docking is permitted (that is, **true** or **false**).

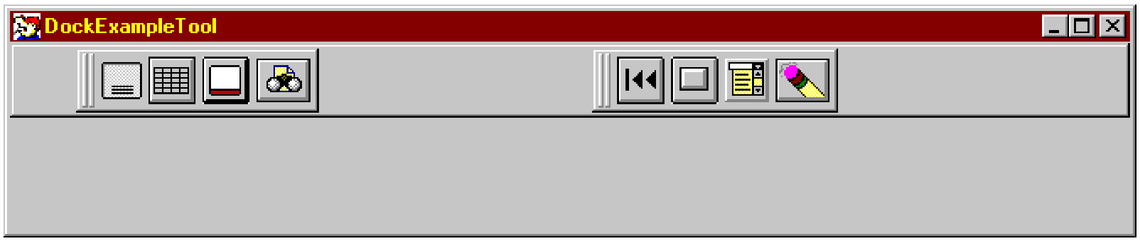
The default value of the **allowDocking** property for **Form** and **Frame** classes is **AllowDocking_None** (that is, only the **None** check box is checked when you access the Allow Docking Property Page dialog and a **Form** or **Frame** control has focus in the JADE Painter).

The default value for **JadeDockContainer** controls is **AllowDocking_Inside** (that is, only the **Inside** check box is checked when you access the dialog when a docking container has focus in the Painter).

For examples, see "Multiple Group Toolbar Example", "Multiple Group Toolbar on a Non-MDI Form Example", and "Using Align All with Multiple Panes", in the following subsections. For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document.

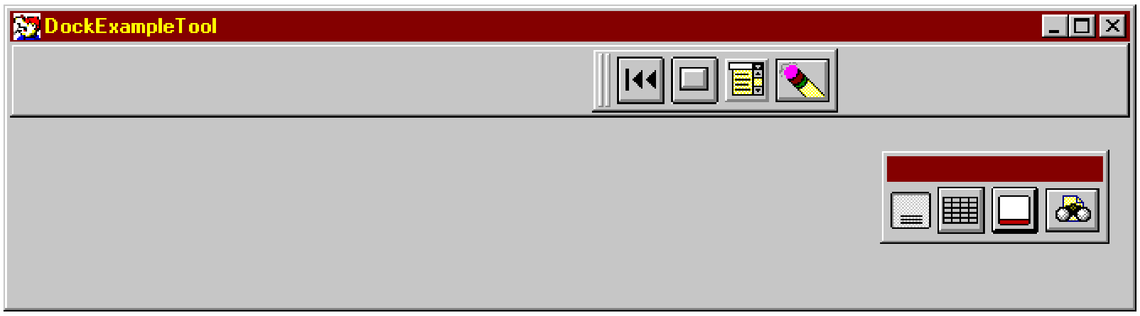
Multiple Group Toolbar Example

The following image is an example of a form that has a **JadeDockContainer** control aligned at the top and that contains two **JadeDockBar** controls. As the form is an MDI frame, there is no need to consider the situations in which the dock bars are floated.

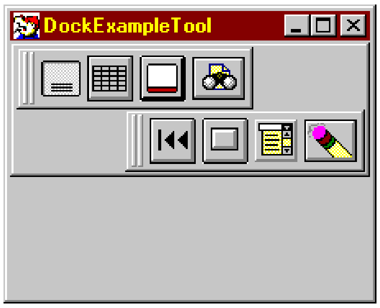


Note The dock bars can be positioned anywhere on the **JadeDockContainer** control, provided that they do not overlap and that only one row is required for the **JadeDockBar** class.

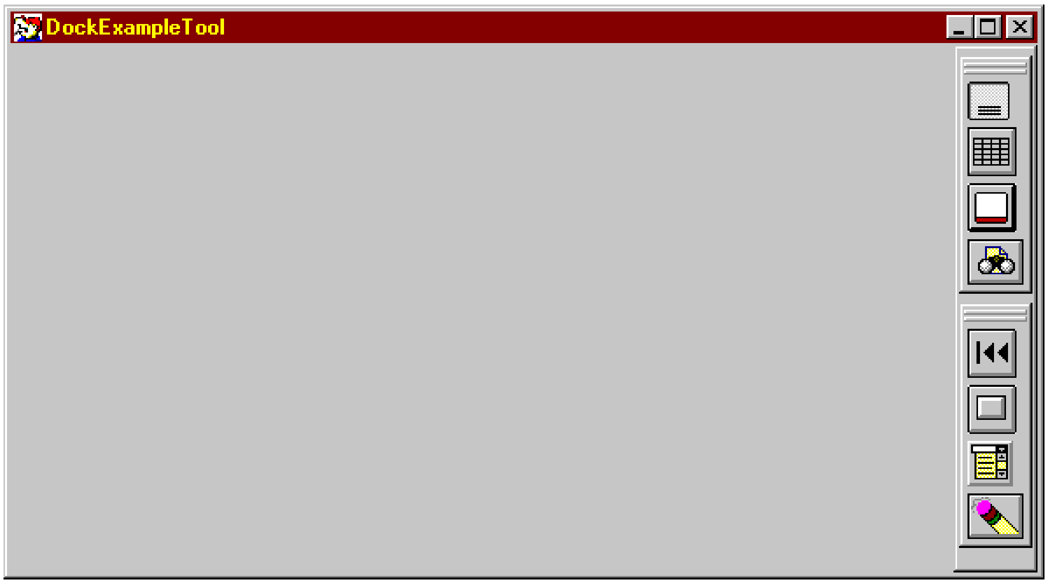
The following image is an example of the result if one dock bar is floated.



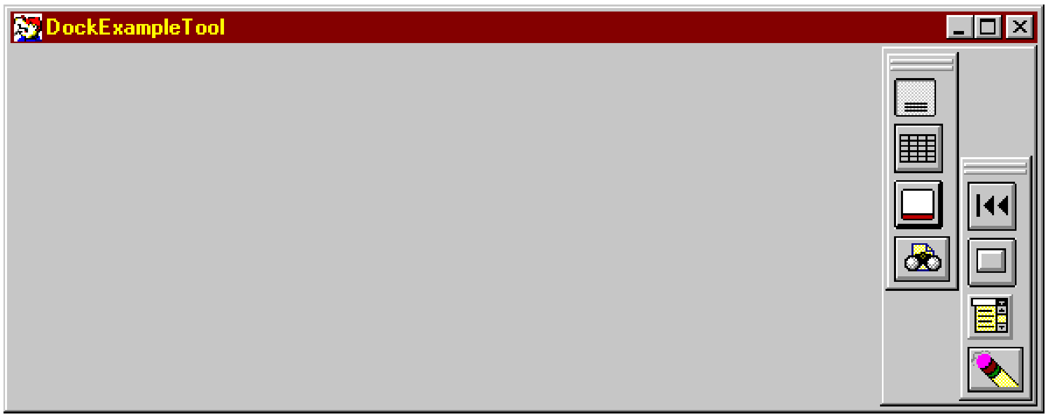
The following image is an example of the result if the width of the form is reduced to the point where the **JadeDockContainer** control would require more than one row.



The following image is an example of the result if the **JadeDockContainer** control is aligned to the right (that is, the **JadeDockBase** class **alignContainer** property is set to **AlignContainer_Right**).

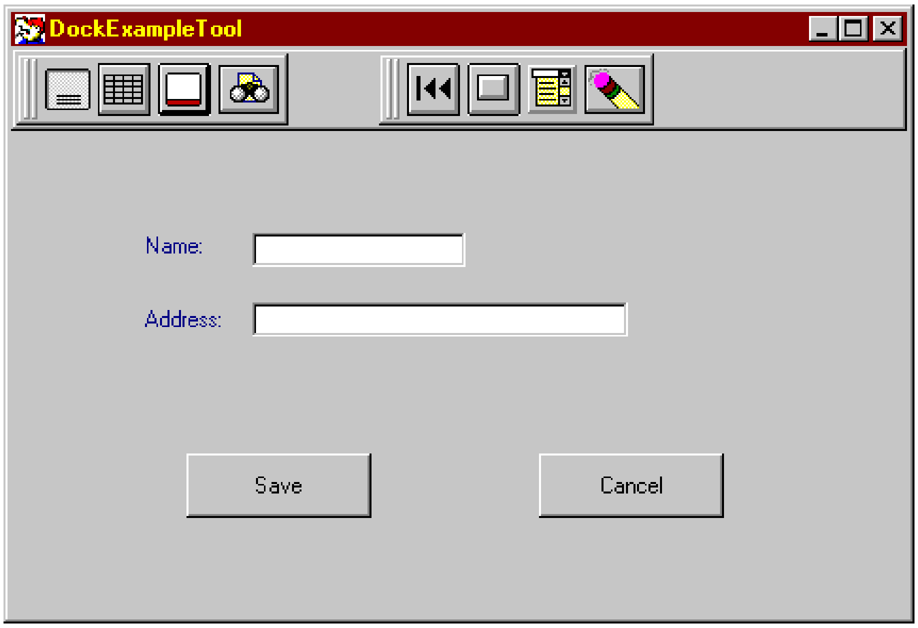


The following image is an example of shrinking the height of the form so that two columns are required.



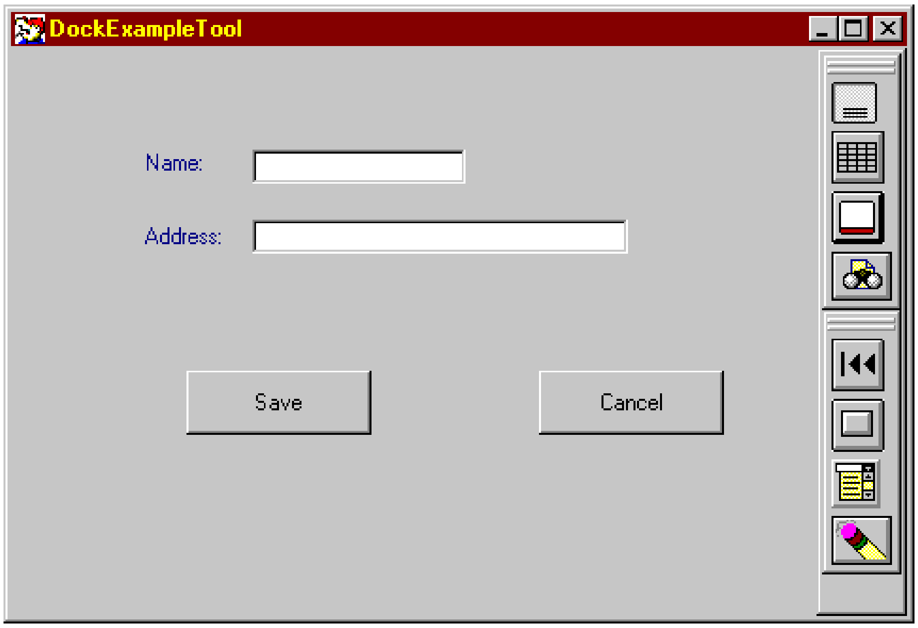
Multiple Group Toolbar on a Non-MDI Form Example

The following image is an example of a form with a **JadeDockContainer** control aligned at the top and containing two **JadeDockBar** controls.



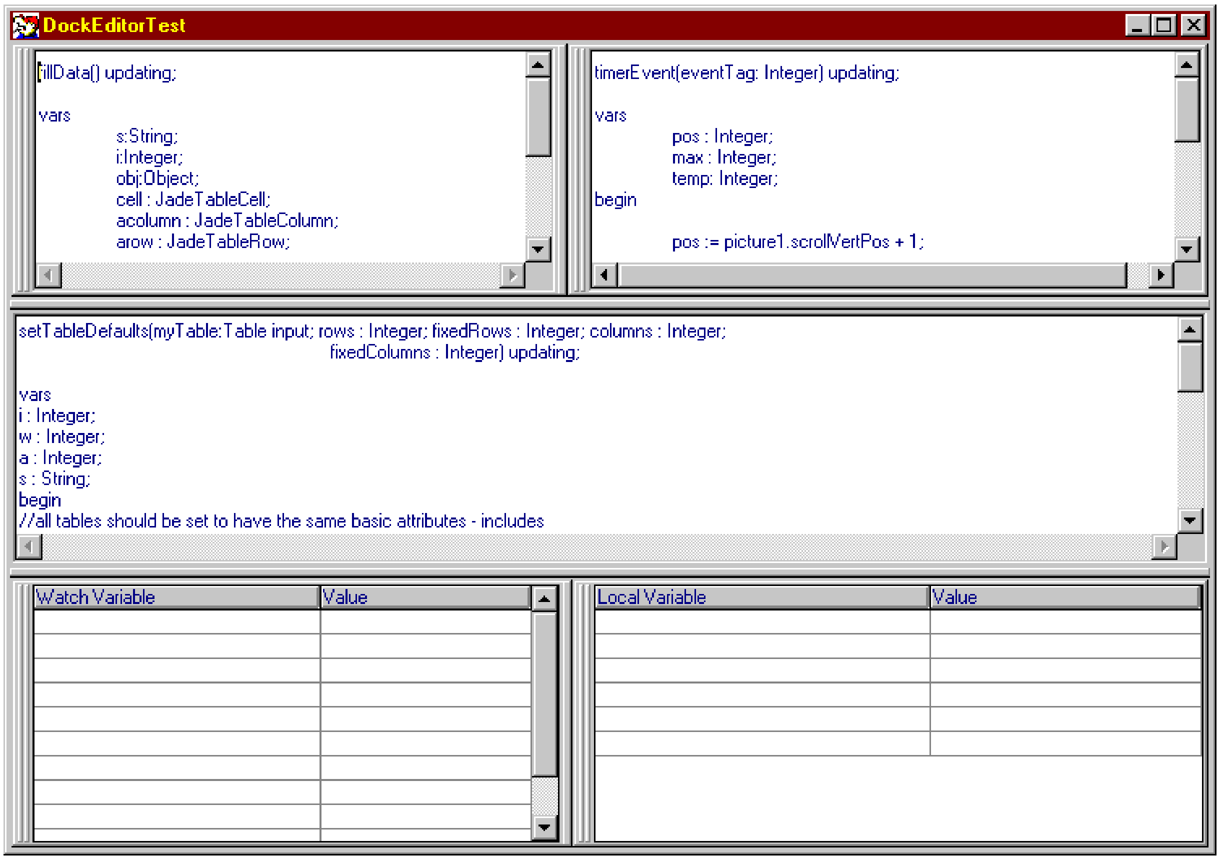
The form in this example contains other controls that are placed on a **Frame** control with the **alignContainer** property set to **AlignContainer_AllHorizontal** (**5**).

In the following image, the **JadeDockContainer** control is docked on the right.



Using Align All with Multiple Panes

The following image is an example of a form with multiple panes and which allows the use of the **AlignContainer_AllVertical** (**6**) and **AlignContainer_AllHorizontal** (**5**) values of the **alignContainer** property and the **AlignChildren_AllVertical** (**2**) value of the **alignChildren** property.



```
fillData() updating;

vars
        s:String;
        i:Integer;
        obj:Object;
        cell : JadeTableCell;
        acolumn : JadeTableColumn;
        arow : JadeTableRow;
```

```
timerEvent(eventTag: Integer) updating;

vars
        pos : Integer;
        max : Integer;
        temp: Integer;
begin

        pos := picture1.scrollVertPos + 1;
```

```
setTableDefaults(myTable:Table input; rows : Integer; fixedRows : Integer; columns : Integer;
                        fixedColumns : Integer) updating;

vars
i : Integer;
w : Integer;
a : Integer;
s : String;
begin
//all tables should be set to have the same basic attributes - includes
```

Watch Variable	Value

Local Variable	Value

The form in the previous example has three layers, as follows.

1. Three **JadeDockContainer** controls with the **alignContainer** property set to **AlignContainer_AllHorizontal** and the form as parent.

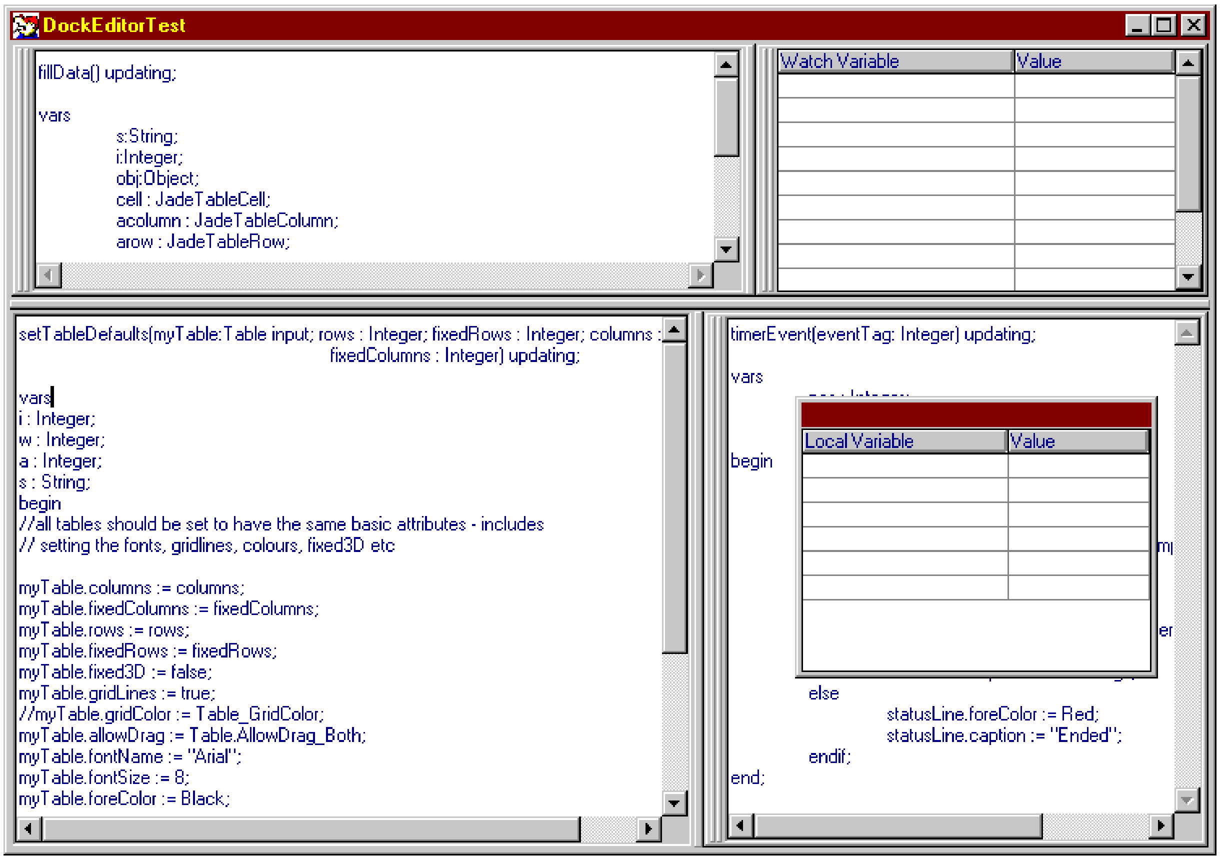
   The **JadeDockContainer** controls have no border drawn except two that have resize bars drawn across the form, allowing the user to resize these base panes.

2. **JadeDockBar** controls in each **JadeDockContainer** control with the value of the **alignContainer** property set to **AlignContainer_AllVertical**. Two of the **JadeDockBar** controls are in the top and bottom **JadeDockContainer** control panes and the other is in the middle pane. Each of the **JadeDockBar** controls has the **alignChildren** property set to **AlignChildren_AllVertical**.

   Note the resize bar that is drawn vertically in the top and bottom panes between the two **JadeDockBar** controls.

3. Each **JadeDockBar** control then has a **TextBox** or a **Table** control as a child that has the **alignChildren** property set to **AlignChildren_AllVertical**.

The form in the following image allows any of the **JadeDockBar** control panes to be floated, docked into another position, resized, or created as a new horizontal pane on the form if the **allowDocking** property of the form permits it.



```
fillData() updating;

vars
        s:String;
        i:Integer;
        obj:Object;
        cell : JadeTableCell;
        acolumn : JadeTableColumn;
        arow : JadeTableRow;
```

Watch Variable	Value

```
setTableDefaults(myTable:Table input; rows : Integer; fixedRows : Integer; columns :
                                        fixedColumns : Integer) updating;

vars
i : Integer;
w : Integer;
a : Integer;
s : String;
begin
//all tables should be set to have the same basic attributes - includes
// setting the fonts, gridlines, colours, fixed3D etc

myTable.columns := columns;
myTable.fixedColumns := fixedColumns;
myTable.rows := rows;
myTable.fixedRows := fixedRows;
myTable.fixed3D := false;
myTable.gridLines := true;
//myTable.gridColor := Table_GridColor;
myTable.allowDrag := Table.AllowDrag_Both;
myTable.fontName := "Arial";
myTable.fontSize := 8;
myTable.foreColor := Black;
```

```
timerEvent(eventTag: Integer) updating;

vars

begin

        else
                statusLine.foreColor := Red;
                statusLine.caption := "Ended";
        endif;
end;
```

Local Variable	Value

allowDrag

Type: Integer

Availability: Read or write at any time

The **allowDrag** property of the **Table** class enables rows and columns of a table to be dragged to new positions within the table.

The **Table** class constants that represent the **allowDrag** property values are listed in the following table.

Table Class Constant	Integer Value	Description
AllowDrag_None	0	Neither columns nor rows can be dragged (the default)
AllowDrag_Rows	1	Rows can be dragged
AllowDrag_Columns	2	Columns can be dragged
AllowDrag_Both	3	Rows and columns can both be dragged

Rows and columns can be dragged only if they have fixed rows and fixed columns, respectively.

To drag a row or column, the user:

1. Clicks on the heading of the fixed column or row that is to be moved.
2. Drags the selected heading to the new position. The column or row is then inserted in front of the column or row to which it was dragged.
3. Releases the mouse.

The code fragment in the following example shows the use of the **allowDrag** property.

```
    table1.allowDrag := Table.AllowDrag_Both;
```

The **Table** class **queryRowMove** or **queryColumnMove** event method is then called. If neither of these event methods is defined, the move is performed. If these event methods are defined, the move is performed only if a value of **true** is returned by the events. The move is aborted if a value of **false** is returned.

Note By default, clicking on a fixed row or column also causes all non-fixed cells of that row or column to be selected. To avoid this, set the **selectMode** property to the appropriate value.

allowInPlace

Type: Boolean

The **allowInPlace** property of the **OleControl** class specifies whether in-place editing can occur.

This property is ignored if any of the following conditions applies.

- The OLE Server is not in-place-capable
- The object is linked
- There is no room for the server to place its toolbar

The default value is **true**.

allowResize

Type: Integer

The **allowResize** property of the **Table** class specifies whether the users can resize the rows and columns of a table. Resizing is achieved by using the mouse to drag the dividers between the fixed rows and columns of a table.

By default, both the rows and columns can be resized by the user.

The **Table** class constants and permitted values for the **allowResize** property are listed in the following table.

Table Class Constant	Value	Description
AllowResize_Any	0	Allow any resizing
AllowResize_Column	1	Allow column width resizing only
AllowResize_Row	2	Allow row height resizing only
AllowResize_None	3	Resizing is not permitted

The code fragment in the following example shows the use of the **allowResize** property.

```
// Set up column heading and sizes
tbl.allowResize     := Table.AllowResize_None;       // No resize allowed
tbl.row             := 1;
tbl.column          := 1;
tbl.text            := "Name";
tbl.columnWidth [1] := 155;
tbl.column          := 2;
tbl.text            := "Code";
tbl.columnWidth [2] := 80;
tbl.column          := 3;
tbl.text            := "Date Released";
tbl.columnWidth [3] := 155;
```

alternatingRowBackColor

Type: Integer

Availability: Read or write at any time

The **alternatingRowBackColor** property of the **ListBox** control (and the **JadeTableSheet** class) specifies an alternate row background color. By default, alternating list box entry and table rows have a background color of **Azure**. When you set this property to a value other than **Azure**, the specified value is used as the default background color of each alternate non-fixed row.

If the value of the **alternatingRowBackColorCount** property is **2**, the first, third, and so on, list box entry and non-fixed row default background color is the **backColor** property value of the list box or table sheet. The second, fourth, and so on, list box entry and non-fixed row default background color is the **alternatingRowBackColor** property value when it is not the default value (otherwise the **backColor** property value of the list box or sheet is used).

If the value of the **backColor** property of a list box entry, cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the entry or cell is ignored and the specific value of the **backColor** property is used.

Note When a list box entry or cell is drawn, the **backColor** property value is overridden by any specified **backColor** value set for that list box entry, cell, its row, or its column.

Note that when the list box or table is scrolled, the colors do not move with a row. The color scheme is applied to the rows, starting with the first visible non-fixed row; for example:

```
table1.accessSheet(1).alternatingRowBackColorCount := 2;
table1.accessSheet(1).alternatingRowBackColor := Azure;

listbox1.alternatingRowBackColorCount := 3;
listbox1.alternatingRowBackColor := DarkGray;
```

Applies to Version: 2018.0.01 and higher

alternatingRowBackColorCount

Type: Integer

Availability: Read or write at any time

The **alternatingRowBackColorCount** property of the **ListBox** control (and the **JadeTableSheet** class) specifies the number of list box entry or table rows at which the alternating background color of each visible list box entry row, non-fixed row, and non-fixed cell is displayed.

If the value of the **alternatingRowBackColorCount** property is:

- Less than or equal to zero (**0**), the background color of each list box entry or non-fixed cell defaults to the value of the **backColor** property of the list box or sheet, or of the list box or table itself if the value of the sheet is not specifically set. The **alternatingRowBackColor** property value is ignored.
- Greater than zero (**0**), for each visible **alternatingRowBackColorCount** list entry, non-fixed row, and non-fixed cell, the background color defaults to the value of the **alternatingRowBackColor** property.

For example, if the count is **2**, the first, third, fifth, and so on, list box entries, non-fixed rows, and the non-fixed cells in that row default to the value of the **backColor** property of the list box or sheet, while the second, fourth, sixth, and so on list entries, non-fixed rows, and the non-fixed cells in that row default to the value of the **alternatingRowBackColor** property.

If the value of the **backColor** property of a list box entry, cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the row or cell is ignored and the specific value of the **backColor** property is used.

Note that when the list box or table is scrolled, the colors do not move with a row. The color scheme is applied to the rows, starting with the first visible non-fixed row; for example:

```
table1.accessSheet(1).alternatingRowBackColorCount := 2;
table1.accessSheet(1).alternatingRowBackColor := Azure;

listbox1.alternatingRowBackColorCount := 3;
listbox1.alternatingRowBackColor := DarkGray;
```

Applies to Version: 2018.0.01 and higher

appletName

Type: String[128]

Availability: Run time only

The **appletName** property of the **WebJavaApplet** class contains the name of the compiled Java applet that is to be inserted into the generated HTML. The applet name is case-sensitive and has the following format.

```
<Java-applet-name>.<class>
```

automaticCellControl

Type: Boolean

Availability: Run time only

The **automaticCellControl** property of the **Control** class specifies whether the **Table** class handles the **cellControl** property automatically (for example, when performance is an issue when you are running in JADE thin client mode over a slow link). When this property is set to the default value of **false**, the **cellInputReady** event must set the text box value from the cell text if a text box is assigned to a table cell and the **TextBox** class **change** event must then set the cell text as it is changed, which may impact on JADE thin client performance, as an event has to be processed for each keystroke.

The **automaticCellControl** property has no meaning unless the control is assigned to a **Table** control as a **cellControl** property. See also the **Table** class **editMask** property.

The default value of **false** means that you must write logic to handle the **cellControl** property, but you can set the **automaticCellControl** property to **true** if you want the **Table** class to manage the **cellControl** property automatically, as follows.

- For a **TextBox** or **JadeEditMask** control, the table loads the text box with the value of the current cell. When the text box is changed, the table cell is updated and the **Table** class **change** event is called. All events are still called if they are defined, but they are generally not needed.
- For a **ComboBox** control, the table sets the combo box **listIndex** property using the text of the current cell. If a new combo box entry is selected, the table cell text is updated and the **Table** class **change** event is called.

The **automaticCellControl** property is currently ignored for any other type of control.

Note When the **automaticCellControl** property is set to **true**, arrow keys can be used to step to other cells for a text box or combo box in a table. If the caret is at the beginning of the text, the left arrow key steps to the prior cell. If the caret is on the top line, the up arrow key steps to the prior row. If the caret is at the end of the text, the right arrow key steps to the next cell or column. If the caret is on the last line of the text, the down arrow key steps to the next row.

In addition, pressing the Alt key and an arrow key performs the requested action regardless of where the caret is placed.

autoSize

Type: Integer (button, OCX, and table controls), Boolean (check box, option button, label, edit mask, status line, and multimedia controls)

Availability: Read or write at any time

The **autoSize** property determines whether a control is automatically resized to fit its contents. The following example shows the use of the **autoSize** property.

```
cb_autoSize_click(checkbox: CheckBox input) updating;
begin
    multimedia.autoSize := checkbox.value;
end;
```

The **autoSize** property settings for **Label** controls are listed in the following table.

Value	Description
true	Automatically resizes the control to fit its contents.
false	Keeps the size of the control constant (the default). Contents are clipped when they exceed the area of the control.

When the **autoSize** property for a **Label** control is set to **true**, the setting of the **wordWrap** property also affects the meaning.

The following table lists the results on the **Label** class **autoSize** property of setting the **wordWrap** property.

Value	Result
false	Text is placed in a single line.
true	Current width of the control is used to size the text using word wrap. This adjusts the height of the control to fit the text. If a single word cannot fit within the width, the label is widened to accommodate that word. If there is no text, the **autoSize** property is ignored.

For **Button** controls, this property determines whether the picture of the button is resized to fit the button. The **autoSize** property settings for **Button** controls are listed in the following table.

Button Class Constant	Value	Description
AutoSize_None	0	No resize of the button picture (the default).
AutoSize_Button	1	Button picture is proportionally resized to fit the button.
AutoSize_Picture	2	Button is resized to fit the picture.

When the value of the **autoSize** property for a **Button** control is set to **AutoSize_None** (**0**) or **AutoSize_Button** (**1**), the image is displayed to the left of the caption in the space remaining. (The caption takes precedence). The size of the icon selected from the assigned icon image is now based on the client height of the button.

When the value is **AutoSize_None** (**0**), the image is not scaled and is clipped to the area remaining after the caption is inserted. When the value is **AutoSize_Button** (**1**), the image is resized proportionally in the area remaining after the caption is inserted.

When the value of the **autoSize** property for a **Button** control is set to **AutoSize_Picture** (**2**), the non-border area of the button is set to the size of the picture set in the **picture** property. Only the picture is displayed; the caption of the button is not displayed.

If the **picture** property is not set, this option is ignored. This option is ideal for creating toolbar buttons that display only an image. If the picture image includes a border, turn off the **borderStyle** property of the button, to avoid duplicate border images.

The **autoSize** property settings for **Ocx** controls are listed in the following table.

Ocx Class Constant	Value	Description
AutoSize_Control	0	Resizes the ActiveX control to the size defined by Painter or your logic (the default)
AutoSize_Object	1	Resizes the ActiveX control to the size of the OLE object

The **autoSize** property settings for **CheckBox** and **OptionButton** controls are listed in the following table.

Value	Description
true	The control is resized to fit the content (the default); that is, the width is the size of the icon displayed plus the width of the text, and the height is the minimum required to show the content.
false	The control height and width are set in the JADE Painter or by logic, except if the height is less than size required to fit the content, in which case the height becomes the minimum required to show the content vertically. If the height is larger than required, the content is centered vertically.

The **autoSize** property settings for **StatusLine** controls are listed in the following table.

Value	Description
true	The height of the status line is adjusted according to the height of any controls placed inside the status line and the font size of the **caption** property for the status line.
false	The height of the status line is set by you (the default).

If the value is **false**, the status line control can be resized vertically; if **true**, the control cannot be manually resized. When a child control is resized, the status line control is also resized.

When the value of the **StatusLine** control **autoSize** property is **true**:

- If a child control **left** property value position is less than zero, the control is moved to be zero (**0**) when the **parentAspect**, **relativeLeft**, and **relativeWidth** property values of the child do not affect the horizontal position (not stretch horizontal, anchor right, and centered horizontal, and the **relativeLeft** and **relativeWidth** property values are **false**).
- If a child control is not fully visible horizontally, the child is right-aligned in the status line control if it can be fully displayed or positioned at zero (**0**) if it cannot when the **parentAspect**, **relativeLeft**, and **relativeWidth** property values of the child do not affect the horizontal position (not stretch horizontal, anchor right, and centered horizontal, and the **relativeLeft** and **relativeWidth** property values are **false**).
- If a child control top position is less than zero (**0**), the control is moved to be zero when the **parentAspect** property value of the child does not affect the vertical position (not stretch vertical, anchor bottom, or centered vertical).
- If a child control is not fully visible vertically, the child is bottom-aligned in the **StatusLine** control when the **parentAspect** property value of the child does not affect the vertical position (not stretch vertical, anchor bottom and centered vertically).
- The values of the **relativeTop** and **relativeHeight** properties of child controls are always set to **false**, as their functionality is not compatible with auto-sizing the height of the **StatusLine** control (as has always been the case).
- All **parentAspect** flag values and **relativeLeft** and **relativeWidth** values are applied.

The height of the **StatusLine** control is then determined by analyzing the child control as follows, to determine the maximum height required. For a child control that does not have a fixed height and has the **parentAspect** property with the:

- **ParentAspect_StretchBottom** flag set, the height required is the **top** position, **height**, and **parentBottomOffset** property values of the child.
- **ParentAspect_AnchorBottom** flag set, the height required is the **height** and **parentBottomOffset** property values of the child; otherwise, the **height** of the child control.

The **autoSize** property determines whether the **MultiMedia** control is automatically sized according to the size of the displayed image.

The **autoSize** property settings for **MultiMedia** controls are listed in the following table.

Value	Description
true	The device or file being played does not have a playback image, the values of the **showPlayBar**, **showName**, **showMode**, and **showPosition** properties are **false**, and the control has no size and is not visible. The size of the control is set to the size of the displayed image (multiplied by the value of the **zoom** property divided by **100**) plus the size of any playbar and caption. If the device or file being played does not involve a playback image, the control is not visible. If the value of the **zoom** property is zero (**0**), **100** is used.
false	When the value of the **zoom** property is zero (**0**), the displayed image is stretched to fit the client area of the control. When the value of the **zoom** property is not zero, the control size is not affected by the size of the displayed image. Only as much of the displayed image that fits in the client area of the control is displayed.

The **autoSize** property settings for **Table** controls are listed in the following table.

Table Class Constant	Value	Description
AutoSize_None	0	No automatic sizing
AutoSize_Row	1	Row resized to fit the table
AutoSize_Column	2	Column resized to fit the table
AutoSize_Both	3	Row and column resized to fit the table (the default)
AutoSize_ColumnMinimum	4	Column resized to the minimum
AutoSize_BothColumnMinimum	5	Both row and column resized to the column minimum

The code fragment in the following example shows the use of the **autoSize** property for the **Table** class.

```
    table1.autoSize := comboAutoSize.listIndex - 1;
```

When the **autoSize** property for the **Table** control is set to:

- **AutoSize_None** (**0**), each column is set to 50 pixels wide and the height of rows is set to the size of a single line of text using the default font for that table.

- **AutoSize_Row** (**1**), the height of rows is set so that all cells in the row can be fully displayed vertically, with the text determined by the font and picture.

  Row automatic sizing has little effect on tables unless you include unstretched pictures in cells, as row heights are set automatically within the table according to the text size and font (which may require word wrap).

  Note Text in table cells wraps only when the value of the **autoSize** property is set to **AutoSize_Row** (**1**).

- **AutoSize_Column** (**2**), the width of columns is set so that all cells in the column can be fully displayed horizontally without word wrapping. The text is determined by the font and picture. The width of the columns is padded, as described following this list.

- **AutoSize_Both** (**3**), the width of columns is established, followed by the height of the row.

The width of the columns is padded, as described following this list.

- **AutoSize_ColumnMinimum** (**4**), the column is resized to the minimum size (that is, the size of the cell).
- **AutoSize_BothColumnMinimum** (**5**), both the width of the column and then the height of the row are resized to the minimum size.

When using the **AutoSize_Both** and **AutoSize_Column** values, if all columns fit within the client width of the table control, the width of each column is incremented so that the columns are stretched to exactly fit the table client width. If not all columns fit within the table client width, the **AutoSize_Row** and **AutoSize_Column** values are equivalent to the **AutoSize_ColumnMinimum** and **AutoSize_BothColumnMinimum** values.

Note Use the **JadeTableColumn** class **maxColumnWidth** property to specify a maximum width (in pixels) for a column when determining the width during the column width auto-size processing.

If you require both column and row automatic sizing, the width of the column is established, followed by the height of the row. When the **autoSize** property is set to **AutoSize_None** (**0**), cells have a minimum width of 50 pixels.

When the property is set to a **Table** class constant representing a non-zero value, cells have a minimum width of 20 pixels, even when they are empty. If a user resizes a row or a column or your JADE logic specifically sets the row height or column width, the automatic sizing of the row or column no longer applies at any subsequent change of cell contents.

The **autoSize** property of the **JadeEditMask** class determines whether a control is automatically resized to fit its contents. If the value of **autoSize** is **false** (the default), the height of the child text box or text boxes is set to the height of the edit mask control. If the last field is a text box child, it has its width expanded to the right edge of the control. Any labels that are defined in the mask are centered vertically in the control. However, if the top position or the height of a text box child is set by the **mask** property, the text box children are automatically sized except for any specific **width** or **height** settings defined in the mask for a child. If there is insufficient room to show a full single line of text in a child text box, the height of the control is expanded regardless of the setting of the **autoSize** property.

If the value of **autoSize** is **true**, the size of the control is determined by the maximum size and height of the literal and text data that can be entered into that field according to the value of the **mask** property (a single line of text).

The size is based on the largest character that can be entered into each character position, which generally means that the control is larger than required. For example, a mask of three alphabetic characters **WWW** takes up considerably more space than **iii**.

In addition, the size can be larger than required because the prompt character requires more space than the character for which it is prompting. The fields of the control are always defined according to the maximum size of the characters in that field, regardless of the value of the **autoSize** property.

autoSpacingX

Type: Integer

Availability: Read or write at any time

The **autoSpacingX** property of the **JadeDockBar** class determines the horizontal spacing in pixels between each child in the **JadeDockBar** control.

Note This property applies only when the **alignChildren** property is set to **AlignChildren_Auto** (**3**).

Children on the left of the dock bar are positioned at half the value of the **autoSpacingX** property from the left.

Spacing of half the value of the **autoSpacingX** property is retained after the end of the last child to the right.

autoSpacingY

Type: Integer

Availability: Read or write at any time

The **autoSpacingY** property of the **JadeDockBar** class determines the vertical spacing in pixels between each child in the **JadeDockBar** control.

Note This property applies only when the **alignChildren** property is set to **AlignChildren_Auto** (**3**).

Children at the top of the dock bar are positioned at half the value of the **autoSpacingY** property from the top. Spacing of half the value of the **autoSpacingY** property is retained after the end of the last child.

autoTab

Type: Boolean

Availability: Read or write at any time

The **autoTab** property of the **TextBox** class automatically moves focus to the next control in the tab order of a form when this property is set to **true** and the **maxLength** property is set to a non-zero value, so that the user does not have to press the Tab key when the user enters the final character into a text box at the position specified by the **maxLength** property.

When the **autoTab** property is set to the default value of **false**, focus is not automatically shifted when the text box becomes full and the user must press the Tab key to cause the focus to move to the next control in the tab order of the form.

Note When the **autoTab** property is set to **true** and the final character is entered in the text box, the focus is shifted after completion of the **keyUp** event method. However, the **autoTab** property is ignored if your application code has already moved focus to another control.

The **autoTab** property of the **JadeEditMask** class takes effect when the control has an edit mask and the last character in that text box field (according to the value of the **mask** property) is entered.

When the value of the **autoTab** property is **false** (the default), no further action occurs and the user must press the Tab key to skip to the next field. If the value of the **autoTab** property is **true**, focus is shifted to the start of the next text box field of the control if there is one or to the next control in the order specified by the **tabIndex** property of the form if focus is on the last text box field.

The **Table** class handles the **autoTab** property of a **TextBox** or **JadeEditMask** control used as a cell control. When the user enters a character into a cell control that results in the **autoTab** process being invoked, the table sets the current column to the next visible, enabled cell of the current row.

If there no such cell, the process moves to the next row and searches for the next visible, enabled cell in that row and so on, until a visible, enabled cell is located. If the end of the table is reached and the **tabOffEnds** property of the **JadeTableSheet** class is set to **true**, focus is shifted to the next control in the order specified by the **tabIndex** property of the form.

This process takes effect if the user enters text that completes the text and the cell has:

- An effective **inputType** of **InputType_TextBox**, **InputType_SignedNumeric** or **InputType_SignedNumeric** and the **maxLength** for the cell is set and the default cell control object has **autoTab** set to **true**, as shown in the code fragment in the following example.

```
        table.inputType := Table.InputType_TextBox;
        table.maxLength := 7;
        table.cellControl.TextBox.autoTab := true;
```

- A **TextBox** cell control with a **maxLength** > 0 and **autoTab** set to **true**.
- A **JadeEditMask** control with **autoTab** set to **true**.

autoURLDetect

Type: Boolean

Availability: Read or write at any time

The **autoURLDetect** property of the **JadeRichText** control specifies whether the control automatically formats a URL when it is typed into the control. (For an example of the use of this property, see "JadeRichText Control Method Example", earlier in this document.)

As the default value for this property is **false**, a URL is not automatically detected.

backBrush

Type: Binary

Availability: Read or write at any time

The **backBrush** property of the **Form** class enables you to tile the background of the form with the bitmap when the property is set to a bitmap. Tiling means that as many copies of the bitmap as are required cover the entire visible form surface. (The process is the same as the Windows Desktop tiling process.) When the **backBrush** property is set for a form, the **backColor** of the form is not used for coloring the background of the form. To clear the **backBrush** property of a form, set the property to **null**.

Note The **Window** class **backBrushStyle** property controls the way in which the **Control** class and **Form** class **backBrush** property images are displayed.

To define a binary that is used instead of the **Window** class **backColor** property, use the **Control** class **backBrush** property. If you set the **backBrush** property to a valid image (only bitmap, PNG, GIF, TIFF, and JPEG files are supported) and the control is not transparent, the control background is drawn repeatedly by using the specified back-brush to cover the entire control. This picture could be a pattern, an actual picture image, a logo, and so on.

Although you can define a **backBrush** property value for all controls, the **BrowseButtons**, **Button**, **Folder**, **MultiMedia**, **Ocx**, and **ActiveXControl** controls ignore it because the control is entirely covered by other drawing.

Use the **transparent** property to enhance the use of the **backBrush** property; for example, to paint only the text of a label over the **backBrush** bitmap without erasing the background area of the label.

The **backColor** property is ignored when the **backBrush** property is set. The **null** default value indicates that the **backColor** property is used to draw the background area of the control unless it is also transparent.

Note As both the Microsoft Edit control draws the text by using the value of the **backColor** property rather than that of the **backBrush** property, the **backBrush** property is ignored for a **TextBox** control and the text box part of a **ComboBox** control.

For details about displaying a background image on a Web form, see the **webFileName** property.

backBrushStyle

Type: Integer

Default: 0

Availability: Read or write at any time

The **backBrushStyle** property of the **Window** class controls the way in which the **Control** class and **Form** class **backBrush** property images are displayed, as shown in the **Form** class **load** method in the following example.

```
load() updating;
begin
    self.backBrushStyle := Form.BackBrushStyle_Center;
end;
```

If the value of the **backBrush** property is null (**""**), the **backBrushStyle** property is ignored.

If the value of the **backBrush** property has an assigned image, the **backBrushStyle** property determines the display style, as indicated by the following **Window** class constant values.

Window Class Constant	Integer Value	The image is...
BackBrushStyle_Tile	0	Converted to a brush and the brush is tiled over the entire client area of the window.
BackBrushStyle_Stretch	1	Stretched to cover the entire client area.
BackBrushStyle_Center	2	Displayed at actual size in the center of the client area.
BackBrushStyle_StretchProport	3	Drawn in isotropic proportions to its original size to fit the client area of the window. If the resized image is narrower than the client width, it is centered horizontally. If the resized image is narrower than the client height, it is centered vertically.

backColor

Type: Integer

Availability: Read or write at any time

The **backColor** property of the **Window** class contains the background color of a window. Changing this property causes a repaint of the window object. (See the **Window** class **getSystemColor** method for a description and example of returning a Windows system color. See also the **backBrush** property.)

For a **Table** control, the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the **backColor** property for the current sheet, cell, column, or row is being accessed.

For a control, setting the **backColor** property in the JADE Painter by selecting the **Parent's Colour** item (or setting the value of the **backColor** property to **#80000000** at run time) instructs the window to use the **backColor** property value of its parent.

For the **JadeTextEdit** control, the **clearAllStyles** method copies the value of this property to the default text style; that is, **STYLE_DEFAULT** (**32**).

JADE uses the RGB scheme for colors. Each property can be set by using the appropriate RGB value. For most controls in the JADE development environment, the default value for the **backColor** property is set to the Windows Background color, which is usually white. For the form and the browse, frame, status line, and folder controls, the default background color is the Windows 3D Face color, which is usually light gray.

The valid range for a normal RGB color is **0** through **16,777,215** (#FFFFFF). The high byte of an integer in this range equals **0**; the lower three bytes (from least to most significant byte) determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is **128**, JADE uses the system colors, as defined in the Control Panel of the user. To determine **Integer** value of a color from the RGB values:

```
int:= RedValue + (GreenValue * 256) + (BlueValue * 256 * 256);
```

The **backColor** property is available on all **Window** objects, but on some controls it is not used when painting the control image; for example, a scroll bar. Setting the **backColor** property of an object causes all graphics drawn on the form or control to be erased. For an MDI frame form, the MDI client window is painted using the **backColor** value of the form. Although you can define a **backColor** property value for all controls, the **BrowseButtons**, **Button**, **Folder**, **MultiMedia**, **Ocx**, and **ActiveXControl** controls ignore it because the control is entirely covered by other drawing.

The **backColor** property is ignored when the **backBrush** property is set. The **null** default value indicates that the **backColor** property is used to draw the background area of the control unless it is also transparent.

For details about printing a background picture over which is drawn the report itself when the **backColor** property is not set to white, see "Layering Print Output", under the **Printer** class "Defining Your JADE Report Layouts", in Chapter 1.

The following examples show the use of the **backColor** property.

```
tranState_click(btn: Button input) updating;
begin
    if process.isInTransactionState then
        tranState.backColor := Green;
        tranState.caption   := 'Begin Transaction';
        commitTransaction;
    else
        tranState.backColor := Red;
        tranState.caption   := 'End Transaction';
        beginTransaction;
    endif;

table1.accessedCell.backColor := Yellow;

dblClick(frame: Frame input) updating;
vars
    colorDialog : CMDColor;
begin
    create colorDialog;
    colorDialog.color := frame.backColor;      // set initial color displayed
    if colorDialog.open = 0 then               // not cancelled and no error
        self.backColor := colorDialog.color;   // use the returned value
    endif;
    delete colorDialog;                        // tidy up
end;
```

bevelColor

Type: Integer

Availability: Read or write at any time

The **bevelColor** property contains the color used to paint the bevel areas of a three-dimensional **Frame** or **StatusLine** control.

For a raised bevel (a control where the **bevelInner** or **bevelOuter** property is set to **Bevel_Raised** (**2**)), the **bevelColor** property applies to the bevel portion on the left and top sizes of the control.

For an inset bevel (the **bevelInner** or the **bevelOuter** property is set to **Bevel_Inset** (**1**)), the **bevelColor** property applies to the bevel portion on the bottom and right sizes of the control.

JADE uses the RGB scheme for colors. Each property can be set by using the appropriate RGB value.

bevelInner

Type: Integer

Availability: Read or write at any time

The **bevelInner** property contains the style of the inner bevel of the **Frame** and **StatusLine** controls.

The following table lists the settings for the **bevelInner** property.

Constant	Value	Description
Bevel_None	0	No bevel is drawn.
Bevel_Inset	1	The bevel appears inset on the screen.
Bevel_Raised	2	The bevel appears raised on the screen.

Use the **bevelInner** property with the **bevelInnerWidth** property. If the **bevelInner** property is set to **Bevel_None** (**0**), the **bevelInnerWidth** property has no effect.

bevelInnerWidth

Type: Integer

Availability: Read or write at any time

The **bevelInnerWidth** property contains the width of the bevel along the four sides of the **Frame** and **StatusLine** controls to determine the height of the three-dimensional shadow effect. The **bevelInnerWidth** property is the size for the inner bevel.

The setting for this property determines the number of pixels that are used to draw the bevel that surrounds the control.

If the **bevelInner** property is set to **Bevel_None** (**0**), the **bevelInnerWidth** property has no effect.

bevelOuter

Type: Integer

Availability: Read or write at any time

The **bevelOuter** property contains the style of the outer bevel of the **Frame** and **StatusLine** controls.

The following table lists the settings for the **bevelOuter** property.

Constant	Value	Description
Bevel_None	0	No bevel is drawn.
Bevel_Inset	1	The bevel appears inset on the screen.
Bevel_Raised	2	The bevel appears raised off the screen.

Use the **bevelOuter** property with the **bevelOuterWidth** property. If the **bevelOuter** property is set to **Bevel_None** (**0**), the **bevelOuterWidth** property has no effect.

bevelOuterWidth

Type: Integer

Availability: Read or write at any time

The **bevelOuterWidth** property contains the width of the bevel along the four sides of the **Frame** and **StatusLine** controls to determine the height of the three-dimensional shadow effect. This property is the size for the outer bevel.

The setting for this property determines the number of pixels that are used to draw the bevel that surrounds the control.

If the **bevelOuter** property is set to **Bevel_None** (**0**), the **bevelOuterWidth** property has no effect.

bevelShadowColor

Type: Integer

Availability: Read or write at any time

The **bevelShadowColor** property contains the color used to paint the bevel areas of a three-dimensional **Frame** or **StatusLine** control.

For a raised bevel (a control where the **bevelInner** or **bevelOuter** property is set to **Bevel_Raised** (**2**)), the **bevelShadowColor** property applies to the bevel portion on the bottom and right sides of the control.

For an inset bevel (the **bevelInner** or the **bevelOuter** property is set to **Bevel_Inset** (**1**)), the **bevelShadowColor** property applies to the bevel portion on the left side and the top of the control.

JADE uses the RGB scheme for colors. Each property can be set by using the appropriate RGB value.

borderColorSingle

Type: Integer

Availability: Read or write at any time

The **borderColorSingle** property of the **Control** class contains the RGB scheme color of the control border when the **borderStyle** property is set to **BorderStyle_Single** (1).

When the border style is single, you can use this property to set the border color to a specific color if you do not want the default color of black used; for example, you could set the value to red (**255**) to indicate at run time that information has not been provided in that control.

borderHeightBottom

Type: Integer

Availability: Read or write at any time

The **borderHeightBottom** property of the **JadeDockBase** class indicates how many extra pixels are drawn in the border area at the bottom of the control. This property applies regardless of the setting of the **Window** class **borderStyle** property.

The default value for the **JadeDockBar** and **JadeDockContainer** controls is **1** pixel.

No border area is drawn for the docking control when it is the child of a floating form.

borderHeightTop

Type: Integer

Availability: Read or write at any time

The **borderHeightTop** property of the **JadeDockBase** class indicates how many extra pixels are drawn in the border area at the top of the control. This property applies regardless of the setting of the **Window** class **borderStyle** property.

The default value for the **JadeDockBar** and **JadeDockContainer** controls is 1 pixel.

No border area is drawn for the docking control when it is the child of a floating form.

borderStyle

Type: Integer

Availability: Read or write at any time

The **borderStyle** property contains the border style for a window. The **borderStyle** property settings for a **Form** subclass are listed in the following table.

Window Class Constant	Value	Description
BorderStyle_None	0	None. A form with no border has no **Maximize** or **Minimize** button, Control-Menu icon, or caption display. If the form has a menu, the setting is changed to fixed single.
BorderStyle_Single	1	Can include Control-Menu icon, title bar, **Maximize** button, and **Minimize** button. Resizable only using **Maximize** and **Minimize** buttons. For forms defined as Web pages, the border is single and sizable.

Window Class Constant	Value	Description
BorderStyle_Sizable	2	For forms defined as Web pages, the border is double (the default).
BorderStyle_Double	3	Not resizable.

Although a form can have any border style, you must ensure that the user can unload the form from the logic associated with a button or menu item if the form does not have a Control-Menu icon.

Caution Defining an MDI child form with the **controlBox**, **maxButton**, or **minButton** property set to **false** affects the way that the form is displayed and behaves when it is maximized. For example, the form cannot be restored to a non-maximized state without switching to another MDI child form, by using Ctrl+F6, and then using the **Restore** Control-Menu command of that form.

If the form is skinned, the default form system menu reflects whether Java allows maximize and minimize operations according to the value of the **borderStyle** property.

The **borderStyle** property settings listed in the following table apply to individual control classes. (The border style that you define for a specific control does not affect any global settings in the application.)

Class Constant	Class	Value	Description
BorderStyle_None	Window	0	No border
BorderStyle_Single	Window	1	Fixed-single-line border (the default)
BorderStyle_3DSunken	Control	2	Sunken three-dimensional effect (two pixels)
BorderStyle_3DRaised	Control	3	Raised three-dimensional effect (two pixels)

When the **borderStyle** property is set to **BorderStyle_Single**, you can set the border color to a specific color if you do not want the default color of black used; for example, you could set the value of the **borderColorSingle** property to red (**255**) to indicate at run time that information has not been provided in that control.

All controls can use the border styles listed in this table, with the following exceptions.

- **Folder** control, as no border is ever drawn.

  If the value of the **borderStyle** property for the active sheet of a **Folder** control is set to any value other than **BorderStyle_None**, the folder does not display a border of its own for the sheet area.

- **Ocx**, **ActiveXControl**, and **MultiMedia** classes permit the **BorderStyle_None** and **BorderStyle_Single** styles only, because the border of these controls is not drawn by JADE.

- **GroupBox** class, as the **BorderStyle_3DSunken** and **BorderStyle_3DRaised** styles produce the existing three-dimensional effect.

  The border of a **GroupBox** control is usually black. If the **backColor** property of a group box is set to gray, the border is drawn with three-dimensional (3D) effects.

No border area is drawn for a docking control when it is the child of a floating form.

Windows will not create a form that has a **caption** line with a single border but instead creates a form with a fixed double border. If the value of the **borderStyle** property is set to **BorderStyle_Single** (**1**) and the form has a caption line, the value of the **borderStyle** property is modified to fixed **BorderStyle_Double** (**3**), to reflect the actual border style in use.

Notes If the **show3D** property is set to **Show3D_UseAppDefault** (**0**) and the application default setting of the **show3D** property for the control is **true** (that is, the control is selected in the **3D Controls** list box on the **Form** sheet of the Define Application dialog) or if the **show3D** property for the control is set to **Show3D_Use3D** (**2**), the effective border is a sunken three-dimensional effect (that is, **BorderStyle_3DSunken**), regardless of the setting of the **borderStyle** property.

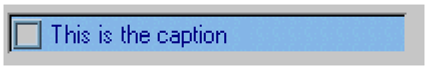
If the **borderStyle** property is set to **BorderStyle_3DSunken** (**2**) or **BorderStyle_3DRaised** (**3**), the **show3D** property is reset to **Show3D_UseBorderStyle** (**3**). If you set the **show3D** property to **Show3D_UseBorderStyle** (**3**), only the **borderStyle** property is used to control the actual border style that is displayed.

If a control is painted using the 3D feature, the border of that control is painted in the 3D style rather than in black.

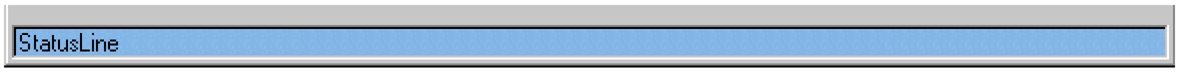
The code fragment in the following example shows the use of the **borderStyle** property.

```
    lbl.borderStyle := BorderStyle_Single;
```

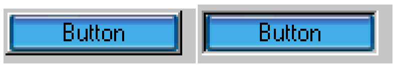
The following is an example of a check box control with a sunken border and a back-brush that has been set.



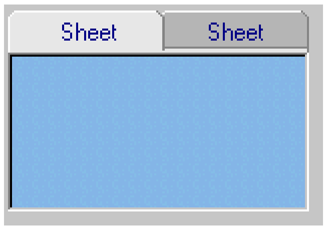
The following is an example of a status line that has a sunken three-dimensional border, a back-brush, and no value set for the **bevelInner**, **bevelOuter**, or **boundaryWidth** property.



The first of the following examples shows a button that has a raised three-dimensional effect and the second example is a button that has a sunken three-dimensional effect.



The following is an example of a sheet that has a sunken three-dimensional effect.



borderWidthLeft

Type: Integer

Availability: Read or write at any time

The **borderWidthLeft** property of the **JadeDockBase** class indicates how many extra pixels are drawn in the border area at the left of the control. This property applies regardless of the setting of the **Window** class **borderStyle** property.

No border area is drawn for the control when it is the child of a floating form.

The default value for the **JadeDockBar** and **JadeDockContainer** controls is **1** pixel.

borderWidthRight

Type: Integer

Availability: Read or write at any time

The **borderWidthRight** property of the **JadeDockBase** class indicates how many extra pixels are drawn in the border area at the right of the control. This property applies regardless of the setting of the **Window** class **borderStyle** property.

No border area is drawn for the control when it is the child of a floating form

The default value for the **JadeDockBar** and **JadeDockContainer** controls is 1 pixel.

boundaryBrush

Type: Integer

Availability: Read or write at any time

The boundary properties control the boundary section of a 3D bevel area of a **Frame** or **StatusLine** control. The boundary area is that area between the inner bevel and the outer bevel of the control.

The **boundaryBrush** property determines whether the boundary area is of a plain color or is painted with a dotted brush.

The settings of the **boundaryBrush** property are listed in the following table.

Constant	Value	Description
BoundaryBrush_Solid	0	Solid brush
BoundaryBrush_Dotted	1	Dotted brush

boundaryColor

Type: Integer

Availability: Read or write at any time

The boundary properties control the boundary section of a 3D bevel area of a **Frame** or **StatusLine** control. The boundary area is that area between the inner bevel and the outer bevel of the control.

The **boundaryColor** property contains the color of the boundary area. The boundary area can be omitted by setting the **boundaryWidth** property to zero (**0**).

boundaryWidth

Type: Integer

Availability: Read or write at any time

The boundary properties control the boundary section of a 3D bevel area of a **Frame** or **StatusLine** control. The boundary area is that area between the inner bevel and the outer bevel of the control.

The **boundaryWidth** property contains the width of the boundary area. If the width is set to zero (**0**), the **boundaryBrush** and **boundaryColor** properties do not apply. The width is measured in pixels.

bubbleHelp

Type: String

Availability: Read or write at any time

The **bubbleHelp** property contains the text that can be displayed as bubble help. This property is defined in the **Window** class, but it is implemented only by control subclasses.

Use the **Application** class **showBubbleHelp** property to control whether bubble help is displayed at run time.

Controls with bubble help text display that text in a bubble below or above the control after the mouse has been positioned over the control for more than a half a second. The bubble help is removed when the mouse is moved off the control, when a mouse button is pressed, or after five seconds.

If the window implements the **mouseHover** event, bubble help is displayed after the **mouseHover** event has been executed. This allows the **mouseHover** event to set the **bubbleHelp** text that is appropriate for the mouse position; for example, the list entry that the mouse is over.

Bubble help is:

- Displayed only when the application of the control is active and that form has focus.
- Not displayed for a control if the **bubbleHelp** text contains spaces only.

This property can be translated when the value of the **Schema** class **formsManagement** property is **FormsMngmt_Single_Multi** (**2**).

See the **ComboBox** class or **ListBox** class for details about automatic bubble help that is displayed for combo boxes and list boxes if the combo box or list box does not have bubble help text defined for it by using the **bubbleHelp** property.

In a **Folder** control, bubble help defined for a sheet is displayed below or next to the tab of that sheet, depending on the setting of the **tabsPosition** property, when the cursor is positioned over the tab. When the cursor is positioned over the sheet itself, no bubble help is displayed. In addition, bubble help is disabled when dragging is in progress. (For details, see the **dragMode** property.)

If bubble help is currently displayed and the next window to which the mouse is moved also has bubble help text, there is no delay in the display of the bubble help for that next control. Bubble help can be of any length. If the bubble help cannot fit in one line from where it is displayed, multiple lines with word wrap are displayed.

The code fragment in the following example shows the use of the **bubbleHelp** property.

```
if listbox.getListIndex(x, y) <> -1 and
    listbox.getListIndex(x, y) <> lastIndex then
    cust := listbox.itemObject [listbox.getListIndex(x, y)].Customer;
    listbox.bubbleHelp := cust.name.toUpper & Cr
                          & "________________________________________"
                          & Cr & cust.address & Cr & cust.contact;
endif;
```

bulletIndent

Type: Integer

Availability: Read or write at any time

The **bulletIndent** property of the **JadeRichText** control contains the indentation (the minimum space in pixels between the bullet or paragraph number and the paragraph text) that is used when the **bulletStyle** property has an integer value other than zero (**0**).

The default value of zero (**0**) indicates that text in the paragraph is not indented.

bulletStyle

Type: Integer

Availability: Read or write at any time

The **bulletStyle** property of the **JadeRichText** control contains the bullet style of the current paragraph. The **bulletStyle** property values are listed in the following table.

JadeRichText Class Constant	Integer Value
BulletStyle_Dot	1
BulletStyle_Lowercase	3
BulletStyle_LowercaseRoman	5
BulletStyle_None (the default)	0
BulletStyle_Number	2
BulletStyle_Uppercase	4
BulletStyle_UppercaseRoman	6
ParagraphFormat_Undefined	#80000000

You cannot assign the **ParagraphFormat_Undefined** value, but it is returned to indicate that the selected text contains multiple paragraphs with different **bulletStyle** property values. Although you can define other values, if you do so, the **BulletStyle_Dot** value is assigned to the paragraph.

For an example of the use of this property, see "JadeRichText Control Method Example", earlier in this document.

buttonPicture

Type: Integer

Availability: Read or write at any time

The **buttonPicture** property provides predefined bitmaps to be placed on a **Button** control. These pictures override any **picture** property setting for the button.

The available pictures and their settings are listed in the following table.

Button Class Constant	Value	Picture	Description
ButtonPicture_None	0	None	No preset picture is displayed (the default)
ButtonPicture_Tick	1	✓	Green check mark
ButtonPicture_Cross	2	✗	Red cross
ButtonPicture_No	3	🚫	Red no-entry symbol
ButtonPicture_Door	8	🚪	Closing door
ButtonPicture_RecycleBin	9	♻	Recycle bin
ButtonPicture_Bin	10	🗑	Trash can

The appearance of the standard buttons and their settings listed in the following table may differ, depending on the operating system on which the JADE application is running.

Button Class Constant	Value	Picture	Description
ButtonPicture_Stop	4	❌	Stop sign
ButtonPicture_Question	5	❓	Question mark
ButtonPicture_Exclamation	6	⚠	Exclamation mark
ButtonPicture_Asterisk	7	ℹ	Asterisk

Pictures are painted to the left of the button caption.

If the button is assigned a **pictureDisabled** image and the button is then disabled, this image is displayed instead.

cachePictures

Type: Boolean

Availability: Read or write at run time only

The **cachePictures** property of the **Picture** class has meaning only when the application is running in JADE thin client mode and when the picture object is attached to a physical window.

By default, setting the picture properties of a **Picture** or **JadeMask** control at run time causes those pictures to be stored in the cache file of the application server and the cache file of the presentation client unless the **UseCacheFile** parameter in the [JadeThinClient] section of the JADE initialization file is set to **false**.

However, you can set this property to **false** in situations where you do not require this caching because the picture is only ever downloaded once and will therefore unnecessarily add entries to those cache files. Examples of where you could set this property to **false** are:

- Using the **Window** or **Control** class **createPicture** method to create a dynamic runtime image that is then displayed
- Displaying a customer invoice that was scanned in

Setting this property to **false** affects the:

- **Picture** class **picture**, **pictureDown**, and **pictureDisabled** properties
- **Picture** class **setPicture** method
- **JadeMask** class **pictureRollOver**, **pictureRollUnder**, and **pictureMask** properties

cancel

Type: Boolean

Availability: Read or write at any time

The **cancel** property applies to **Button** and **JadeMask** controls. If the **cancel** property is set to **true**, that button or mask control is marked as the **Cancel** button. Assign this status to a button marked **Cancel**. (Pressing Esc has the same effect as pressing the **Cancel** button or mask control.)

Note If the form has a **Button** control with the **cancel** property set to **true** and an open combo box has focus, the Esc key is processed by the combo box; not by the **Cancel** button. The **Cancel** button action occurs only if the combo box list is not open.

When the user presses Esc, if there is an enabled visible button or mask control with the **cancel** property set to **true** on the same form, the focus is transferred to that button or JADE mask control and a **click** event is caused on the button or mask control, regardless of which control on the form had the focus. The default value is **false**.

It is your responsibility to ensure that only one button or JADE mask control on a form has the **cancel** property set to **true**. If not, the first such enabled visible button or mask control that is encountered with that status receives the event. You must write logic to respond to the event.

canHaveFocus

Type: Boolean

Availability: Read or write at any time (**JadeMask**) or at run time only (**BaseControl**)

The **canHaveFocus** property of the **BaseControl** class or **JadeMask** class specifies whether the base control or JADE mask control can have focus.

If focus is denied, the control still responds to mouse clicks and mouse-over situations.

The default value of **true** indicates that the control can accept focus. If the control should not have focus, set the property value to **false** (for example, you would usually do so in the **windowCreated** method of the **BaseControl** subclass).

canPaste

Type: Boolean

Availability: Read-only at run time

The **canPaste** property of the **JadeTextEdit** control specifies whether the clipboard contains text that can be pasted into the control and the value of the **readOnly** property is **false**.

canRedo

Type: Boolean

Availability: Read-only at run time

The **canRedo** property of the **JadeTextEdit** control specifies whether the undo and redo action list contains editor actions that can be redone.

canUndo

Type: Boolean

Availability: Read-only at run time

The **canUndo** property of the **JadeTextEdit** control specifies whether the undo and redo action list contains editor actions that can be undone.

caption

Type: String

Availability: Read or write at any time

For the **Form** class, the **caption** property contains the text displayed in the title bar of the form. When the form is minimized, this text is displayed on the task bar.

When you create a new object, its default caption is the same as the default name for both forms and controls. This default caption includes the object name and an integer; for example, **text1** or **Form1**. For a more descriptive label, set the **caption** property to the required value.

The code fragment in the following example shows the use of the **caption** property.

```
    caption := caption &  '  -  ' & process.signOnUserCode;
```

In the **Button**, **CheckBox**, **JadeDockBase**, **Form**, **Frame**, **GroupBox**, **JadeMask**, **Label**, **MenuItem**, **OptionButton**, **Sheet**, **StatusLine** classes, this property can be translated when the value of the **Schema** class **formsManagement** property is **FormsMngmt_Single_Multi** (**2**).

You can use the **caption** property to assign an access key to a control or menu.

In the caption, include an ampersand character (**&**) immediately preceding the character you want for an accelerator (shortcut) key. That character is then underlined.

Press Alt and the underlined character to move the focus to that control. To include an ampersand character in a caption without creating an access key, enter two ampersand characters (**&&**). A single ampersand is displayed in the caption and no characters are underlined.

Note The **tabIndex** property of a control affects its associated accelerator key. If you press the accelerator key for a control that does not have the tab stop ability, the focus moves to the next enabled and visible control in the tab order that can receive the focus.

The maximum length of a caption for a form is 255 characters. The maximum length for controls that have captions is 32,767 characters. The maximum length of a caption for a menu is 100 characters.

If a form has a menu, a Control-Menu icon, or a **Maximize** or **Minimize** button, the caption area is always displayed, regardless of whether the caption is empty or not. Alternatively, if the **borderStyle** property is set to **BorderStyle_None** (no caption) or the caption is empty, the form does not display the caption area of the form.

When an MDI child form is maximized within an MDI form, the caption of the child form is included with the caption of the parent form.

For a **JadeMask** control, caption alignment (that is, word wrapping) occurs within the rectangle defined by the **captionLeft**, **captionTop**, **captionHeight**, and **captionWidth** properties. If the value of the **captionWidth** property is zero (**0**), the width of the caption region is the value of the **clientWidth** property less the value of the **captionLeft** property. If the value of the **captionHeight** property is zero (**0**), the height is the value of the **clientHeight** property less the value of the **captionTop** property. The caption is never shown as disabled.

For a **Label** control, set the **autoSize** property to **true**, to automatically resize the control to fit its caption. For a **GroupBox** control, the caption is displayed in the border that is offset from the top of the control. If there is no caption, the border is drawn as an entire rectangle.

The **caption** property of the **JadeDockBase** class determines the caption of the floating form parent of the control. The **caption** property is not used if the control is not floating.

The default value for the **JadeDockBar** and **JadeDockContainer** controls is a null string (**""**).

captionHeight

Type: Integer

Availability: Read or write at any time

The **captionHeight** property of the **JadeMask** class contains the height of the caption region of the control. The default value is zero (**0**).

Caption alignment (word wrapping) occurs within the rectangle defined by the **captionLeft**, **captionTop**, **captionHeight**, and **captionWidth** properties.

If the value of the **captionHeight** property is zero (**0**), the height is the value of the **clientHeight** property less the value of the **captionTop** property.

captionLeft

Type: Integer

Availability: Read or write at any time

The **captionLeft** property of the **JadeMask** class contains the left position of the caption region of the control. The default value is zero (**0**).

Caption alignment (word wrapping) occurs within the rectangle defined by the **captionLeft**, **captionTop**, **captionHeight**, and **captionWidth** properties. If the value of the **captionWidth** property is zero (**0**), the width of the caption region is the value of the **clientWidth** property less the value of the **captionLeft** property.

captionTop

Type: Integer

Availability: Read or write at any time

The **captionTop** property of the **JadeMask** class contains the top position of the caption region of the control.

The default value is zero (**0**).

Caption alignment (word wrapping) occurs within the rectangle defined by the **captionLeft**, **captionTop**, **captionHeight**, and **captionWidth** properties. If the value of the **clientHeight** property is zero (**0**), the height is the value of the **clientHeight** property less the value of the **captionTop** property.

captionWidth

Type: Integer

Availability: Read or write at any time

The **captionWidth** property of the **JadeMask** class contains the width of the caption region of the control. The default value is zero (**0**).

Caption alignment (word wrapping) occurs within the rectangle defined by the **captionLeft**, **captionTop**, **captionHeight**, and **captionWidth** properties.

If the value of the **captionWidth** property is zero (**0**), the width of the caption region is the value of the **clientWidth** property less the value of the **captionLeft** property.

case

Type: Integer

Availability: Read or write at any time

The **case** property determines the automatic case conversion of text entered into a **TextBox** control.

The settings of the **case** property are listed in the following table.

TextBox Class Constant	Value	Description
Case_None	0	None (the default). No conversion is performed.
Case_Lower	1	Convert to lowercase. All uppercase characters are converted to lowercase as they are entered.
Case_Upper	2	Convert to uppercase. All lowercase characters are converted to uppercase as they are entered.
Case_UpperFirst	3	Uppercase first character. The text box accepts only an alphabetic character as the first character of the text box, and it is automatically converted to uppercase.
Case_LowerFirst	4	Lowercase first character. The text box accepts only an alphabetic character as the first character of the text box, and it is automatically converted to lowercase.

If the **case** property is set to any value other than the default value of **Case_None** (**0**), the **dataType** property is set to the default value of **DataType_AlphaNumeric** (**0**) and the **decimals** property is set to zero (**0**); that is, no exception is raised.

cellControl

Type: Control

Availability: Read or write at run time only

The **cellControl** property of the **Table** control allows control over the input and display within the table by defining a user-supplied control that is placed over the cell by default when that cell becomes current. This control can be of any **Control** type. The control receives all of the events for that control and it can be manipulated by your JADE logic, as required.

You can set this property for a sheet, row, column, or cell of the table.

Each control that is assigned by using this **cellControl** property is made a child of the table and is displayed or hidden, as required, as the current cell is changed. Changing the parent of that control away from the table clears all **cellControl** property values referring to that control.

The code fragment in the following example shows the use of the **cellControl** property.

```
table1.accessCell(table1.row, table1.column).cellControl := myTextBox;
```

When the table has focus and a cell with an effective **cellControl** value becomes the current cell or the cell position or size changes, the following actions are performed:

1. The **cellInputReady** event method is called, allowing the control to initialize itself for that cell. This event passes the position and size of the cell where the control will be placed when returning from the event call. Your JADE logic can change these values, if required.

2. When returning from the **cellInputReady** event method, the cell is positioned and resized as indicated.

3. The control is then made visible.

4. The control receives focus if it is enabled and is allowed focus.

It is the responsibility of the control to update the contents of the cell with any change. Unpredictable results may occur if the control is repositioned or resized outside the **cellInputReady** event method.

Use of this **cellControl** property is exclusive to the **inputType** property. If both properties have an effective value for a cell, the **cellControl** property takes precedence. If the **cellControl** property is not set and the **inputType** property is set to **InputType_TextBox** or **InputType_ComboBox**, JADE creates a control of the type specified in the **inputType** property, which you can then access.

To implement the handling of a **TextBox** control to achieve the same functionality as the default **Table** class **InputType_TextBox** value, only the following is required.

- In the **change** event for the text box:

  ```
  table1.text := textbox1.text
  ```

- In the **cellInputReady** event for the table:

  ```
  textbox1.text := table1.text
  ```

To implement the handling of a **ComboBox** control to achieve the same functionality as the default **Table** class **InputType_ComboBox** value, only the following is required when the combo box entries are already loaded.

- In the **click** event for the combo box:

```
        table1.text := combo1.text
```

- In the **cellInputReady** event for the table:

```
        combo1.text := table1.text
```

The only other actions that are performed by the **Table** class parent of a **cellControl** property are:

- Handling the **tabKey** property.
- Handling arrow keys in the **TextBox** control. If an arrow key would logically move off the displayed text, the arrow key is processed as a table cell movement key (that is, left, up, right, or down).
- Handling the Page Up and Page Down keys in the **TextBox** or **JadeEditMask** control.
- The control is hidden if focus moves to another control.
- Tabbing to the next control using the tab key while the **cellControl** has focus tabs away from the **Table** control.
- If the control has no specific font set, the font of the current cell is used.

For details about automatically controlling a **ComboBox** or **TextBox** control assigned to a **Table** control as a **cellControl** property (for example, when performance is an issue when running in JADE thin client mode over a slow link), see the **Control** class **automaticCellControl** property.

clientHeight

Type: Integer

Availability: Read or write at run time only

The **clientHeight** property of the **Form** class contains the height of the client area of a form in pixels. The client area of a form is the area inside the borders, caption, menu bar, or scroll bars where controls can be placed. The position of child controls is relative to the top left of this client area.

This property can also be used to set the height of the client area of a form in pixels, by changing the property. Changing this property resizes the form so that the client area sizes match the specified value.

The **clientHeight** property can be useful when using graphical methods and changing the **scaleMode** property of a form.

The value of the **clientHeight** property is the same as that of the **scaleHeight** property when the **scaleMode** property value of a form is pixels.

The code fragment in the following example shows the use of the **clientHeight** property.

```
    mktForm.height := self.clientHeight - self.frame1.height;
```

See also the **clientHeight** method for controls.

clientWidth

Type: Integer

Availability: Read or write at run time only

The **clientWidth** property of the **Form** class contains the width of the client area of a form in pixels. The client area of a form is the area inside the borders, caption, menu bar, or scroll bars where controls can be placed.

The position of child controls is relative to the top left of this client area.

This property can also be used to set the width of the client area of a form in pixels, by changing the value of this property. Changing this property resizes the form so that the client area sizes match the specified value. The **clientWidth** property can be useful when using graphical methods and changing the **scaleMode** property of a form.

The value of the **clientWidth** property is the same as that of the **scaleWidth** property when the **scaleMode** property value of a form is pixels. See also the **clientWidth** method for controls.

clipControls

Type: Boolean

Availability: Read or write at any time

The **clipControls** property of the **Form** class specifies whether the Windows environment creates a clipping region that excludes controls contained by the object.

Setting this property to **true** means:

- Graphical methods for the form or control cannot draw over child controls.
- When the form or control is repainted, the area occupied by the child controls is clipped out of the paint process; that is, the child control is not affected by the parent paint and is painted only if it is actually required.

  This can result in less repaint "flash", because controls are not erased and repainted as often as when the **clipControls** property value is **false**.

This is a logical property for the **Form** class and **BaseControl**, **GroupBox**, **Frame**, **Picture**, and **StatusLine** controls.

The **clipControls** property settings are listed in the following table.

Value	A clipping region is…
true	Created around child controls of the form or control before a **paint** event (the default).
false	Not created around child controls before a **paint** event. Complex forms usually load and repaint faster.

Clipping is the process of determining which parts of a form or container (for example, a **Frame** control) are painted when the form is displayed. An outline of the form and controls is created in memory. The Windows environment uses this outline to paint some parts, such as the background, without affecting other parts; for example, the contents of a text box.

Tip As the clipping region is created in memory, setting the **clipControls** property to **false** can reduce the time that is needed to paint or repaint a form.

code

Type: Binary

Availability: Run time only

The **code** property of the **WebJavaApplet** class contains the compiled Java applet that is to be inserted into the generated HTML.

This code is relative to the base URL for the applet that is specified in the **codebase** property; it cannot be absolute.

Some Web browsers display a message on the status line if the applet is not available.

codebase

Type: String[128]

Availability: Run time only

The **codebase** property of the **WebJavaApplet** class contains the base URL of the Java applet that is to be inserted into the generated HTML. This base URL is the directory that contains the Java applet code.

If this property is set to null (**""**), the URL of the Web page is used.

column

Type: Integer

Availability: Read or write at run time only

The **column** property of the **Table** class contains the current column on the current sheet of a table control.

The **column**, **row**, and **sheet** properties define the current cell when accessing certain properties within the table control; for example, the **inputType**, **text**, **picture**, and **selected** properties of a cell.

If the current cell is visible on the table and the table control has focus, the cell has a focus rectangle painted on it. Column access is **1**-relative.

Changing the value of the **column** property does not cause the table to be repainted.

The following examples show the use of the **column** property.

```
while counter >= 1 do
    table1.column   := counter;
    table1.selected := true;
    counter         := counter - 1;
endwhile;

cdtable_keyDown(table:   Table input;
                keyCode: Integer io;
                shift:   Integer) updating;
vars
    cd    : CD;
    count : Integer;
begin
    if keyCode = J_key_Stop and table.row > 1 and table.column = 2
```

```
                 and table.itemObject <> null then
            cd := table.itemObject.CD;
            count := app.msgBox("Delete cd " & cd.title & "?", "Confirm Delete",
                                MsgBox_Yes_No);
            if count = MsgBox_Return_Yes then
                beginTransaction;
                    app.myCDList.remove(cd);
                    cd.trackList.purge;
                    delete cd;
                commitTransaction;
                table.deleteRow(table.row);
            endif;
        endif;
    end;
```

columns

Type: Integer

Availability: Read or write at run time only

The **columns** property contains the number of columns on the current sheet of a **Table** control.

Increasing columns adds empty columns to that existing sheet. Decreasing columns deletes excess columns, discarding any existing data in those columns.

Changing the **columns** property value to zero (**0**) empties the sheet of any data.

Note Changing the value of the **columns** property can affect the current values of the **column**, **row**, **topRow**, and **leftColumn** properties.

To delete a column, use the **deleteColumn** method.

You can assign a maximum of 16,000 columns to a **Table** control. However, depending on the number of rows that are also assigned to the table, the amount of memory required to handle a large number of columns limits the number of columns that can be handled, in practice.

The following examples show the use of the **columns** property.

```
load() updating;
begin
    table1.sheetCaption := "FirstTable";
    table1.columns      := 5;      // sets number of table columns
    table1.rows         := 20;
    table1.text         := 'A' & Tab & 'B' & Tab & 'C' & Tab & 'D' & Tab
                           & 'E' & Tab;
end;

selectDay(date: Date) updating;
vars
    r, c : Integer;
begin
    accessMode := 1;
    r := 2;
    while r <= rows do
        row := r;
```

```
            c := 1;
            while c <= columns do
                column := c;
                if text = date.day.String then
                    foreColor := calendar.backColor;
                    backColor := calendar.foreColor;
                else
                    foreColor := calendar.foreColor;
                    backColor := calendar.getSystemColor(Color_BtnFace);
                endif;
                c := c + 1;
            endwhile;
            r := r + 1;
        endwhile;
        row := 1;
        column := 1;
    end;
```

columnVisible

Type: Boolean array

Availability: Read or write at run time only

The **columnVisible** property enables a column of a **Table** control to be displayed or hidden, or the visibility status to be obtained. Setting the visible status of a column causes a repaint.

The following example shows the use of the **columnVisible** property.

```
tableGroup_dragDrop(groupbox: GroupBox input; win: Window input; x: Real;
                    y: Real) updating;
// If a table column is dropped onto the tableGroup, make it invisible
// and toggle the corresponding menuItem entry
vars
    ix : Integer;
    mi : MenuItem;
begin
    if win.name = 'theTable' then
        theTable.columnVisible[theTable.column] := false;
        ix := 0;
        while ix < theTable.columns do
            mi := mTableColumns.getMenuItem(ix);
            if mi.caption = theTable.text then
                mi.checked := false;
                return;
            endif;
            ix := ix + 1;
        endwhile;
        win.dragMode := 0;
    endif;
end;
```

columnWidth

Type: Integer array

Availability: Read or write at run time only

The **columnWidth** property enables the size of a column of a **Table** control to be accessed. The width of the column cannot exceed the maximum width of the table. The default value is **50** pixels.

The user can also change the **columnWidth** property by dragging a fixed-column boundary with the mouse.

The **columnWidth** property contains an array of integer values with the same number of items as the **columns** property.

Setting the **columnWidth** value of a column causes a repaint. Setting the **columnWidth** property of a column to zero (**0**) causes it to use the default column width; that is, 50 pixels.

Use the **columnVisible** property to hide a column rather than change its width.

The code fragment in the following example shows the use of the **columnWidth** property.

```
count := tbl.columns - 1;
while count > 0 do
    tbl.columnWidth[count] := (tbl.width / tbl.columns).rounded;
    count := count - 1;
endwhile;
tbl.columnWidth[tbl.columns] := tbl.width.Integer - (tbl.columns - 1)
               * (tbl.width / tbl.columns).rounded;
```

comboIndex

Type: Integer

Availability: Read or write at run time only (for a cell with **inputType** set to **3** only)

The **comboIndex** property of the **Table** class contains the index of a combo box in a cell of a **Table** control.

This property applies only to a cell that has the **inputType** property set to **InputType_ComboBox** (**3**). The current **sheet**, **row**, and **column** properties determine the cell to which the reference applies.

When such a cell is selected, a combo box filled with list entries replaces the displayed text. Using the **comboList** property sets this list. The combo box entry that is selected is determined by the setting of **comboIndex** property (**1**-relative).

When the user selects a combo box entry, the value of the **comboIndex** property is set to the index of the selected entry. The value of the text for the cell is set to the text of the selected list entry. A **change** event is then caused. If logic sets the text of the cell, the text must match an entry in the combo box list.

The default value is **-1** (that is, no entry is selected).

This property is not affected by the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property, as the property always applies to the current cell.

comboList

Type: String

Availability: Read or write at run time only (for a cell with **inputType** set only to **3**, for a combo box)

The **comboList** property contains the list entries displayed in a combo box in the sheet, column, row, or cell of a **Table** control. This property applies only to a cell that has an effective **inputType** property set to **InputType_ ComboBox** (**3**).

The current **sheet**, **row**, and **column** properties determine the cell to which the reference applies. This property is affected by the rules that apply to the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property. When such a cell is selected, a combo box filled with list entries replaces the displayed text. The combo box entry that is selected is determined by the setting of the **comboIndex** property (**1**-relative).

The value of the **comboList** property is set by using a single string, with list entries separated by a tab character; for example:

```
    table1.comboList := "one" & Tab & "two" & Tab & "three";
```

When the user selects a combo box entry, the value of the **comboIndex** property is set to the index of the selected entry. The value of the text for the cell is set to the text of the selected list entry. A **change** event is then caused.

If logic sets the text of the cell, the text must match an entry in the combo box list.

The default value is an empty string (**null**).

contextMenuOptions

Type: Integer

Availability: Read or write at any time

The **contextMenuOptions** property of the **JadeRichText** control contains the menu options that are visible when the inbuilt popup menu of the control is displayed. The context (or popup) menu enables users to perform basic edit operations and it supports basic paragraph formatting such as setting bulleting, fonts, and indents.

The **contextMenuOptions** property can contain one or more of the values listed in the following table.

Constant	Value	Comment
MenuOption_All	#7FFFFFFF	The default value, which displays all context menu items
MenuOption_Bullet	#00008000	
MenuOption_Copy	#00000008	
MenuOption_Custom	#80000000	Customized context menu is displayed when a right-click action generates the **contextMenu** event
MenuOption_Cut	#00000004	
MenuOption_Find	#00000020	
MenuOption_Font	#00000080	
MenuOption_InsertObject	#00010000	
MenuOption_InsertTable	#00020000	

Constant	Value	Comment
MenuOption_None	0	No context menu is displayed
MenuOption_Object	#00800000	
MenuOption_ObjectProperties	#00040000	
MenuOption_PageSetup	#00000200	
MenuOption_Paragraph	#00000100	
MenuOption_Paste	#00000010	
MenuOption_Print	#00000400	
MenuOption_Redo	#00000001	
MenuOption_Replace	#00000040	
MenuOption_SepCutCopyPaste	#02000000	Separator that is displayed after the **Cut**, **Copy**, and **Paste** context menu items
MenuOption_SepFindReplace	#04000000	Separator that is displayed after the **Find** and **Replace** context menu items
MenuOption_SepFontParaBullet	#08000000	Separator that is displayed after the **Font**, **Paragraph**, and **Bullet Style** context menu items
MenuOption_SepInsert	#10000000	Separator that is displayed after the **Insert Object** and **Insert Table** context menu items
MenuOption_SepPrint	#20000000	Separator that is displayed after the **Page Setup** and **Print** context menu items
MenuOption_SepRedoUndo	#01000000	Separator that is displayed after the **Redo** and **Undo** context menu items
MenuOption_Undo	#00000002	

You can specify one or more values, by separating each value with a plus symbol (+) or a minus symbol (-).

The first of the following examples displays only the **Cut**, **Copy**, and **Paste** menu items on the context menu. The second example displays all menu items except for the **Insert Object** and **Insert Table** menu items.

```
rtfRichText.contextMenuOptions := JadeRichText.MenuOption_Cut +
                                  MenuOption_Copy + MenuOption_Paste;

rtfRichText.contextMenuOptions := JadeRichText.MenuOption_All -
                                  MenuOption_InsertObject - MenuOption_InsertTable;
```

controlBox

Type: Boolean

Availability: Read or write at any time

The **controlBox** property of the **Form** class specifies whether a Control-Menu icon is displayed on a form at run time. (The Control-Menu icon at the top left of a form takes the icon image of the form. For example, the Control-Menu icon for JADE development environment forms is the JADE logo and in Microsoft Word it is displayed as the Word icon image.)

The **controlBox** property settings are listed in the following table.

Value	Description
true	Displays the Control-Menu icon (the default).
false	Removes the Control-Menu icon.

Setting the **controlBox** property to **true** causes a form with the **borderStyle** property set to **BorderStyle_None** (**0**) to change to a **borderStyle** value of **BorderStyle_Single** (**1**). Additionally, setting the **controlBox** property to **true** causes the form to be displayed with a caption box if it did not have one. The setting of this property after the window has been created is ignored when an MDI child form is maximized.

Both modal and non-modal windows can include a Control-Menu icon.

The commands available at run time depend on the settings for related properties. For example, setting the **maxButton** and **minButton** properties to **false** disables the **Maximize** and **Minimize** commands on the Control-Menu but the **Move** and **Close** commands remain available.

Note Without a Control-Menu icon on a form or an MDI menu displayed by right-clicking in the caption of an MDI child form, the user has no way of closing a form. The form can be closed only when the **unloadForm** method is called from logic.

controlChildren

Type: ControlArray

Availability: Read at run time only

The **controlChildren** property of the **Window** class contains a reference to an array of all of the immediate children of the window (form or control); that is, the window is a direct parent of each control in the array. The collection is in no particular order.

The array is changed if the z-order of a control is changed by logic or if the parent of a control is changed.

Applies to Version: 2016.0.01 and higher

createRegionFromMask

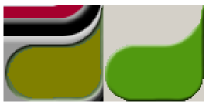
Type: Boolean

Availability: Read or write at any time

The **createRegionFromMask** property of the **JadeMask** class specifies whether a region is created around the mask picture on the control.

This property enables you to display an irregular shaped control on a form by hiding the display of the area within the rectangle that does not contain the mask picture. By default, a region is not created from the mask; that is, this property is set to **false** by default.

The first example in the following image shows a JADE mask control on a form with no region created and the second example shows a region created from the mask picture on the same control (that is, the property is set to **true**).



currentColumn

Type: Integer

Availability: Read-only at run time

The **currentColumn** property of the **JadeTextEdit** control contains the column within the current line at which the caret is located. The first column is **1**.

currentLine

Type: Integer

Availability: Read or write at run time only

The **currentLine** property of the **JadeTextEdit** control contains the number of the current line containing the caret. The first line is **1**.

When you set this property, the caret moves to the first character of the specified line and it scrolls into view. A value less than **1** means the first line of text. A value greater than or equal to the value returned by the **lineCount** method means the last line of text.

currentPosition

Type: Integer

Availability: Read or write at run time only

The **currentPosition** property of the **JadeTextEdit** control contains the current caret position in the editor as a zero-based offset from the start of the text.

When you set this property, the caret moves to the specified character and scrolls into view. A value less than zero (**0**) means the first character of text. A value greater than the result of the **getTextLength** method means the position following the last character of text. The selection is cleared.

dataType

Type: Integer

Availability: Read or write at any time

The **dataType** property of a **TextBox** control contains the type of data that can be entered in the text box by the user.

The settings for the **dataType** property are listed in the following table.

TextBox Class Constant	Value	Description
DataType_AlphaNumeric	0	Any character that can be entered (the default).
DataType_Numeric	1	Combined with the **decimals** property, this property ensures that the user can enter a valid decimal value only.
DataType_SignedNumeric	2	Same as **DataType_Numeric** (**1**), except that a leading negative sign is also permitted.
DataType_Currency	3	The currency text is in the form defined by the currency format of the locale. By default, the **decimals** property is set to **-1**, indicating that the number of decimal places for currency defined by the current locale is used. To manually control how many decimal places are allowed, set the **decimals** property to a value other than **-1**. The text can include the currency symbol, negative sign, thousands separators, and decimal places; for example, **$123,456.34**.
DataType_ShortDate	4	The date text is in the form defined by the short date format of the locale. For example, under English (New Zealand), the default format is *d/MM/yyyy* (for example, 7/08/2010). Note that when altering the text, deleting characters that would make the date invalid is not permitted; for example, deleting the day **7** or a **/** character. To make such changes, select the character to be replaced and then enter the new value, or delete the date from the right up to the required position. Incomplete dates are allowed, where the date text entered so far is valid.
DataType_LongDate	5	The date text is in the form defined for the locale's long date format. For example, under English (New Zealand), the default format is *dddd*, *d MMMM yyyy* (for example, Sunday, 7 March 2010). Note that when altering the text, deleting characters that would make the date invalid is not permitted; for example, deleting the day **7** or a **/** character. To make such changes, select the character to be replaced and then enter the new value, or delete the date from the right up to the required position. Incomplete dates are allowed, where the date text entered so far is valid. In addition, when the day name is displayed, that day must match the day for the date. To alter the date, it will most likely require the day name to be removed before the date can be achieved. Note that the month names can be entered in the full or short format.
DataType_Time	6	The time text is in the form defined for the locale's long time format. For example, under English (New Zealand), the default format is *h:mm:ss tt* (for example, 5:29:02 p.m.). Note that when altering the text, deleting characters that would make the date invalid is not permitted; for example, deleting the hour **7** or a **:** character. To make such changes, select the character to be replaced and then enter the new value, or delete the date from the right up to the required position. Incomplete times are allowed, where the time text entered so far is valid.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is not defined or it is set to **false**, no local client overrides are used.

The numeric text is validated as each character is entered. If the result of the assignment would result in an invalid numeric as defined by the **dataType** and **decimals** properties, the entered character is rejected.

Setting the **dataType** property for a text box control to a numeric type is rejected if the current text in the text box does not conform to the rules defined by the current **dataType**, **decimals**, and **maxLength** properties.

If the **case** property is set to any value other than the default value of **Case_None** (**0**), the **dataType** property is set to the default value of **DataType_AlphaNumeric** (**0**) and the **decimals** property is set to zero (**0**); that is, no exception is raised.

decimals

Type: Integer

Availability: Read or write at any time

The **decimals** property is used in conjunction with the **dataType** property to indicate whether the text in a **TextBox** control is a **Decimal** primitive type numeric.

If the **dataType** property is set to **DataType_Currency** (**3**), setting the **decimals** property to **-1** indicates that the number of decimal places for currency defined by the current locale is used. To manually control how many decimal places are allowed, set the **decimals** property to a value other than **-1**.

If the **dataType** property is set to **DataType_Numeric** (**1**) or **DataType_SignedNumeric** (**2**), the **decimals** property indicates the maximum number of decimal places that the numeric value can have. The numeric text is validated as each character is entered. If the result of the assignment results in an invalid numeric as defined by the **dataType** property, the character that is entered is rejected.

If the **case** property is set to any value other than the default value of **Case_None** (**0**), the **dataType** property is set to the default value of **DataType_AlphaNumeric** (**0**) and the **decimals** property is set to zero (**0**); that is, no exception is raised.

Use the **decimals** property for the **Table** class to specify that a cell controlled by the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property can accept decimal input when the value of the **inputType** property is set to **InputType_TextNumeric** (**4**) or **InputType_SignedNumeric** (**6**).

Specify a value greater than zero (**0**) but less than the value of the **maxLength** property to specify the number of decimal places that numeric text in a table cell can have. Setting the **decimals** property to **-1** indicates that decimals are allowed but no limit is imposed on the number of decimal places.

The value of the **decimals** property (if non-zero) cannot equal or exceed the value of the **maxLength** property if that is non-zero.

Note Setting the value of the **decimals** property is rejected if the current text in the text box does not conform to the rules defined by the current value of the **dataType** (for text box controls), **inputType** (for **Table** controls), **decimals**, and **maxLength** properties.

default

Type: Boolean

Availability: Read or write at any time

The **default** property applies to **Button** and **JadeMask** controls. If the **default** property is set to **true**, that button or mask control is marked as the default. If this button or mask control has focus or if a control that is not a button or mask control has focus, a black border is drawn around the button or mask control to indicate that it is the default button or mask control.

When the user presses the Enter key, if the focus is not on a button or mask control and there is an enabled visible button or mask control on the same form with the **default** property set, the focus is transferred to that button or mask control and a **click** event is caused on that **Button** or **JadeMask** control, regardless of the control that had the focus on the form.

If another button or mask control has the focus, it processes the Enter key and generates a **click** event. The default value for any button or mask control is **false**.

It is your responsibility to ensure that only one button or mask control has the **default** property set to **true**. If not, the first such enabled visible button or mask control found with that status receives the event. You must write logic to respond to the event.

defaultLineHeight

Type: Integer

Availability: Read or write at any time

The **defaultLineHeight** property of the **ComboBox** class and the **ListBox** class specifies the default height of lines in a combo box or list box, independent of the font. This property represents pixels and defaults to zero (**0**), indicating that the height of the line is determined by the font selected for the combo box or list box.

If the value of the property is greater than the height of the font in the combo box or list box, the value of the **defaultLineHeight** property is used to determine the line height of the list. If the value is less than the height of the font, the list line height is determined from the combo box or list box font.

This property for a combo box control, implemented from version 2020.0.01, has no impact on the size of the combo box itself; only the height of the list items when the list is displayed. To assist touchscreen users, it enables you to specify the default line height of a list in a combo box (for example, **cbt.defaultLineHeight := 25;**).

The code fragment in the following example sets the height of the lines in a list box to two pixels higher than the font size.

```
if listbox1.defaultLineHeight = 0 then
    listbox1.defaultLineHeight := listbox1.getLineHeight() + 2;
endif;
```

If the value of the **defaultLineHeight** property is greater than zero (**0**), the **ListBox** class **getLineHeight** method returns that value. See also the **Table** class **defaultRowHeight** property.

defaultRowHeight

Type: Integer

Availability: Read or write at any time

The **defaultRowHeight** property of the **Table** class specifies the default height of rows in a table, independent of the font. This property represents pixels and defaults to zero (**0**), indicating that the height of the row is determined by the font selected for the table.

If the value of this property is non-zero, the value is used to determine the height of all rows in the table that are not specifically set.

If the height of a row is set by logic, the specified value is used. If the height of a table row is not set by logic, if the value of the **defaultRowHeight** property is:

- Greater than the calculated required row height, the value of the **defaultRowHeight** property is used as the row height
- Less than the calculated required row height, the calculated row height value is used

Note This default height is also used as the height of any cell that has a value set for the **Table** class **inputType** property.

The code fragment in the following example shows the use of the **defaultRowHeight** property.

```
if table1.defaultRowHeight = 0 then
    table1.defaultRowHeight := table1.rowHeight[1] + 2;
endif;
```

See also the **ListBox** class **defaultLineHeight** property.

description

Type: String

Availability: Read or write at any time

The **description** property of the **Window** class contains a textual description of the object of the window. The description can be in the range 0 through 32,767 characters. This description is for documentation only and is not automatically displayed.

Any change to the value at run time is not retained after the form on which the control (or the form itself) is unloaded.

The **description** property for the **Form** or **Control** class defined for a Web page displays bubble help if the value of the **bubbleHelp** property is null (**""**) and the control has a picture (for example, a **Button** control).

disabledForeColor

Type: Integer

Availability: Read or write at any time

The **pictureMask** property of the **JadeMask** class determines the color of disabled displayed text in a **JadeMask** control unless the value of this property is zero (**0**). By default, the foreground color is black.

If the value of this property is zero (**0**), the value the **foreColor** property is used to display disabled text if the control has a current picture displayed. If the control has no pictures, it behaves like a button and a **disabledForeColor** property value of zero (**0**) means disabled text is disabled as it is for a button.

disableEvents

Type: Boolean

Availability: Read or write at any time

The **disableEvents** property of the **Window** class specifies whether all events associated with a form or control are currently disabled (ignored).

User logic associated with an event is not executed when the events are disabled.

The settings of the **disableEvents** property are listed in the following table.

Value	Description
false	All events are enabled (the default)
true	Disables (ignores) the events

Setting the **disableEvents** property to **true** for a form disables all events for the form and any of its controls. If a form has the **disableEvents** property set to **true**, requests to unload the form using a click on the close menu, a button, or the Alt+F4 keystrokes will be rejected.

In addition, if the form is an MDI child, the user request of the unloading of the MDI frame using these same mechanisms will also be rejected. If the value of the **disableEvents** property remains **true** and the **unloadForm** method of the **Form** class is called, the **queryUnload** and **unload** events are not called. (This handling of the **disableEvents** property is intentional.)

Use this property when events are to be ignored during some operation; for example, when performing a process that takes some time and where the user may get impatient and attempt to click the **Go** button again. Its advantage over using the **enabled** property is that it does not change the appearance of the form or the controls.

disableReason

Type: String

Availability: Read or write at any time

The **disableReason** property of the **Window** class contains the help text that is displayed providing a reason why a control or menu is disabled.

JADE does not use this property. It is your responsibility to display the text, as appropriate.

displayAsIcon

Type: Boolean

Default Value: False

The **displayAsIcon** property of the **OleControl** class specifies whether only the icon of the OLE application is displayed.

displayHotKey

Type: Boolean

Availability: Read or write at run time only

The display of text in each cell of a table control displays ampersand (**&**) characters by default.

Set the **displayHotKey** property of the **Table** class to **true** so that ampersand (**&**) characters are displayed as though the text represents a hot key description and the character following an ampersand (**&**) character is underlined.

The default value is **false**.

Tip This property is used by the JADE Menu Design facility, which displays a simulated menu.

dragCursor

Type: Binary

Availability: Read or write at any time

The **dragCursor** property of the **Window** class contains a specific cursor image for display during the drag process, instead of the standard drag mouse cursor that is displayed by default when a window is dragged.

The assigned image must be either a Windows cursor or icon.

This image is displayed only when the form or control to which it is assigned has the **dragMode** property set to **DragMode_Drag** (**1**).

When the window dragging process is terminated, the mouse cursor is reset to an arrow.

dragMode

Type: Integer

Availability: Read or write at any time

The **dragMode** property of the **Window** class contains the drag mode for a form or control. Any form or control can be dragged and dropped on any other form or control belonging to the application.

This process is initiated by setting the **dragMode** property of a form or control to **DragMode_Drag** (**1**). A default or user-assigned drag cursor is displayed while the drag process is in progress. This process is usually initiated by logic when the user presses the left mouse button over the form or control that is to be dragged. Any form or control of the application over which the mouse cursor moves receives a **dragOver** event.

The settings for the **dragMode** property are listed in the following table.

Window Class Constant	Value	Description
DragMode_None	0	Drag mode not in effect
DragMode_Drag	1	Drag mode
DragMode_Drop	2	Causes the drag process to terminate and generates a **dragDrop** event

The form or control of the application over which the drag process is terminated receives a **dragDrop** event. The dragging process is terminated when a left **mouseUp** or **mouseLeave** event is received or by logic setting the value of the **dragMode** property of the dragged form or control to **DragMode_Drop** (**2**).

Setting the value of the **dragMode** property to **DragMode_None** (**0**) also aborts the drag process and does not issue a **dragDrop** event. While dragging is in progress, no normal mouse events are available for logic processing.

Note The **Application** class **showBubbleHelp** property is ignored when dragging is in progress.

The following example shows the use of the **dragMode** property.

```
table1_mouseDown(table: Table input; button: Integer;
                 shift: Integer; x: Real; y: Real) updating;
begin
    if button = 1 then
        dragMode := DragMode_Drag;
    endif;
    if button = 2 then
        popupMenu(menuItemAction, x.Integer, y.Integer);
    endif;
    selectedColumn := table.column;
end;
```

For the **ListBox** control, the **dragListIndex** method can also be interrogated during the **dragOver** and **dragDrop** events to determine which entry of the list box was involved.

For the **Table** control, the **dragSheet**, **dragColumn**, and **dragRow** methods can also be interrogated during the **dragOver** and **dragDrop** events to determine which sheet and cell of the table was involved.

drawGrip

Type: Integer

Availability: Read or write at any time

The **drawGrip** property of the **JadeDockBase** class indicates whether a grip bar is drawn in the border area of the control. This bar is intended as a place on the control in which the user can click the mouse for dragging purposes.

The **drawGrip** property settings are listed in the following table.

JadeDockBase Class Constant	Integer Value	Description
DrawGripBar_Double	2	A double grip bar is drawn
DrawGripBar_None	0	No grip is drawn
DrawGripBar_Single	1	A single grip bar is drawn

The grip bar adds additional spacing within the border area of the control, and it is centered in the area defined by the size of the grip bar and the extra border space defined by the **borderWidthLeft** or **borderHeightTop** property, depending on how the control is aligned.

If the control is aligned left or right, the grip bar is drawn at the top of the control. If the control has no alignment but its parent is aligned left or right, the grip bar is also drawn at the top of the control. If neither of these is the case, the grip bar is drawn at the left of the control.

No border area (including the grip bar) is drawn for the control when it is the child of a floating form.

The default value for the **JadeDockBar** control is **DrawGripBar_Double** (**2**) and the default value for the **JadeDockContainer** control is **DrawGripBar_None** (**0**).

dropDown

Type: Integer

Setting the **dropDown** property indicates that a table need occupy only the space required for one row and yet still provide all the features of an expanded display.

If this property value is non-zero for a **Table** control, the table alters its size so that only the current row is displayed when it loses the focus. The current **left**, **top**, and **width** values are retained.

The settings of the **dropDown** property are listed in the following table.

Table Class Constant	Value	Description
DropDown_None	0	No close up occurs
DropDown_Click	1	Close up on lost focus, open up only when clicked
DropDown_Auto	2	Close up on lost focus, open up when focus is received

edgeColor

Type: Integer[4]

Availability: Read or write at any time

The **edgeColor** property of the **JadeTextEdit** control contains the color of the indicator line or background, depending on the value of the **edgeMode** property, that is used to show that a line has exceeded the length specified by the value of the **edgeColumn** property.

The default value is zero (**0**); that is, black.

edgeColumn

Type: Integer[4]

Availability: Read or write at any time

The **edgeColumn** property of the **JadeTextEdit** control contains the number of the column at which the long linemark indicator is displayed. The default value is zero (**0**).

The valid range is zero (**0**) through **200**.

edgeMode

Type: Integer

Availability: Read or write at any time

The **edgeMode** property of the **JadeTextEdit** control contains the mode that is used to display long lines in the text editor.

The **JadeTextEdit** class constants that specify the edge mode are listed in the following table.

Class Constant	Value	Description
SC_EDGE_BACKGROUND	2	The background color of characters after the column limit is changed to the value of the **edgeColor** property.
SC_EDGE_LINE	1	A vertical line is drawn at the column number specified in the **edgeColumn** property. The line is drawn at a position based on the width of a space character in the default style. This edge mode may not work very well if your styles use proportional fonts, if your styles have varied font sizes, or you use a mixture of bold, italic, and normal text font attributes.
SC_EDGE_NONE	0	Long lines are not marked (the default value).

editMask

Type: String

Availability: Read or write at any time

The **editMask** property of the **Table** class sets the mask (using the **accessMode** property) used for input for a table cell, row, column, or sheet when the value of the **Table** class **inputType** property is set to **InputType_EditMask** (**7**).

When a table cell, row, column, or sheet is defined as a **JadeEditMask** control by setting the input type to **InputType_EditMask** (**7**), the input control acts like a text box for input by default.

Notes Using an edit mask control causes the column width for the effective cell to be enlarged to fit the **JadeEditMask** control that is to be displayed unless the value of the **columnWidth** property has been set by logic.

Format the text of a cell that has an **inputType** property value of **InputType_EditMask** so that it obeys the rules implied by the edit mask. The text of the edit mask control for a cell is set and updated by using the **JadeEditMask** class **text** property.

The **Control** class **automaticCellControl** property has meaning for the edit mask input type.

If the text of a cell does not match the rules implied by the edit mask, an exception is raised when the cell is activated for user input.

enabled

Type: Boolean

Availability: Read or write at any time

The **enabled** property of the **Window** class specifies whether the form, control, or menu can respond to user-generated events. It can also result in the physical appearance of a control being changed; for example, grayed (disabled). The **enabled** property settings are listed in the following table.

Value	Description
true	Enables the object to respond to events (the default).
false	Prevents the object from responding to events. For controls on Web pages, the object is not displayed.

This property allows forms and controls to be enabled or disabled at run time.

If a disabled cell, row, or column has a specified value of the **foreColor** property, the text in a disabled cell is displayed using that color. The text in a disabled cell is displayed using the Windows disabled text color if the **foreColor** property of the cell is not specifically set using the **foreColor** property values of the cell, row, or column.

Unlike the **disableEvents** property, disabling a control can result in the appearance of the control changing. For example, the text of a **Button** control is dimmed when the button is disabled. If the control with the focus (or its parent control) is disabled, the focus is moved to the next visible and enabled control in the tab order.

The code fragment in the following example shows the use of the **enabled** property.

```
if listFaults.itemText[listFaults.listIndex] <> "" then
    bCloseFault.enabled := true;
endif;
```

endOfLineMode

Type: Integer[4]

Availability: Read or write at any time

The **endOfLineMode** property of the **JadeTextEdit** control defines the character sequence used to indicate the end of a line in the text editor. This property controls what is inserted into the text buffer when a new line is required, including a new line keystroke (for example, when Enter is pressed), new lines inserted from the clipboard, and new lines loaded from a file.

Text copied to the clipboard or to a file has end-of-line characters converted to the platform-native equivalent.

The value can be one of the **JadeTextEdit** class constants listed in the following table.

Class Constant	Integer Value	Description
SC_EOL_CR	1	Carriage return character
SC_EOL_CRLF	0	Carriage return / line feed character (the default value)
SC_EOL_LF	2	Line feed character

expandedHeight

Type: Integer

The **expandedHeight** property of the **Table** class contains the height in pixels of the table when it is expanded. The expanded height defaults to the height of the table when it was last expanded.

You can set this property only when the table is closed up.

For more details, see the **dropDown** property.

firstLineIndent

Type: Integer

Availability: Read or write at any time

The **firstLineIndent** property of the **JadeRichText** control contains the distance (in pixels) between the left edge of the first line of text in the selected paragraph and the left edge of subsequent lines in the same paragraph.

For an example of the use of this property, see "JadeRichText Control Method Example", earlier in this document.

This value can be negative. If multiple paragraphs are selected and each line has a different value, the property contains **ParagraphFormat_Undefined** (#80000000).

The default value of zero (**0**) indicates that the first line of text is not indented.

firstVisibleLine

Type: Integer[4]

Availability: Read or write at any time

The **firstVisibleLine** property of the **JadeTextEdit** control contains the number of the first line that is visible in the client area of the text editor, in the range **1** through the value of the **lineCount** property.

Setting the value outside of the range **1** through the value of the **lineCount** property results in the value of the **firstVisibleLine** property being capped to that range.

fixed3D

Type: Boolean

Availability: Read or write at any time

The **fixed3D** property of the **Table** class specifies whether a three-dimensional (3D) button image is painted on the cells in the fixed area of a table. This 3D effect assumes a **gray** value for the **backColor** property for the cell.

The **Table** control has two types of rows and columns: fixed and non-fixed. Fixed rows and columns are most often used for column and row headings. Non-fixed rows and columns are most often used for data display.

A fixed row and column is always displayed on the sheet, while non-fixed rows and columns can be scrolled out of view. Clicking on a cell in a fixed column selects all non-fixed cells in the column. Clicking on a cell in a fixed row selects all non-fixed cells in the row. Clicking on a non-fixed cell selects only that cell.

Positioning the mouse over a boundary between columns and rows in the fixed area enables the boundary line to be dragged by clicking and dragging. This changes the height and width of a column. (See also the **allowResize** property.)

The default value of the **backColor** property of a fixed cell is **gray**.

fixedColumns

Type: Integer

Availability: Read or write at any time

The **fixedColumns** property of the **Table** class contains the number of fixed columns in a table.

The default value for the **fixedColumns** property is **1**.

The **Table** control has two types of columns: fixed and non-fixed. Fixed columns are most often used for row headings. Non-fixed columns are most often used for data display. A fixed column is always displayed on the sheet, while non-fixed columns can be scrolled out of view.

Clicking a cell in a fixed column selects all non-fixed cells in the row. Clicking on a cell selects only that cell.

Positioning the mouse over a boundary between columns in the fixed area enables the boundary line to be dragged by clicking and dragging. This changes the height and width of a column. (See also the **allowResize** property.)

The default value of the **backColor** property of a fixed cell is **gray**.

Setting the **fixedColumns** property to a value greater than the defined number of columns causes the number of columns to be increased to the new number of fixed columns.

fixedRows

Type: Integer

Availability: Read or write at any time

The **fixedRows** property of the **Table** class contains the number of fixed rows in a table. The default value for the **fixedRows** property is **1**. The **Table** control has two types of rows: fixed and non-fixed.

Fixed rows are most often used for column headings. Non-fixed rows are most often used for data display. A fixed row is always displayed on the sheet, while non-fixed rows can be scrolled out of view.

Clicking a cell in a fixed row selects all non-fixed cells in the column. Clicking on a cell selects only that cell.

Positioning the mouse over a boundary between rows in the fixed area enables the boundary line to be dragged by clicking and dragging. This changes the height and width of a row. (See also the **allowResize** property.)

The default value of the **backColor** property of a fixed cell is **gray**.

Setting the **fixedRows** property to a value greater than the defined number of rows causes the number of rows to be set to the new number of fixed rows.

floatingStyle

Type: Integer

Availability: Read or write at any time

The **floatingStyle** property of the **JadeDockBase** class indicates whether the control can be floated or dragged. The **floatingStyle** property settings are listed in the following table.

JadeDockBase Class Constant	Integer Value	Comment
FloatingStyle_Close	1	
FloatingStyle_NoClose	2	Default value for the **JadeDockBar** control
FloatingStyle_None	0	Default value for the **JadeDockContainer** control

Use the **FloatingStyle_None** (**0**) value to indicate that the control is just a container that can be neither floated nor dragged.

Use the **FloatingStyle_Close** (**1**) value if the control can be floated and dragged. When a control that has this style is floated, the floating form is displayed with a **Close** button that allows the user to close the form.

If the user closes the floating form, the docking control is transferred back to being an invisible child of the original JADE form and requires JADE logic action to make the control visible to the user again.

The dock control that is being hidden receives a **JadeDockBar** or **JadeDockContainer** class **docked** event method. You can determine this situation by checking the value of the **visible** property of the control.

Use the **FloatingStyle_NoClose** (**2**) value when the control can be floated and dragged. When a control that has this style is floated, the floating form is displayed without a **Close** button and the user cannot close the form. The form is destroyed only when the control is docked back into the original form or when the original form is closed.

focusBackColor

Type: Integer

Availability: Read or write at any time

The **focusBackColor** property of the **Control** class contains the background color of a control when the control has focus or a child of the control has focus. You can use the **focusBackColor** and **focusForeColor** properties to give the user a better visual prompt as to which control has focus.

The default value of zero (**Black**) indicates that the property is always ignored when drawing the control.

JADE uses the RGB scheme for colors. The valid range for a normal RGB color is zero (**0**) through **16,777,215** (**#FFFFFF**). The high byte of a number in this range equals **0**; the lower three bytes (from least to most significant byte) determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is **128**, JADE uses the system colors, as defined in the Control Panel of the user. To determine the **Integer** value of a color from the RGB values:

```
int:= RedValue + (GreenValue * 256) + (BlueValue * 256 * 256);
```

When the value of the **focusBackColor** property is not **Black**, that property value is used instead of the value of the **backColor** property to erase the control area when the control has focus or a child of the control has focus.

Note If the control is transparent, the value of the **focusBackColor** property is not used. (The control area is not erased as part of the painting of the control.)

Although this property is defined in the **Control** class, it is not relevant to all controls. The controls that make use of this property must be capable of gaining the focus, they can be control parents, and they cannot be external controls such as .NET controls.

The controls that use the **focusBackColor** property and support definition in the JADE Painter are:

- BaseControl
- Button
- CheckBox
- ComboBox
- Folder
- Frame
- GroupBox
- Sheet
- ListBox
- OptionButton

- Picture
- StatusLine
- Table
- TextBox
- JadeMask
- JadeDockBar
- JadeDockContainer
- JadeEditMask
- JadeRichText.

In addition, the **focusBackColor** property is defined in the **JadeSkinControl** class so that you can specify it as part of a control skin on the Jade Skin Maintenance dialog. When a skin is assigned to a control that uses this property, the value is used when the control has focus or a child of the control has focus if all of the following are true.

- The value is not **Black** (**0**)
- The value of the equivalent **focusBackColor** property of the control is **Black** (**0**)
- The equivalent **backColor** value of the control is the default for the control

Note If the control or the skin of the control has a brush defined to erase the background area of the control, the effective value of the **focusBackColor** property is used instead.

Applies to Version: 2016.0.01 and higher

focusForeColor

Type: Integer

Availability: Read or write at any time

The **focusForeColor** property of the **Control** class contains the foreground color used to display text associated with a control when the control has focus or a child of the control has focus. You can use the **focusForeColor** and **focusBackColor** properties to give the user a better visual prompt as to which control has focus.

The default value of zero (**Black**) indicates that the property is always ignored when drawing the control.

JADE uses the RGB scheme for colors. The valid range for a normal RGB color is zero (**0**) through **16,777,215** (**#FFFFFF**). The high byte of a number in this range equals **0**; the lower three bytes (from least to most significant byte) determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is **128**, JADE uses the system colors, as defined in the Control Panel of the user. To determine the **Integer** value of a color from the RGB values:

```
int:= RedValue + (GreenValue * 256) + (BlueValue * 256 * 256);
```

When the value of the **focusForeColor** property is not **Black**, that property value is used instead of the value of the **foreColor** property to draw the text associated with the control when the control has focus or a child of the control has focus.

Although this property is defined in the **Control** class, it is not relevant to all controls. The controls that make use of this property must be capable of gaining the focus, they can be control parents, and they cannot be external controls such as .NET controls.

The controls that use the **focusForeColor** property and support definition in the JADE Painter are:

- BaseControl
- Button
- CheckBox
- ComboBox
- Folder
- Frame
- GroupBox
- Sheet
- ListBox
- OptionButton
- Picture
- StatusLine
- Table
- TextBox
- JadeMask
- JadeDockBar
- JadeDockContainer
- JadeEditMask
- JadeRichText.

In addition, the **focusForeColor** property is defined in the **JadeSkinControl** class so that you can specify it as part of a control skin on the Jade Skin Maintenance dialog. When a skin is assigned to a control that uses this property, the value is used when the control has focus or a child of the control has focus if all of the following are true.

- The value is not **Black** (**0**)
- The value of the equivalent **focusForeColor** property of the control is **Black** (**0**)
- The equivalent **foreColor** value of the control is the default for the control

Applies to Version: 2016.0.01 and higher

foldFlags

Type: Integer[4]

Availability: Read or write at any time

The **foldFlags** property of the **JadeTextEdit** control contains the flags that are used to control how folded section lines are marked when the **folding** property is set to **true**. The fold point highlight is a line is drawn edge-to-edge in the text area. The fold flag has no effect on what is drawn in the fold margin.

The default value of this property for the **JadeTextEdit** class is zero (**0**) and for the **JadeEditor** class it is **16**.

The fold flag can be one of the values listed in the following table.

Value	Description
0	No line is drawn around fold point lines
2	Draw line above, if expanded
4	Draw line above, if contracted
8	Draw line below, if expanded
16	Draw line below, if contracted (this value looks like an underline)

foldSymbols

Type: Integer[4]

Availability: Read or write at any time

The **foldSymbols** property of the **JadeTextEdit** control defines the set of fold point symbols used to mark fold point lines, when the value of the **folding** property is set to **true**.

The default value for the **JadeTextEdit** class is **SC_FOLDSYM_ARROWS** (**0**) and for the **JadeEditor** class it is **SC_FOLDSYM_TREESQUARE** (**3**).

The **foldSymbols** property value can be one of the **JadeTextEdit** class constants listed in the following table.

Class Constant	Integer Value
SC_FOLDSYM_ARROWS	0
SC_FOLDSYM_PLUSMINUS	1
SC_FOLDSYM_TREEROUND	2
SC_FOLDSYM_TREESQUARE	3

Alternatively, you can construct your own fold point symbol set by using the **setLinemarkAttributes** method to define the attributes of linemarks in the range **25** through **31**. (For details, see the Scintilla **SCI_MARKERDEFINE** command.)

folding

Type: Boolean

Availability: Read or write at any time

The **folding** property of the **JadeTextEdit** control specifies whether text editor lines can be folded to make them hidden. By default, the value of this property is **false**; that is, folding is not enabled.

When the value of the **folding** property is set to **true**, the fold margin (margin 2) is displayed and lines are marked as fold points (fold marks), based on the hierarchical structure of the editor text and the current language. Fold mark symbols indicate which text lines are fold points and their current stat (that is, contracted or expanded).

As fold point lines are chosen by the lexical analyzer of the current language, they are language-sensitive.

For the JADE language with normal folding, the start of a fold block is tied to the word at the end of the block instruction header (for example, **then** or **do**) and the end of a fold block is bound to the line preceding the block instruction trailer (**end**, **endif**, **endwhile**, and so on). With compact folding, the start of the fold block is tied to the first word of the block instruction and the end of the fold block is tied to the line including the instruction trailer.

The character sequences **//{** and **//}** are also recognized as fold block start and end, respectively.

When fold marks are displayed in the margin, clicking the mouse expands a contracted fold mark or contracts an expanded fold block. In addition to showing markers in the fold margin, you can use the **foldFlags** property to cause the editor to draw a line around a fold point lines using the foreground color of the default text style.

You can use the **foldSymbols** property to define which symbol set is displayed in the fold margin.

The mouse actions that you can perform within the fold margin of the **JadeTextEdit** control are listed in the following table.

Mouse Action	Result
Ctrl+Shift+left-click	Toggles (expands or contracts) all outer (parent) fold points
Shift+left-click	Toggles (expands or contracts) the nearest fold point and all lower-level (child) fold points to match
Left-click	Toggles (expands or contracts) the nearest fold point

fontBold

Type: Boolean

Availability: Read or write at any time

The **fontBold** property of the **Control** class specifies whether the font style is bold. This property is defined for all controls, but it has no meaning in some cases. For example, a scroll bar control has no text, and therefore the font is not relevant.

The settings for the **fontBold** property are listed in the following table.

Value	Description
true	Turns on the bold formatting
false	Turns off the bold formatting (the default)

For the **Table** control, the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the font property for the current sheet, cell, column, or row is being accessed. Changing the sheet font causes the default table row height to be changed.

Changing the font for a cell or row causes the current height of the row to change, unless the **rowHeight** property has been set for the row. Changing the font for a column causes all rows that do not have the **rowHeight** property set to have at least the font height size.

Use the **fontBold** property to format text, either in the JADE development environment or at run time by using logic.

For the **JadeTextEdit** control, the **clearAllStyles** method copies the value of this property to the default text style; that is, **STYLE_DEFAULT** (**32**).

Note The font uses the application font if the **fontName** property for the control is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

Use the **Control** class **setFontProperties** method to change the **fontBold**, **fontName**, and **fontSize** properties at the same time.

fontItalic

Type: Boolean

Availability: Read or write at any time

The **fontItalic** property of the **Control** class specifies whether the font style is italic. Although this property is defined for all controls, it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

The settings for the **fontItalic** property are listed in the following table.

Value	Description
true	Turns on the italic formatting
false	Turns off the italic formatting (the default)

For the **Table** control, the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the font property for the current sheet, cell, column, or row is being accessed.

Changing the sheet font causes the default table row height to be changed. Changing the font for a cell or row causes the current height of the row to change, unless the **rowHeight** property has been set for the row. Changing the font for a column causes all rows that do not have the **rowHeight** property set to have at least the font height size.

Use the **fontItalic** property to format text, either in the JADE development environment or at run time by using logic.

For the **JadeTextEdit** control, the **clearAllStyles** method copies the value of this property to the default text style; that is, **STYLE_DEFAULT** (**32**).

Note The font defaults to the application font if the **fontName** property for the control is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

fontName

Type: String[31]

Availability: Read or write at any time

The **fontName** property of the **Control** class contains the font used to display text in a control.

The **fontName** property is defined for all controls, but it does not have any meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

Note The font of a control defaults to the application font if the **fontName** property of the control is set to **Default** when printing the form. At run time, the **fontName** property returns an empty string if the control is using the default application font. Use **app.fontName** to obtain the actual font name.

For the **Table** class, the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the **fontName** property for the current sheet, cell, column, or row is being accessed. Changing the sheet font causes the default table row height to be changed. Changing the font for a cell or row causes the current row height to change, unless the **rowHeight** property has been set for the row. Changing the font for a column causes all rows not having the **rowHeight** property set to have at least the font height size.

For the **JadeTextEdit** control, the **clearAllStyles** method copies the value of this property to the default text style; that is, **STYLE_DEFAULT** (**32**).

The default value for the **fontName** property is determined by the system. Fonts that are available with JADE vary, according to your system configuration, display devices, and printing devices.

Notes Changing the **fontName** property to an empty string causes the control to use the default font. The **fontBold** and **fontSize** properties revert to the font of the application.

If a control is using the default font for the application (that is, this property contains the **""** null value), changing any font property of the control causes the control to use a local font constructed by using the application font values with the changed font attribute.

Use the **Control** class **setFontProperties** method to change the **fontName**, **fontBold**, and **fontSize** properties at the same time.

fontSize

Type: Real

Availability: Read or write at any time

The **fontSize** property contains the size of the font to be used for text displayed in a control.

The **fontSize** property of the **Control** class is defined for all controls, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore font is not relevant.

Use the **fontSize** property to format text in the required font size. The default value is determined by the system. To change the default, specify the size of the font in points.

For the **Table** class, the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the **fontSize** property for the current sheet, cell, column, or row is being accessed.

For the **JadeTextEdit** control, the **clearAllStyles** method copies the value of this property to the default text style; that is, **STYLE_DEFAULT** (**32**).

Changing the sheet font causes the default table row height to be changed. Changing the font for a cell or row causes the current row height to change, unless the **rowHeight** property has been set for the row. Changing the font for a column causes all rows not having the **rowHeight** property set to have at least the font height size.

Note The font defaults to the application font if the **fontName** property for the control is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

Use the **Control** class **setFontProperties** method to change the **fontSize**, **fontBold**, and **fontName** properties at the same time.

fontStrikethru

Type: Boolean

Availability: Read or write at any time

The **fontStrikethru** property of the **Control** class specifies whether the font style is strikethrough. This property is defined for all controls, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

The settings for the **fontStrikethru** property are listed in the following table.

Value	Description
true	Turns on the strikethrough formatting
false	Turns off the strikethrough formatting (the default)

For the **Table** class, the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the font property for the current sheet, cell, column, or row is being accessed.

Changing the sheet font causes the default table row height to be changed. Changing the font for a cell or row causes the current height of the row to change, unless the **rowHeight** property has been set for the row. Changing the font for a column causes all rows that do not have the **rowHeight** property set to have at least the font height size.

Use the **fontStrikethru** property to format text, either in the JADE development environment or at run time by using logic.

Note The font defaults to the application font if the **fontName** property for the control is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

fontUnderline

Type: Boolean

Availability: Read or write at any time

The **fontUnderline** property of the **Control** class specifies whether the font style is underlined. This property is defined for all controls, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

Use the **fontUnderline** property to format text, either in the JADE development environment or at run time by using logic.

The settings for the **fontUnderline** property are listed in the following table.

Value	Description
true	Turns on the underline formatting
false	Turns off the underline formatting (the default)

For the **Table** control, the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the font property for the current sheet, cell, column, or row is being accessed.

Changing the sheet font causes the default table row height to be changed. Changing the font for a cell or row causes the current height of the row to change, unless the **rowHeight** property has been set for the row. Changing the font for a column causes all rows that do not have the **rowHeight** property set to have at least the font height size.

For the **JadeTextEdit** control, the **clearAllStyles** method copies the value of this property to the default text style; that is, **STYLE_DEFAULT** (**32**).

Note The font defaults to the application font if the **fontName** property for the control is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

foreColor

Type: Integer

Availability: Read or write at any time

The **foreColor** property of the **Control** class contains the foreground color used to display text in a window. Changing the **foreColor** property causes a repaint of the window object. JADE uses the RGB scheme for colors.

Using the appropriate RGB value can set each property. The default setting of this property for most controls is the **WINDOW_TEXT** system default color. (See the **Window** class **getSystemColor** method for a description and example of returning a Windows system color.)

If a disabled cell, row, or column has a specified value of the **foreColor** property, the text in a disabled cell is displayed using that color. The text in a disabled cell is displayed using the Windows disabled text color if the **foreColor** property of the cell is not specifically set using the **foreColor** property values of the cell, row, or column.

The valid range for a normal RGB color is zero (**0**) through 16,777,215 (#FFFFFF). The high byte of a number in this range equals **0**; the lower three bytes (from least to most significant byte) determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is **128**, JADE uses the system colors, as defined in the Control Panel of the user. To determine the **Integer** value of a color from the RGB values:

```
int:= RedValue + (GreenValue * 256) + (BlueValue * 256 * 256);
```

For the **JadeTextEdit** control, the **clearAllStyles** method copies the value of this property to the default text style; that is, **STYLE_DEFAULT** (**32**).

For the **Table** class, the **accessMode** property determines whether the **foreColor** property for the current sheet, cell, column, or row is being accessed. The **foreColor** property is available on all control objects, but it is not relevant to some controls when painting the control image; for example, a **ScrollBar** control.

For a control, setting the **foreColor** property in the JADE Painter by selecting the **Parent's Colour** item (or setting the value of the **foreColor** property to **#80000000** at run time) instructs the window to use the **foreColor** property value of its parent.

The code fragment in the following example shows the use of the **foreColor** property.

```
table1.accessedCell.foreColor:=Blue;
```

form

Type: Form

Availability: Read-only at run time

The **form** property of the **Control** class and **MenuItem** class contains a reference to a form of a control or menu from the object.

Use this property when the control or menu object is passed as a parameter to a generalized method, so that the logic can still access the form of the control or menu.

The code fragment in the following example shows the use of the **form** property.

```
if lbl = null then
    create lbl transient;
    lbl.parent := self;
    form.addControl(lbl);
    if isInPainter then
        lbl.actualEnabled := false;
    endif;
    lbl.borderStyle := BorderStyle_Single;
    lbl.alignment   := Alignment_Center_Middle;
    if lbl = monthLabel then
        lbl.caption := today.monthName;
    else
        lbl.caption := today.year.String;
    endif;
endif;
```

formatOut

Type: String[100]

Availability: Read or write at any time

The **formatOut** property of the **TextBox** class or **Label** class contains the system-defined formats of data in text boxes or labels that you specify during printing.

The system-defined data formats are listed in the following table.

Option	Action
=date	Prints the current date as specified in Control Panel.

Option	Action
=direct	Sends the text of the control formatted in the font of the control directly to the printer. This provides you with the ability to send commands to the print driver; for example, the facsimile (fax) number when printing to a fax device. (See the **JadePrintDirect** class for details about the transient class that holds direct printing output directives.)
=formatdate	Prints the date in the format supplied in the **formatOut** text box of the Properties dialog **Specific** sheet in JADE Painter, as shown in the following example. `=formatdate dd/MM/yyy`
=longdate	Prints the current date in the long date format.
=page	Prints the current page number.
=pagenofm	Prints the current page number of the total number of pages in the document (for example, **2 of 8**).
=shortdate	Prints the current date in the short date format.
=time	Prints the current time (in hh:mm:ss am / pm format).
=totalpages	Prints the total number of pages in the document (for example, **8**).

When the **=pagenofm** and **=totalpages** option are included in labels or text boxes on report frames, they are drawn to the size of the resulting text regardless of the original size of the label or text box. No clipping occurs if the text is larger than the width of the control. The **alignment** properties of the label or text box will still be used. To achieve this, the printed output is first written to a temporary file and the count of the total number of pages is inserted into the print output at the specified positions.

Print output is then processed as follows if a report uses the total page count feature.

1. The output is written to a temporary file.

2. For reports output directly to a printer (that is, the **formatOut** property is set to **=direct**), printing cannot commence until the entire report is completed.

3. For reports being stored in the database (that is, reports that use the **Printer** class **setReport** method), output is retrieved from the temporary file and stored in the database only after the printer is closed.

   This is most evident when running in JADE thin client mode, as the printed output must be retrieved from the presentation client and passed to the application server at the end of the report rather than page by page, as the report is produced.

The code fragment in the following example shows the use of the **formatOut** property.

```
printForm.label7.formatOut := '=formatDate ' & dateFormats[formatIndex];
formatIndex.bump;
printForm.label8.caption := app.printer.String;
result := app.printer.print(printForm.detail);
```

See also the example of the **formatOut** property under "JadePrintDirect Class", in Chapter 1.

fullName

Type: String

Availability: Read or write at run time only

The **fullName** property contains the full name of the OLE object in an **OleControl**.

This name defaults to the OLE class or file name used to create the object.

The **fullName** property allows the object to have an identifying description assigned to the control and OLE object.

gridColor

Type: Integer

Availability: Read or write at any time

The **gridColor** property contains the color of grid lines in a **Table** control.

The default value is light gray.

gridLines

Type: Boolean

Availability: Read or write at any time

The **gridLines** property specifies whether lines are drawn between the rows and columns of the current sheet of the **Table** control.

The default value is **true**.

hasPictures

Type: Boolean

Availability: Read or write at any time

The **hasPictures** property controls the type of display for the **ListBox** control or the **ComboBox** control, by specifying whether the picture images are displayed. The default value is **false**.

Each list entry in the control can be made up of the following parts.

- TreeLines
- Plus/Minus
- Pictures
- Item picture
- Text

TreeLines are lines drawn between items in the hierarchy, to display the parents of an item and its siblings.

Plus/Minus is a bitmap or icon that is displayed when the entry has subitems. The image represents the expanded or collapsed state of that part of the hierarchy. A suitable bitmap or icon can be assigned to each of the **picturePlus** and **pictureMinus** properties for this part of the display. The default value is a plus (**+**) and minus (**-**) sign.

Pictures is an icon or bitmap that represents whether the list entry is a leaf (that is, it has no subentries) or is an expanded or collapsed entry. A suitable bitmap or icon can be assigned to the **pictureClosed**, **pictureOpen**, and **pictureLeaf** properties for this part of the display. The default value is a closed folder, open folder, and a document.

The parts of the display are optional. The **hasPlusMinus**, **hasTreeLines**, and **hasPictures** properties control these parts of the list. If the pictures are larger than the list line size, they are scaled to fit.

Clicking the treeline, plus/minus, or picture area of the display generates a **pictureClick** event. Logic must then expand or collapse the item, as required. As this process is not automatic, the expand or collapse process can be controlled by the user. For example, the subentries may not be loaded into the list box until an entry is expanded.

By default, the appropriate picture icon is displayed. Setting the **itemPictureType** property can also manually control the type of picture. When the **itemPictureType** property is set for an item, no further automatic picture assignment is performed based on expand, collapse, or subitems.

The type of image displayed for the plus/minus display is always automatic. If you want to use these images with user control, set the picture properties to these icons.

To cause the **pictureDblClick** event, double-click any portion of an entry to the left of the text.

Setting the **sorted** property for a list box control sets the **hasPictures**, **hasPlusMinus**, and **hasTreeLines** properties to **false**.

To select the list entry (to set the **listIndex** property), the text portion of the entry must be clicked or the arrow keys used.

hasPlusMinus

Type: Boolean

Availability: Read or write at any time

The **hasPlusMinus** property controls the type of display for the **ListBox** control or the **ComboBox** control, by specifying whether the plus/minus images are displayed. The default value is **false**.

Each list entry in the control can be made up of the following parts.

- TreeLines
- Plus/Minus
- Pictures
- Item picture
- Text

TreeLines are lines drawn between items in the hierarchy, to display the parents of an item and its siblings.

Plus/Minus is a bitmap or icon that is displayed when the entry has subitems. The image represents the expanded or collapsed state of that part of the hierarchy. A suitable bitmap or icon can be assigned to each of the **picturePlus** and **pictureMinus** properties for this part of the display. The default value is a plus (+) and minus (-) sign.

Pictures is an icon or bitmap that represents whether the list entry is a leaf (that is, it has no subentries) or is an expanded or collapsed entry. A suitable bitmap or icon can be assigned to the **pictureClosed**, **pictureOpen**, and **pictureLeaf** properties for this part of the display. The default value is a closed folder, open folder, and a document.

The parts of the display are optional. The **hasPlusMinus**, **hasTreeLines**, and **hasPictures** properties control these parts of the list. If the pictures are larger than the list line size, they are scaled to fit.

Clicking the treeline, plus/minus, or picture area of the display generates a **pictureClick** event. Logic must then expand or collapse the item, as required. As this process is not automatic, the expand or collapse process can be controlled by the user. For example, the subentries may not be loaded into the list box until an entry is expanded.

By default, the appropriate picture icon is displayed. Setting the **itemPictureType** property can also manually control the type of picture. When the **itemPictureType** property is set for an item, no further automatic picture assignment is performed based on expand, collapse, or subitems.

The type of image displayed for the plus/minus display is always automatic. If you want to use these images with user control, set the picture properties to these icons.

To cause the **pictureDblClick** event, double-click any portion of an entry to the left of the text.

Setting the **sorted** property for a list box control sets the **hasPictures**, **hasPlusMinus**, and **hasTreeLines** properties to **false**.

To select the list entry (to set the **listIndex** property), the text portion of the entry must be clicked or the arrow keys used.

hasTreeLines

Type: Boolean

Availability: Read or write at any time

The **hasTreeLines** property controls the type of display for the **ListBox** control or the **ComboBox** control, by specifying whether the tree lines are drawn. The default value is **false**.

Each list entry in the control can be made up of the following parts.

- TreeLines
- Plus/Minus
- Pictures
- Item picture
- Text

TreeLines are lines drawn between items in the hierarchy, to display the parents of an item and its siblings.

Plus/Minus is a bitmap or icon that is displayed when the entry has subitems. The image represents the expanded or collapsed state of that part of the hierarchy. A suitable bitmap or icon can be assigned to each of the **picturePlus** and **pictureMinus** properties for this part of the display. The default value is a plus (+) and minus (-) sign.

Pictures is an icon or bitmap that represents whether the list entry is a leaf (that is, it has no subentries) or is an expanded or collapsed entry. A suitable bitmap or icon can be assigned to the **pictureClosed**, **pictureOpen**, and **pictureLeaf** properties for this part of the display. The default value is a closed folder, open folder, and a document.

The parts of the display are optional. The **hasPlusMinus**, **hasTreeLines**, and **hasPictures** properties control these parts of the list.

Clicking the treeline, plus/minus, or picture area of the display generates a **pictureClick** event. Logic must then expand or collapse the item, as required. As this process is not automatic, the expand or collapse process can be controlled by the user. For example, the subentries may not be loaded into the list box until an entry is expanded.

By default, the appropriate picture icon is displayed. Setting the **itemPictureType** property can also manually control the type of picture. When the **itemPictureType** property is set for an item, no further automatic picture assignment is performed based on expand, collapse, or subitems.

The type of image displayed for the plus/minus display is always automatic. If you want to use these images with user control, set the picture properties to these icons.

To cause the **pictureDblClick** event, double-click any portion of an entry to the left of the text.

Setting the **sorted** property for a list box control sets the **hasPictures**, **hasPlusMinus**, and **hasTreeLines** properties to **false**.

To select the list entry (to set the **listIndex** property), the text portion of the entry must be clicked or the arrow keys used.

height

Type: Real

Availability: Read or write at any time

The **height** property of the **Window** class contains the dimensions of an object. Measurements are calculated using the following.

- For controls, the external height of the control
- For forms, the external height of the form, including the borders and title bar

For a form, the **height** property is always in pixels. For a control, the **scaleMode** property units of the parent control determine the height. The **scaleMode** property defaults to pixels. For a form or control, the value for the **height** property changes as the object is sized by the user or by logic. The maximum limit for all objects is system-dependent. (See also the **parentAspect** property.)

Note You cannot change this property for **CheckBox**, **OptionButton**, and **ComboBox** controls whose **style** property is *not* **Style_Simple** (**1**).

If the value of the **height** property plus the value of the **top** property is greater than 32,767 pixels, the resulting window extents may be unpredictable.

Windows limits forms and controls to a maximum height of 32,767 pixels. Setting a value larger than the maximum height results in a value of 32,767 pixels being used.

The code fragments in the following examples show the use of the **height** property.

```
table1.height := table1.rowHeight[1] * table1.rows;

startName.top := theTable.top + theTable.height + 10;

pictureEnlarged.height := (pictureNormal.height/(x - xStart)) * 100;
```

helpContextId

Type: Integer

Availability: Read or write at any time

The **helpContextId** property of the **Window** class contains an associated context number for an object. This property is used to provide context-sensitive help for your application.

If the **helpKeyword** property is also set, the keyword is used in preference to the context number. For context-sensitive help on an object in your application, you must assign the same context number to both the object and to the associated help topic when you compile your help file.

If you have created a Windows environment help file for your application (that is, a **.hlp** or **.chm** file, or Hypertext Markup Language (**.htm** or **.html**) files), when a user presses the F1 function key, JADE automatically calls help and requests the topic identified by the current context number (or the **helpKeyword** property).

The current context number is the value of the **helpContextId** property for the object that has the focus. For a control, if the **helpContextId** property is set to zero (**0**) and the **helpKeyword** property value is **null**, JADE examines the corresponding properties on the window that is the **parent** window of the control, which is either the form or another control. If the **helpContextId** property is set to zero (**0**) and the **helpKeyword** property value is **null** for the parent window, the *grandparent* window is examined and so on, until the form is reached.

If a non-zero current context number cannot be found, the Contents section of the help file is requested. If the **helpFile** property of the **Application** class is not set, no help file is opened.

Notes Building a help file requires the Microsoft Windows Help Compiler or the Adobe Acrobat application.

As PDF files require string values as help destinations, see the **PdfHelpIdPrefix** parameter under "JADE Help Section [JadeHelp]", in the *JADE Initialization File Reference* for details about formatting a **helpContextId** property into a string value.

helpKeyword

Type: String

Availability: Read or write at any time

The **helpKeyword** property of the **Window** class contains the text used to access the help file. If a help keyword is provided for a form, control, or menu, this text is used to access the help file when the user presses the F1 function key for help while the focus is on that object.

The current keyword is the value of the **helpKeyword** property for the object that has the focus.

In the **Window** and **MenuItem** classes, this property can be translated when the value of the **Schema** class **formsManagement** property is **FormsMngmt_Single_Multi** (**2**).

For a control, if the **helpKeyword** property is empty and its **helpContextId** property is set to zero (**0**), JADE examines the corresponding properties on the window that is the **parent** window of the control, which is either the form or another control. If the **helpKeyword** property is empty and its **helpContextId** property is set to zero (**0**) for the parent window, the *grandparent* window is examined and so on, until the form is reached.

If no help keyword or context number can be found, the **Contents** section of the help file is requested. If the **helpFile** property of the **Application** class is not set, no help file is opened. If the **helpContextId** property is also set, the **helpKeyword** value is used in preference to the context number.

Note Building a help file requires the appropriate third-party tool (that is, the Adobe Acrobat application, Microsoft Windows Help Compiler, or any other Windows help compiler).

When help is requested, if the help file specifies a:

- Web HTML help, detected by the value of the **helpFile** property starting with a recognized URL scheme (that is, **http://** or **https://**), JADE attempts to construct a URL to pass to the default Web browser.

  If the value of the **helpKeyword** property starts with a recognized URL scheme, the **helpKeyword** URL is used; otherwise the value of the **helpKeyword** property is appended to the value of the **helpFile** property to become the URL to use, as shown in the following code fragment example.

  ```
  mybtn.helpKeyword := "Form1/button1.html";
  mybtn.showHelp;
  ```

  When help is invoked directly via the **Window** class **showHelp** method or via the user pressing the help key (F1), a URL is created and the default browser is invoked to display the URL.

- Portable Document Format (PDF) file (detected by the **.pdf** file suffix), JADE attempts to execute Adobe Acrobat to handle the file. JADE checks the Windows registry for the Acrobat Reader (**AcroRd32**) or for the **acrobat** executable program. If Acrobat Reader is not found, the help request is ignored and entries explaining the cause of the failure are output to the **jommsg.log** file. If Acrobat is located, it is initiated for the PDF help file defined in JADE.

  For a **helpKeyword** help request, the **helpKeyword** property is passed to Acrobat as a **named destination**, which Acrobat uses to position the help file display. As there are no equivalent concepts in a PDF file of any other type of help request (for example, **helpContextId**, index request, and so on), only the first page of the PDF file is displayed for a help request other than using the **helpKeyword** property.

- Windows help file (detected by the **.hlp** file suffix), JADE automatically calls help and requests the topic identified by the current **helpKeyword** property or the **helpContextId** property.

- Compiled help file (detected by the **.chm** file suffix), JADE calls the **HtmlHelp** entry point of the **htmlhelp.dll** file and requests the topic identified by the current **helpKeyword** property or the **helpContextId** property. You can use the compiled help file (**.chm**) format files when producing online help for HTML thin client applications, for example.

- The **helpKeyword** property can contain a help file name before the keyword, separated by a semicolon. This help file (which can be a **.pdf**, **.hlp**, or **.chm** file) is specific to this **helpKeyword** property, and overrides the default value; for example:

  ```
  btnHelp_click(btn: Button input) updating;
  vars
  begin
      if fldFolder.topSheet = shtSelect then
          btn.helpKeyword := "DevRef.pdf;selectinglibraryacxautomationdrg10";
      elseif fldFolder.topSheet =  shtLibrary then
          btn.helpKeyword := "DevRef.pdf;namelibrary_activex";
      elseif fldFolder.topSheet =  shtObjects then
          btn.helpKeyword := "DevRef.pdf;namingobjectclassesacxautomationdrg10";
      elseif fldFolder.topSheet =  shtInterfaces then
          btn.helpKeyword := "DevRef.pdf;naminginterfacesacxautomationdrg10";
      elseif fldFolder.topSheet =  shtConstants then
          btn.helpKeyword := "DevRef.pdf;namingconstantsacxautomationdrg10";
      endif;
  ```

```
        btn.showHelp;
    end;
```

Tip Although it is more efficient to use a single help file, specified in the **Help File** text box on the **Application** sheet of the Define Application dialog, this feature is intended for situations in which multiple help files are required for a single application.

- When handling automatic Help menu items, if a **helpContextId** or **helpKeyword** property is specified on the Help menu **Index** automatic menu item, the destination of the help is based on the value of the **helpContextId** or **helpKeyword** property. In addition, the **click** event method is not executed.

hintBackColor

Type: Integer

Availability: Read or write at any time

The **backColor** property of the **TextBox** class contains the color with which text box background is displayed when hint text is displayed. (JADE uses the RGB scheme for colors.)

Hint text displayed in the text box is drawn using the value of the **hintBackColor** property except when it is **#80000000** (the default), in which case the **TextBox** control **backColor** property value is used.

If the text box text is not empty or no value has been set for the **hintText** property, this property is ignored.

Applies to Version: 2016.0.01 and higher

hintForeColor

Type: Integer

Availability: Read or write at any time

The **hintForeColor** property of the **TextBox** class contains the color with which text is displayed when hint text is displayed. (JADE uses the RGB scheme for colors.)

Hint text displayed in the text box is drawn using the value of the **hintForeColor** property except when it is **#80000000** (the default), in which case the **TextBox** control **foreColor** property value is used.

If the text box text is not empty or no value has been set for the **hintText** property, this property is ignored.

Applies to Version: 2016.0.01 and higher

hintText

Type: String

Availability: Read or write at any time

The **hintText** property of the **TextBox** class contains the text displayed in an empty text box as a hint. If the text box is not empty or hint text is not defined, this property is ignored.

When hint text is displayed, the text foreground and background colors are colored using the values of the **hintBackColor** and **hintForeColor** colors, if they are set.

Hint text cannot be selected or deleted, and the cursor is always positioned at the beginning of the text.

As soon as text is entered or pasted into the text box, the hint text is removed and replaced with the specified or pasted text. If the entire text is removed, the hint text is displayed again.

When hint text is displayed, the:

- Back ground of the text box is drawn using the value of the **hintBackColor** property except when it is **#80000000**, in which case the **TextBox** control **backColor** property value is used.
- Text of the text box is drawn using the value of the **hintForeColor** property except when it is **#80000000**, in which case the **TextBox** control **foreColor** property value is used.

The values of the **dataType**, **case**, and **selectionStyle** properties are ignored when the hint text is displayed; for example, hint text can be displayed for a numeric field. The hint text, which is always displayed in the case of its defined string, can never be selected. In addition, for a password text box, the hint text is displayed as clear text (that is, it is not displayed using asterisk (*****) characters).

If the values of the **hintBackColor** and **hintForeColor** properties are both zero (**0**), the default values of **#80000000** (that is, the **TextBox** control **backColor** and **foreColor** properties, respectively) are used instead.

Applies to Version: 2016.0.01 and higher

horizontalSpace

Type: Integer

The **horizontalSpace** property of the **WebJavaApplet** class contains the number of pixels on each side of the applet on your Web page.

hyperlink

Type: String

The **hyperlink** property of the **Label** class and **Picture** class contains a hyperlink string that is programmatically attached to the label or picture control and that displays HyperText links to static text.

The maximum length of a hyperlink string is 64,000 characters.

If the value of this property is **null** for a Web label, a logic-type link is assumed and the associated **click** event for the label is executed, if one exists.

The formats of Web label hyperlinks are:

```
welcome.html

welcome.html#page1

http://domain-name/virtual-directory/welcome.html

http://domain-name/virtual-directory/welcome.html#page1
```

For a **WebJavaApplet** control, if your Java applet is contained in an archive file, set the **hyperlink** attribute to the name of the **.jar** file.

hyperlinkColumn

Type: Integer array

The **hyperlinkColumn** property of the **Table** class contains an array of integers that represent a column in each row of a table that has an associated HyperText link. If the value of an array element is zero (**0**), the row does not have an associated HyperText link.

Note The **hyperlinkColumn** property is ignored for JADE applications that are not Web-enabled; that is, applications running in standard JADE clients and thin clients. For these applications, you should use the **setHyperlinkCell** method of the **Table** class or the **hyperLink** property of the **JadeTableCell** class.

Clicking a column that has a HyperText link causes a **rowColumnChg** event for the table. If no **rowColumnChg** event exists for the table, the HyperText link does nothing.

Note The **hyperlinkColumn** property is not changed when the table is sorted, which can result in the wrong cells being rendered as hyperlinks.

icon

Type: Binary

Availability: Read or write at any time

The **icon** property of the **Form** class or **Sheet** class contains the icon displayed for a minimized form or on the tab of a sheet in a folder. (The **Application** class **icon** property acts as the default for any form or sheet that does not have a defined icon.) Use this property to specify a custom icon for any form that the user can minimize at run time; for example, you can assign a unique icon to a form to indicate the function of the form.

Specify the icon by loading it using the Properties window in Painter in the JADE development environment, or load it at run time by assigning it from another icon or by using the **loadPicture** method of the **Application** class (**app.loadPicture**).

If the icon is loaded from a file, it must have an icon format (that is, a **.ico** file). If you do not specify a custom icon, the application icon is used. If no application icon was specified, the JADE default icon is used. JADE creates a large and a small icon for use with a form if they are present in the icon file when the **app.icon** and **form.icon** properties are set.

Note To see the icon of a form, the form must be minimized. The **borderStyle** property must be set to **BorderStyle_Single** (1) or **BorderStyle_Sizable** (2). The **minButton** property must be set to **true**.

At run time, you can assign the **icon** property of an object to the **icon** property of another object or use the **loadPicture** method of the **Application** class to load an icon from a file.

If you set the **icon** property of a **Sheet** control to an icon image (the only image type that is accepted), that icon is displayed in the folder tab for the sheet. The icon is scaled to fit the tab height. Use the **tabsHeight** property to control the height, and therefore the size, of the displayed sheet icon. Use the **tabsAlignment** property to control the placement of the sheet icon on the tab.

ignoreHeight

Type: Boolean

Availability: Read or write at any time

Set the **ignoreHeight** property of the **WebHTML** class to **true** to specify that text defined in the JADE development environment is resized to fit the height of the HTML on a Web page accessed using Internet Explorer 4.0 (or higher).

The default value of **false** specifies that text retains the height defined in the JADE development environment when displayed on a Web page.

ignoreSkin

Type: Boolean

Availability: Read or write at any time

Set the **ignoreSkin** property of the **Window** class to **true** to specify that the window is drawn without a skin. By default, this property is set to **false** and the window (form or control) uses any appropriate skin.

For details about using skins to enhance your runtime applications, see Chapter 2 of your *JADE Runtime Application Guide*.

See also the **Application**::**setApplicationSkin** method, "JadeSkinApplication Class", and "JadeSkinControl and Subclasses", in Chapter 1. See also the **Form** class **setApplicationSkin** method, later in this document.

ignoreWidth

Type: Boolean

Availability: Read or write at any time

Set the **ignoreWidth** property of the **WebHTML** class to **true** to specify that text defined in the JADE development environment is resized to fit the width of the HTML on a Web page accessed using Internet Explorer 4.0 (or higher).

The default value of **false** specifies that text retains the width defined in the JADE development environment when displayed on a Web page.

indentGuides

Type: Boolean

Availability: Read or write at any time

The **indentGuides** property of the **JadeTextEdit** control specifies whether vertical indentation guidelines are displayed in the client area. The default value is **false** (that is, guidelines are not displayed).

Specify the foreground and background colors of the indent mark by calling the **setStyleAttributes** method with the **styleNumber** parameter set to **STYLE_INDENTGUIDE** (**37**).

Note It is recommended that the background color be left as the default background and that the foreground should be light gray (for example, **#C0C0C0**).

indentWidth

Type: Integer[4]

Availability: Read or write at any time

The **indentWidth** property of the **JadeTextEdit** control contains the width in characters of the text editor indentation if you do want to override the value of the **tabWidth** property.

The indent width can be a value in the range zero (**0**) through **100**. By default, the **indentWidth** property is set to zero (**0**), indicating that the **tabWidth** value is used.

index

Type: Integer

Availability: Read-only at run time only

The **index** property for control and menu items is used only when controls are cloned by using the **loadControl** method. The **index** property of controls created in the JADE development environment is set to zero (**0**).

The user passes the value of this property to the method and it is stored with the control properties. It can then be used as an identifier to differentiate between the control copies that all have the same name.

Most commonly, you would assign the **index** values sequentially, using them like indexes.

initialContent

Type: String

Availability: Read or write only during development

The **initialContent** property of the **JadeRichText** control contains the initial content of the control. This property would normally be set only from the JADE Painter Properties dialog.

When the control is created, it is initialized from this value. If the text starts with a valid RTF header sequence (for example, **"{\rtf"**), the text is loaded by the RTF reader.

inputType

Type: Integer

Availability: Read or write at run time only

The **inputType** property contains the type of input (if any) that is accepted by a cell of a table. You can assign the **inputType** property to each cell, row, and column of a **Table** control. The types of input that can be accepted are listed in the following table.

Constant	Value	Description
InputType_None	0	Cell does not allow user input (the default).

Constant	Value	Description
InputType_CheckBox	1	The cell displays a check box bitmap, which can be toggled by using the mouse, space bar, or Enter key. The text of the cell is set to **0** or **1**, according to the state. The initial value is set to **0** if there is no text or the first character of any text is **0**; if not, it is set to **1**. User changes to the state of the check box control generate a **change** event. **Note** A disabled cell with **InputType_CheckBox** draws the check box as disabled.
InputType_TextBox	2	When the cell is current, the cell allows text input. As the text changes, the **change** event is called. The **maxLength** property can be set for the cell to control the amount of text that can be entered. The default value is **0** (that is, there is no limit).
InputType_ComboBox	3	**ComboBox** (non-sorted drop-down list only). Users can select an entry from the list by using the mouse or an arrow or RETURN key. The **comboList** property sets the list entries, and the **comboIndex** property sets the current selected entry. When the user selects a list entry or when logic sets a **comboIndex** property value, the value of the **text** property for the cell is set to the list entry. When the user selects a new entry, a **change** event is generated. If logic sets the **text** property of a cell, that text must correspond exactly to one of the list entries.
InputType_TextNumeric	4	When the cell is current, the cell allows only numeric text input. As the text changes, the **change** event is called. The **maxLength** property can be set for the cell, to control the amount of text that can be entered. The default value is **0** (that is, there is no limit). Use the **decimals** property if you want to enter decimal places into cells that accept numeric text input.
InputType_Default	5	Returns the cell, column, row, or sheet to its default **inputType** property setting. This constant is useful, for example, if you set a column to **InputType_None** so that it is temporarily disabled, and you later want to enable its input using the default input type.
InputType_SignedNumeric	6	Handles the entry of a signed numeric value in a table input field. Use the **decimals** property if you want to enter decimal places into cells that accept signed numeric input.
InputType_EditMask	7	The input control acts like an edit mask text box for input by default. For details about setting the mask used for input for a cell, row, column, or sheet, see the **JadeEditMask** class or the **Table** class **editMask** property.

When using a numeric text box (the **inputType** property is set to **InputType_TextNumeric**), the following events are still fired when the entered key is invalid.

- **Form**::**keyDown**
- **TextBox**::**keyDown**
- **Form**::**keyPress**

This enables the form to process keys such as the Enter key in the **Form**::**keyDown** or **keyPress** events and the control to process these keys in its **keyDown** event, therefore allowing application users to use the numeric keypad when a form requires a large amount of numeric data entry.

The **inputType** property applies to the current sheet, row, and column for the table.

Notes The **accessMode** property determines whether the **inputType** property is for the current sheet, cell, column, or row is being accessed.

For a **TextBox** control, pressing the left arrow key when the vertical bar symbol (**|**) is in the left position moves the focus one cell left. Similarly, pressing the right arrow key advances the focus to the next cell on the right when the vertical bar symbol is at the end of the text.

When focused on a text or combo entry cell, the normal table cell movement functions controlled by arrow keys, Home, End, Page Up, Page Down, and Enter keys may have specific meaning to the text or combo box. Use the Alt key in combination with the navigation key to override the text box or combo box actions.

Any picture assigned to a cell with the **inputType** property set to a non-zero value is displayed only when that cell does not have focus. Similarly, the cell **alignment** property is affected only when the cell does not have focus.

The check box type is always displayed with centered alignment.

The size of a row is automatically enlarged to fit the minimum size text or combo box.

When the **inputType** property value is set to **InputType_ComboBox** or **InputType_TextBox**, you can assign a control object to the cell, to enable your logic to affect input from these controls.

The object is associated with the cell, row, or column by referencing the **cellControl** property when the input type is set to text box or combo box.

Note If the cell, row, or column already contains an effective **cellControl** property value, the existing object is returned.

When an object is associated with the default input associated with table cells of the required type, the control object can then be manipulated from the logic of that control. This control receives neither focus nor **cellInputReady** events.

When a cell is initialized to receive input from a default **inputType** property, the input control is set up as follows, and any attempt to modify these properties results in undefined results.

- The control position and size are set to cover the cell.
- By default, the **backColor**, **foreColor**, and the font properties use the values associated with the current cell.
- The alignment of the text box is set to the alignment of the cell.
- The text in the text box text is set to the value of the text in the cell.
- The **maxLength** property of the text box is set corresponding to the value of the **inputType** and **decimals** properties of the cell.
- The combo box is cleared.
- The combo box is filled with the entries from the **comboList** property of the cell.
- The **listIndex** property of the combo box is set corresponding to the text of the cell.
- The focus is moved to the text box or the combo box.

The following examples show the use of the **inputType** property.

```
buttonAddColumn_click(btn : Button input) updating;
begin
    if selectedColumn <> null then
        table1.inputType := Table.InputType_TextBox;
        table1.insertColumn(selectedColumn);
        table1.clearAllSelected;
        selectedColumn := null;
        comboBoxColMoveTo.addItem("Move to Column " &
                                 (comboBoxColMoveTo.listCount + 1).String);
        comboBoxColumns.addItem((comboBoxColumns.listCount + 1).String);
        refresh;
    else
        app.msgBox("You must select a column", "No column selected",
                    MsgBox_OK_Only);
        return;
    endif;
end;
```

```
foreach count in 1 to 10 do // allows decimal input of column 9 for 10 rows
    table1.row := count;
    table1.accessCell(count, 9);
    table1.accessedCell.inputType := Table.InputType_SignedNumeric;
    table1.accessedCell.decimals  := 2;
endforeach;
```

insertMode

Type: Boolean

Availability: Read or write at any time

The **insertMode** property of the **JadeEditMask** class specifies whether the initial setting of the control is in **Insert** or **Overwrite** mode. The default value of **true** indicates that characters are inserted into the text box as they are entered (as they are for a standard **TextBox** control). If the value is **false**, the character at the caret position is replaced (that is, **Overwrite** mode applies).

There is no visual indication of the mode that the user is in. If the user presses the Insert key and the control has focus, the current **insertMode** is toggled and the change is reflected in the value of this property.

Notes Changing the value of the **insertMode** property in your logic or by the user does not affect any other control.

This property has no effect when the character at the caret position is a prompt character (which is always overwritten) or a literal character. If the same literal character is entered, the caret skips to the next character position that can be entered and any other character is ignored.

integralHeight

Type: Boolean

Availability: Read or write at any time

When the **integralHeight** property of **TextBox** controls and **ListBox** controls is set to **true**, the control height is determined by multiples of text height (that is, a multiple of the character size of the text displayed in that control). When this property is set to **false**, the control height is not determined by the text height.

By default, the integral height of both single line and multiple line text box controls is set to **false** and the integral height of list box controls is set to **true**.

Notes As the text box does not show partial text lines when the **integralHeight** property is set to **false**, the control offsets the text area from the top (that is, any *extra* space appears at the top of the text box).

The **integralHeight** property is ignored if the parent of **TextBox** and **ListBox** controls has the **alignChildren** property set.

itemBackColor

Type: Integer array

Availability: Read or write at run time only

The **itemBackColor** property contains a reference to the background color of each item in a **ListBox** control or **ComboBox** control. This property contains an array of color values (integers) with the same number of items as that returned by the **listCount** method in a list box control or combo box control.

Each entry in a list automatically assumes the background color assigned to the list box or combo box control by using the **backColor** property. However, you can individually assign each entry its own value.

The following example shows the use of the **itemBackColor** property.

```
loadListBox() updating;
vars
    obj : Object;
begin
    app.mousePointer := self.MousePointer_HourGlass;
    if currentDict <> null then
        foreach obj in currentDict do
            listInstances.addItem(obj.display);
            listInstances.itemObject [listInstances.newIndex] := obj;
            if listInstances.newIndex.isEven then
                listInstances.itemBackColor[listInstances.newIndex] :=
                              LightYellow;
            endif;
        endforeach;
    endif;
    app.mousePointer := self.MousePointer_Arrow;
end;
```

itemData

Type: Integer array

Availability: Read or write at run time only

The **itemData** property contains a reference to an array of integer values with the same number of items as that returned by the **listCount** method. Each inserted entry in a **ComboBox** or **ListBox** control has an **itemData** property value associated with it.

Use the **itemData** property to associate a specific number with each item in a combo box or list box. You can then use these numbers in logic to identify the items. For example, you can use the identification number of an employee to identify each employee name in a list box.

When you fill the list box, also fill the corresponding elements in the **itemData** property array with the employee numbers.

Use the **itemData** property for an index into an array of data structures associated with items in a list box, particularly when the list box is sorted. After the item is added to the control, use the return value of the **addItem** property or the **newIndex** method to obtain the index at which the entry was added. (See also the **itemObject** property.)

Note When you insert an item into a list by using the **addItem** or **addItemAt** method, an entry is also physically inserted in the **itemData** property array and is initialized to zero (**0**).

The following example shows the use of the **itemData** property.

```
getProdNumber(): Integer;
vars
    ref : Integer;
begin
    ref := listProds.itemData[listProds.listIndex];
    return ref;
end;
```

itemEnabled

Type: Boolean array

Availability: Read or write at run time only

The **itemEnabled** property allows individual items in a **ListBox** or **ComboBox** control to be disabled or enabled.

The **itemEnabled** property contains a reference to an array of **Boolean** values with the same number of items as returned by the **listCount** method.

By default, each entry in a list is enabled. The entries must be specifically disabled from logic. An item that is disabled has no impact on the logic that can be performed against it, only on the user actions for that entry in a list box or combo box.

A disabled entry does not respond to mouse or keyboard actions, and its text appears grayed. No events are sent when a disabled item is clicked. Using the arrow keys to move between entries in the list skips over disabled items.

The code fragment in the following example shows the use of the **itemEnabled** property.

```
while count <= listbox.listCount do
    listbox.itemEnabled[count] := true;
    count := count + 1;
endwhile;
```

itemExpanded

Type: Boolean array

Availability: Read or write at run time only

The **itemExpanded** property enables the expansion (or collapse) status of each item in a **ListBox** control or a **ComboBox** control to be obtained or set.

The **itemExpanded** property contains a reference to an array of **Boolean** values with the same number of items as returned by the **listCount** method.

Expanding an item that is not visible automatically expands all of the parents of the entry so that it becomes visible.

The **itemExpanded** property of an entry without subitems is always set to **false**. Setting the **itemExpanded** property for such an entry has no effect. If the specified item has subitems, the **itemExpanded** property returns whether the item has been expanded or is currently collapsed.

Setting this property to **true** results in the expansion of the subitems (if they are not already expanded). Setting this property to **false** results in the hiding of all subitems (if they are not already collapsed).

The code fragments in the following examples show the use of the **itemExpanded** property.

```
while count < listOrg.listCount do
    listOrg.itemExpanded [count] := true;
    count := count + 1;
endwhile;

if listbox.itemExpanded[picIndex] then
    // Remove all of the Retail Sale Items for this Category.
    childIndex := listbox.findStringExact(picIndex, $ItemsForSale);
    if listbox.itemExpanded[childIndex] then
        zRemoveItemsFromList(childIndex);
    endif;
    // Remove all of the Tender Sale Items for this Category.
    childIndex := listbox.findStringExact(picIndex, $ItemsForTender);
    if listbox.itemExpanded[childIndex] then
        zRemoveItemsFromList(childIndex);
    endif;
endif;
```

Entries are not automatically expanded or collapsed. You have control over this process. In most cases, you need to add the following logic to the **pictureClick** method.

```
list1.itemExpanded[picIndex] := not list1.itemExpanded[picIndex];
```

In this example, the **picIndex** value is the parameter that is passed to this method.

itemForeColor

Type: Integer array

Availability: Read or write run time only

The **itemForeColor** property allows the text color of each item in a **ListBox** or **ComboBox** control to be assigned. The **itemForeColor** property contains a reference to an array of integer color values (integer, **1**-relative) with the same number of items as that returned by the **listCount** method.

Each entry in the list automatically assumes the text color assigned to the control assigned by the **foreColor** property. However, you can individually assign each item its own text value.

The following example shows the use of the **itemForeColor** property.

```
setColor(prod: Product) updating;
begin
    if prod.inStock then
        listProducts.itemForeColor[listProducts.newIndex] := Red;
    else
        listProducts.itemForeColor[listProducts.newIndex] := Gray;
    endif;
    if listProducts.newIndex.isEven then
        listProducts.itemBackColor[listProducts.newIndex] := LightYellow;
    endif;
end;
```

itemLevel

Type: Integer array

Availability: Read or write at run time only

The **itemLevel** property enables the hierarchical level of each item in a **ListBox** control or a **ComboBox** control to be obtained or set.

The **itemLevel** property contains a reference to an array of integer values with the same number of items as returned by the **listCount** method. The level of an item must be in the range 1 through 63. Setting this property is ignored if the list box value of the **sorted** property is set to **true**.

The level to which an item is set is rejected if it creates an invalid hierarchy. The rules are:

- The first entry in the list must have a level of **1**.
- The level that is set cannot be greater than the level of the prior entry + 1. If this were allowed, the set item would have no immediate parent.
- The level of the next entry cannot be greater than the set level + 1. If this were allowed, it would leave the next entry without an immediate parent.

The expanded state of the changed entry is set to **false**. All the subitems that it inherits are collapsed and made not visible. The changed item adopts a visibility that is implied by the expansion status of its parent.

Subitems that the entry lost because of the change of level adopt the visibility and expansion status derived from their new parents.

The following example shows the use of the **itemLevel** property.

```
displayEmployees(emp:   Employee;
                 level: Integer) updating;
vars
    e : Employee;
begin
    listOrg.addItem(emp.name);
    listOrg.itemLevel[listOrg.newIndex] := level;
    foreach e in emp.myEmployees do
        displayEmployees(e, level + 1);
    endforeach;
end;
```

itemObject

Type: Object array

Availability: Read or write at run time only, read-only when a collection is attached to a combo box or list box

The **itemObject** property enables you to store an object with each entry in a **ListBox** or **ComboBox** control, and with each cell of a **Table** control. This then allows logic to retrieve that object when the user clicks on the entry. For example, for each customer shown in a list, a reference to the **Customer** object can be stored with the list entry.

Each inserted entry in a **ComboBox** and **ListBox** control has an **itemObject** property value associated with it. The **itemObject** property contains a reference to an array of object values with the same number of items as the **listCount** method of a control.

Each cell of a table control has an **itemObject** property value associated with it.

For **ComboBox** and **ListBox** controls, use the **itemObject** property array indexed by the required entry. For **Table** controls, the **itemObject** property refers to the currently selected cell defined by the **sheet**, **row**, and **column** properties.

Notes When you insert an item into a list by using the **addItem** or **addItemAt** method, a **null** object entry is also inserted in the **itemObject** property array.

As the object reference that is stored is of the **Object** class, it may then need to be cast to the required class so that it can be used.

An **itemObject** reference is not a true collection but a wrapper that allows you to use the **[ ]** syntax when referencing the window definition of a control (for example, **table.itemObject[indx]**) and in an iterator (for example, **foreach obj in listBox.itemObject**). The **Collection** classes they point to are internal pseudo arrays (that is, arrays of GUI-related information only in the JADE run time module), in which the only methods that are implemented are the **Collection** class **size** and **size64** methods, and the **Array** class **at**, **atPut**, and **createIterator** methods. No other collection methods are implemented. (For details, see the **ListBox**, **ComboBox**, and **Table** classes.)

To search row, column, or cell objects, use the **findObject** method.

The following example shows the use of the **itemObject** property.

```
bDelete_click(btn: Button input) updating;
vars
    prod    : Product;
    company : Company;
begin
```

```
        if listProducts.listIndex = -1 then
            app.msgBox("You must make a selection", "Error", 0);
            return;
        endif;
        beginTransaction;
            prod := listProducts.itemObject [listProducts.listIndex].Product;
            delete prod;
        commitTransaction;
    end;
```

When the items in a combo box or list box are determined by using the **listCollection** method or **displayRow** event method, the value of the **itemObject** property is automatically assigned to the collection entry associated with the list entry. This value cannot be changed.

itemPicture

Type: Picture array

Availability: Read or write at run time only

The **itemPicture** property allows individual items in a **ListBox** control or **ComboBox** control to be assigned a picture that is displayed just before the text and after any images drawn by using the **hasPictures**, **hasTreeLines**, and **hasPlusMinus** properties.

Setting the value of this property to **null** clears the picture.

Note If you click on the **itemPicture** property, the list item receives the **pictureDblClick** or **pictureClick** event, and the list item is not selected. Only when the list item text is clicked is the item selected.

itemPictureType

Type: Integer array

The **itemPictureType** property enables the type of picture of each item in a **ListBox** control or a **ComboBox** control to be obtained or set. This property contains a reference to an array of integer values with the same number of items as returned by the **listCount** method.

By default, the picture type of each entry in a list box is set automatically, according to whether it has subitems and is expanded or not.

The settings of the **itemPictureType** property are listed in the following table.

ComboBox or ListBox Class Constant	Value	Description
ItemPictureType_Closed	0	Entry with subitems closed
ItemPictureType_Open	1	Entry with subitems open
ItemPictureType_Leaf	2	Entry with no subitems

The automatic value can also be determined by using the **itemHasSubItems** or **itemExpanded** method.

If the user wants to manually control which picture is displayed, the picture type of an entry can be set. However, no automatic assignment of the picture is made for that entry from then on. This process could be used in situations where the subitems are not loaded until the entry is expanded.

The following examples show the use of the **itemPictureType** property.

```
// Change picture to the open book
foreach count in 1 to listBoxCustomer.listCount do
    listBoxCustomer.itemPictureType[count] := ListBox.ItemPictureType_Open;
endforeach;

loadListBox updating;
vars
    prod  : Product;
    count : Integer;
begin
    if app.myCompany.allProducts <> null then
        foreach prod in app.myCompany.allProducts do
            listInstances.addItem(prod.display);
            count := 1;
            if prod.logo <> null then
                listInstances.itemPictureType[count] :=
                                      ListBox.ItemPictureType_Leaf;
            endif;
            count := count + 1;
        endforeach;
    endif;
end;
```

itemSelected

Type: Boolean array

Availability: Read or write at run time only

The **itemSelected** property enables the selection status of each item in a **ListBox** control to be obtained or set. The **itemSelected** property contains a reference to an array of **Boolean** values with the same number of items as returned by the **listCount** method. Use this property when users can make multiple selections in a list box, to quickly check which items in a list are selected. You can also use this property from logic to select or deselect items in a list.

For details about making multiple selections in list boxes, see the **multiSelect** property.

If only one item is selected, you can use the **listIndex** property to get the index of the selected item. However, in a multiple selection, the **listIndex** property returns the index of the item contained within the focus rectangle, whether or not the item is actually selected.

When multiple items are currently selected, the value of the **listIndex** property is the last of the items selected. One or more items can be selected, with the value of the **listIndex** property being none of those items (for example, when you select an item, press the Shift key and select another item, then press the Ctrl key and remove the selection of one of the previously selected items).

itemText

Type: String array

Availability: Read or write at run time only

The **itemText** property enables you to access the text of an item in a **ListBox** or **ComboBox** control.

The item of the list box or the combo box control must have been added by using the **addItem** or **addItemAt** method, or by associating a collection with a list box or a combo box. The index of the first item is **1** and the index of the last item is **listCount** method.

Initially, list boxes and combo boxes contain an empty list. Changing the text of an item in a sorted list box or combo box causes the item to be sorted into its correct sorted position. Its new (or unchanged) position can be obtained by using the **newIndex** method.

Note The **text** property returns the same as the **itemText** property for the currently selected item.

The code fragment in the following example shows the use of the **itemText** property.

```
if listFaults.itemText[listFaults.listIndex] <> "" then
    bCloseFault.enabled := true;
endif;
```

language

Type: Integer[4]

Availability: Read or write at run time only

The **language** property of the **JadeTextEdit** control contains the programming language used in the text editor and selects the lexical analyzer that is used to calculate text styling and fold points.

The lexical analyzers for all currently Scintilla-supported languages are built into the **JadeTextEdit** control. However, only a small number of the most-common languages have the necessary text styles and keywords included in the global settings table.

The fully supported languages are represented by the **JadeTextEdit** class constants listed in the following table.

Constant	Value	Constant	Value
SCLEX_BASH	62	SCLEX_BATCH	12
SCLEX_CONF	17	SCLEX_CPP	3
SCLEX_CSS	38	SCLEX_DIFF	16
SCLEX_HTML	4	SCLEX_JADE	65
SCLEX_JAVA	65539	SCLEX_JAVASCRIPT	131075
SCLEX_MAKEFILE	11	SCLEX_PERL	6
SCLEX_PROPERTIES	9	SCLEX_PS	42
SCLEX_PYTHON	2	SCLEX_TEXT	1 (the default)
SCLEX_VB	8	SCLEX_VBSCRIPT	28
SCLEX_XML	5		

For more details, see "Supported Languages" under "Using the JadeTextEdit Control", earlier in this document.

For details about the full set of languages that are understood by the Scintilla-based **SciTE** text editor application, see http://scintilla.sourceforge.net/SciTEDoc.html.

languageId

Type: Integer

Availability: Read-only at any time

The **languageId** property of the **JadeEditMask** control contains the locale associated with the language identifier to be used by the control.

The default value of zero (**0**) indicates that the control uses the current locale of the user.

Set this property to a specific language identifier if you want to force the data entry to always use the locale associated with that language.

largeChange

Type: Integer

Availability: Read or write at any time

The **largeChange** property contains the amount of change (in pixels) to the **value** property in a **ScrollBar** control when the user clicks the area between the scroll box and scroll arrow.

You can specify any valid integer greater than the value of the **min** property and less than the value of the **max** property. By default, the **largeChange** property value is **100**.

The Windows environment automatically sets proportional scrolling increments for scroll bars on form windows, combo boxes, and list boxes, based on the amount of data in the control. For a scroll bar control, however, you must specify these increments.

Use the **largeChange** property to set scrolling increments appropriate to how the scroll bar is being used.

As scroll bars use the Windows scroll bar API calls that set the thumb size to a size that reflects the size of the scroll bar range and the value of the **largeChange** property, the larger the value of the **largeChange** property, the larger the thumb size.

The following example shows the use of the **largeChange** property.

```
buttonFillRight_click(btn: Button input) updating;
vars
    count    : Integer;
    start    : Time;
    iter     : Iterator;
    theArray : ObjectArray;
begin
    app.mousePointer  := self.MousePointer_HourGlass;
    start             := app.clock.Time;
    listBoxRight.clear;
    textBoxStart.text := null;
    iter              := app.myCompany.allProducts.createIterator;
    foreach count in 1 to listBoxRight.lines do
        iter.next(myProduct);
        listBoxRight.addItem(myProduct.name);
    endforeach;
    if theArray = null then
        create theArray transient;
    else
```

```
            theArray.clear;
        endif;
        app.myCompany.allProducts.copy(theArray);
        listBoxScrollBar.min         := 1;
        listBoxScrollBar.value       := 1;
        listBoxScrollBar.max         := theArray.size - listBoxRight.lines - 1;
        listBoxScrollBar.largeChange := (theArray.size/20).Integer;
    epilog
        labelRight.caption := "Time Taken := " & ((app.clock.Time -
                              start).Integer/1000).String & " Seconds";
        app.mousePointer:= self.MousePointer_Arrow;
    end;
```

Typically, you set the **largeChange** property in the JADE development environment Painter. You can also reset it in logic at run time when the scrolling increment must change dynamically.

Use the **max** and **min** properties to set the maximum and minimum ranges of a scroll bar control.

left

Type: Real

Availability: Read or write at any time

The **left** property of the **Window** class contains the distance between the internal left edge of an object and the left edge of the client area of the container (the non-border area).

When the parent is a **Form** class or a **BaseControl** or **Picture** control, the left position is also offset by the amount that the parent is scrolled. The **left** property for a form is always expressed in pixels.

Note If the value of the **left** property plus the value of the **width** property is greater than 32,767 pixels, the resulting window extents may be unpredictable.

For controls, the **left** property is expressed in units controlled by the **scaleMode** property of the parent of the control. The default value of the **scaleMode** property is pixels. (See also the **parentAspect** property.)

The value for the **left** property changes as the object is moved by the user or by logic. Changing the left or top dimension of a form does *not* cause a form **move** event.

The following examples show the use of the **left** property.

```
    pictureEnlarged_mouseMove(pict:   Picture input;
                              button: Integer;
                              shift:  Integer;
                              x, y:   Real) updating;
    begin
        if movePic = true then
            pictureEnlarged.move(pictureEnlarged.left + (x - moveX),
                                 pictureEnlarged.top + (y - moveY),
                                 pictureEnlarged.width, pictureEnlarged.height);
            pictureEnlarged.refreshNow;
        endif;
    end;

    setTableAndScroll();
    begin
```

```
        if rowsVisible - 1 < theCollectionSize then
            theTableScroll.height      := theTable.height;
            theTableScroll.max         := theCollectionSize;
            theTableScroll.largeChange := rowsVisible - 1;
            if theTableScroll.visible then
                theTableScroll.left    := theTable.left + theTable.width - 2;
            else
                theTableScroll.left    := theTable.left + theTable.width - 2;
                theTableScroll.value   := 1;
            endif;
        elseif theTableScroll.visible then
            theTable.width := theTable.width + theTableScroll.width;
            theTableScroll.visible := false;
        endif;
    end;
```

When multiple monitors are running on a workstation and a form is saved in the JADE Painter, the values of the **left** and **top** properties are converted to be relative to the top and left of the primary monitor.

leftColumn

Type: Integer

Availability: Read or write at run time

The **leftColumn** property contains the column that is displayed at the left edge of the non-fixed area of the current sheet of a **Table** control. This value may be decreased automatically by the control if lower values can still display the remainder of the columns.

The **leftColumn** property has no meaning if the display does not require a scroll bar.

Changing the **row** or **column** property does not change the rows or columns that are displayed, as these are changed only by the **leftColumn** and **topRow** properties.

The following example shows the use of the **leftColumn** property.

```
    convertPositionToColumn(xPos: Real): Integer updating;
    // Return the table column whose left and width positions cover the
    // x coordinate passed
    vars
        originalColumn : Integer;
        ix             : Integer;
    begin
        originalColumn := theTable.column;
        ix := theTable.leftColumn;
        while ix <= theTable.columns do
            theTable.column := ix;
            if theTable.positionLeft <= xPos then
                if xPos <= (theTable.positionLeft +
                            theTable.columnWidth[ix]) then
                    theTable.column := originalColumn;
                    return(ix);
                endif;
            endif;
            ix := ix + 1;
        endwhile;
        theTable.column := originalColumn;
        return(0);
    end;
```

leftIndent

Type: Integer

Availability: Read or write at any time

The **leftIndent** property of the **JadeRichText** control contains the distance (in pixels) between the left edge of the control and the left edge of the current selection or text added after the insertion point. If multiple paragraphs are selected and each has a different value, the property contains **ParagraphFormat_Undefined** (#80000000).

The default value of zero (**0**) indicates that the control is not indented.

lineWidth

Type: Integer

Availability: Read or write at any time

The **lineWidth** property of the **JadeRichText** control contains the maximum width (in pixels) of a line of text.

When this property is set to the default value of zero (**0**), the width of the line is the same as the width of the control width.

If the line width is greater than the width of the control, horizontal scrolling is enabled.

listIndex

Type: Integer

Availability: Not available at design time, read or write at run time

The **listIndex** property contains the index of the currently selected item in the **ComboBox** or **ListBox** control. The settings of the **listIndex** property are listed in the following table.

Value	Description
ItemNotFound (-1)	Indicates no item is currently selected (the default). For a combo box control, it also indicates the user has entered new text into the text box portion of the combo box.
n	A number indicating the index of the currently selected item.

The **text** property and **itemText** property **[list1.listIndex]** return the string for the currently selected item.

The first item in the list is **listIndex 1**, and the last entry is the same number of items as that returned by the **listCount** method.

Changing the **listIndex** property from logic does not generate a **click** event.

Setting the **listIndex** property from logic changes the portion of the list box that is displayed. (You can also use the **topIndex** property to control the entries that are shown to the user.)

When multiple items are currently selected, the value of the **listIndex** property is the last of the items selected. One or more items can be selected, with the value of the **listIndex** property being none of those items (for example, when you select an item, press the Shift key and select another item, then press the Ctrl key and remove the selection of one of the previously selected items).

For details about automatically controlling a **ComboBox** control assigned to a **Table** control as a **cellControl** property (for example, when performance is an issue when running in JADE thin client mode over a slow link), see the **Control** class **automaticCellControl** property.

The following example shows the use of the **listIndex** property.

```
listProducts_dblClick(listBox: ListBox);
vars
    pd : ProductDisplay;
begin
    if listProducts.itemText[listProducts.listIndex] <> "" then
        create pd;
        pd.fault := app.myCompany.allProducts.getAtKey(getProdNumber);
        pd.centreWindow;
        pd.showModal;
    endif;
end;
```

listObject

Type: Object

Availability: Not available at design time, read or write at run time

The **listObject** property contains a reference to the associated object of the currently selected item in the **ComboBox** or **ListBox** control.

Setting the **listObject** property selects the entry associated with the requested object. This property would normally be used only when a collection is associated with the combo box or list box.

The **listObject** property returns the **itemObject** property value of the currently selected list entry. The value that is returned is equivalent to:

```
obj := list1.itemObject[list1.listIndex];
```

If the value of the **listIndex** property is **-1**, a null object is returned.

Setting the **listObject** property value selects the first list entry where the **itemObject** property value equals the **listObject** property object. If there is no such list entry, an exception is raised. The value of the **listIndex** property is set to the selected list entry. When multiple items are currently selected, the value of the **listIndex** property is the last of the items selected. One or more items can be selected, with the value of the **listIndex** property being none of those items (for example, when you select an item, press the Shift key and select another item, then press the Ctrl key and remove the selection of one of the previously selected items).

Setting the **listObject** property is equivalent to:

```
foreach indx in 1 to list1.listCount do
    if obj = list1.itemObject[indx] then
        list1.listIndex := indx;
        return;
    endif;
endforeach;
```

Changing the **listObject** property (and therefore the **listIndex** property) from logic does not generate a **click** event. Setting the **listObject** property from logic does not change the portion of the list box that is displayed. Use the **topIndex** property to control the entries that are shown to the user.

The following examples show the use of the **listObject** property.

```
buttonFillLeft_click(btn: Button input) updating;
vars
    start : Time;
    count : Integer;
begin
    app.mousePointer       := self.MousePointer_HourGlass;
    start                  := app.clock.Time;
    count                  := app.myCompany.allProducts.size;
    listBoxLeft.listCollection(app.myCompany.allProducts, true, 0);
    listBoxLeft.listObject := listBoxLeft.itemObject [count];
    labelLeft.caption      := "Time Taken := " & ((app.clock.Time -
                              start).Integer/1000).String & " Seconds";
    app.mousePointer       := self.MousePointer_Arrow;
end;

    groupBoxWeb.visible        := false;
    website := listBox1.listObject.Customer.customerWebSite;
    if website <> null and currentSession <> null then
        groupBoxWeb.caption    := "Click to access Web Site";
        labelWebSite.caption   := "http://" & website;
        labelWebSite.hyperlink := "http://" & website;
    else
        groupBoxWeb.caption    := "No Web Site registered";
        labelWebSite.caption   := null;
        labelWebSite.hyperlink := null;
    endif;
...
```

listWidth

Type: Integer

Availability: Read or write at any time

The **listWidth** property contains the width (in pixels) of the drop-down list box portion of the **ComboBox** control. By default, the list is the same width as the combo box (that is, **listWidth = 0**).

Setting this property to a value greater than zero (**0**) sets the width of the drop-down list to that number of pixels. For example, if a combo box contains the following list, the combo box only needs to be wide enough to show the code value.

```
N - None
S - Single
M - Married
```

You can use the **listWidth** property to set the width of the drop-down list so that it displays the full text of each item in the list.

Note The **listWidth** property does not apply to a simple (**Style_Simple**) or to a spin (**Style_SpinBox**) style combo box. The property value is reset to zero (**0**) for combo boxes of these styles.

markerMargin

Type: Boolean

Availability: Read or write at any time

The **markerMargin** property of the **JadeTextEdit** control specifies whether the marker margin containing linemarks is displayed in the text editor.

The default value is **false**. When you set the value of this property to **true**, the default line margin width is **20** pixels. Use the **marker.margin.width** named attribute to examine and change the margin width. Clicking in this margin performs line selection

mask

Type: Integer

Availability: Read-only at any time

The **mask** property of the **JadeEditMask** control contains a concatenated series of symbols that define the characteristics of the editing requirement and the actions to be taken. The **mask** property enables you to define the edit mask:

- In the JADE Painter at application development time, by using the Properties dialog to define a mask of the **JadeEditMask** control selected on the Painter form, specifying the edit mask that you require for the control in the **mask** property on the **Specific** sheet of the dialog
- Dynamically at run time, by constructing it in your JADE application logic

Note A string of concatenated symbols cannot contain spaces between the symbols.

This property can be translated when the value of the **Schema** class **formsManagement** property is **FormsMngmt_Single_Multi** (**2**).

The valid edit mask symbols are listed in the following table. (Note that the **MMM** month abbreviation for dates in the **Symbol** column indicates the three-character month abbreviation. Although a locale may have more than three characters for the full abbreviation, JADE uses only the first three.)

Symbol	Description	Comments
A	Alpha (entry required).	**a** through **z** or **A** through **Z**.
a	Alpha character (entry optional).	**a** through **z** or **A** through **Z**.
9	Numeric (entry required).	**0** through **9**.
#	Numeric (entry optional).	**0** through **9**.
C	Character (entry required).	Any character.
c	Character (entry optional).	Any character.
ddMMyy	Date with 2-digit year and numeric month.	Example: **230901**. The date is expected in current locale order.
ddMMMyy	Date with 2-digit year and three-character alpha month abbreviation.	Example: **23Sep01**. The date is expected in current locale order and month name format.

Symbol	Description	Comments
ddMMyyyy	Date with 4-digit year and numeric month.	Example: **23092001**. The date is expected in current locale order.
ddMMMyyyy	Date with 4-digit year and three-character alpha month abbreviation.	Example: **23Sep2001**. The date is expected in current locale order and month name format.
dd<*d*>MM<*d*>yy	Date with 2-digit year and numeric month. Fields separated by the same delimiter (any character).	Example: **23/09/01**. The date is expected in the current locale order.
dd<*d*>MMM<*d*>yy	Date with 2-digit year and three-character alpha month abbreviation. Fields separated by the same delimiter (any character).	Example: **23/Sep/01**. The date is expected in current locale order and month name format.
dd<*d*>MM<*d*>yyyy	Date with 4-digit year and numeric month. Fields separated by the same delimiter (any character).	Example: **23/09/2001**. The date is expected in current locale order.
dd<*d*>MMM<*d*>yyyy	Date with 4-digit year and three-character alpha month abbreviation. Fields separated by the same delimiter (any character).	Example: **23/Mar/2001**. The date is expected in current locale order and month name format.
dd	Day of the month.	Separate day number field.
MM	Month number.	Separate month number field.
MMM	Three-character alphanumeric abbreviation of the month.	Separate month name field in current locale format.
yy	2-digit year.	Separate 2-digit year field.
yyyy	4-digit year.	Separate 4-digit year field.
hh	Hour.	Hour number in the range **0** through **23**.
mm	Minutes.	Minute number in the range **0** through **59**.
ss	Seconds.	Second number in the range **0** through **59**.
>	Uppercase next character.	Takes effect only if the next mask character is **A**, **a**, **C**, or **c**.
>>	Uppercase all characters following in this text box	Applies only to mask characters **A**, **a**, **C**, or **c**. field.
<	Lowercase next character.	Takes effect only if the next mask character is **A**, **a**, **C**, or **c**.
<<	Lowercase all characters following in this text box field.	Applies only to mask characters **A**, **a**, **C**, or **c**.
\	Treat the next character as a literal.	To insert a \ character into the mask, use \\. Similarly, to enter a specific locale-dependent character (for example, use **\\:** for **:**).
"	Start of a text string terminated by the next double quote **"** symbol.	Example: **">"**. To insert a quote character, use **\\"**. This symbol is the double quote character only.

Symbol	Description	Comments
H<*numeric*>	Height of this control (label or text box) field.	Numeric can be any length. Field terminated by the next non-numeric value.
L<*numeric*>	Left position of this control (label or text box) field.	Numeric can be any length. Field terminated by the next non-numeric value.
T<*numeric*>	Top position of this control (label or text box) field.	Numeric can be any length. Field terminated by the next non-numeric value.
W<*numeric*>	Width of a segment (label or text box) of the mask, except for the last segment, which always extends to the right-hand edge of the control.	Numeric can be any length. Field terminated by the next non-numeric value; for example, **W30"Date"@dd/MMM/yyyy**, where the label is created with a width of 30.
R	Field is right-aligned.	Default value is left-aligned.
.	Decimal point.	Actual character expected is locale-dependent.
-	Negative sign.	Actual character expected is locale-dependent.
,	Numeric separator.	Separator is automatically inserted into the numeric as the number is entered. The actual character is locale-dependent.
/	Date separator.	Actual character is locale-dependent.
:	Time separator.	Actual character is locale-dependent.
$	Currency symbol.	Actual character is locale-dependent.
|	List of valid characters that can be entered for the character position.	For example: **'A|abc|'** means only **a**, **b**, or **c** can be entered into the alpha field. Can only follow edit mask types **A**, **a**, **C**, or **c**.
[	Start of numeric range for previous **#** or **9** numeric field. Must be an integer type field.	For example: **'##9[0-255]'** means the 3-digit numeric field must be in the range 0 through 255.
]	End of numeric range field else a literal if there was no corresponding **[** symbol.	See above (that is, the comments for the **[** symbol).
@	End of current label or text box. A new field is begun. Previous field is a label if it consists only of literals.	For example, **'999@-@999'** to define two numeric fields split by a dash character label.
Any other symbol	Treated as a literal.	For details, see the first item in the following list.

When using the **JadeEditMask** class mask property, note the following points.

- The Windows Control Panel setting is used to convert a two-digit year into a four-digit year for a two-digit edit mask year of **'yy'** when the value of the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is set to **true**. By default, years:
  - 00 through 29 become 2000 through 2029
  - 30 through 99 become 1930 through 1999

If the value of the **EnhancedLocaleSupport** parameter is **false** (the default), the year is calculated using the current century.

- As JADE reserves the right to implement additional features in the future that will give special meaning to other symbols, it is better to enclose any text box literal that is not in the previous table in double quote symbols (that is, **""**) or place the **** backslash symbol in front of the literal, to avoid future conflict.
- Having a mask does not guarantee that the data is valid, as a user might enter only part of the data for a field and never complete the rest. For more details, see the **isValid** and **isEmpty** methods and the **validate** event method.
- The user cannot create a date with an invalid day number and month combination. For example, if a user enters **31** and then attempts to enter a month of **4** or **Jun**, the entry is rejected.
- For a date that has an abbreviated month, the entered characters must appear somewhere in a month name for the month to be valid for a locale-dependent month.
- An invalid day number cannot be entered for a month that has been entered, with the exception of 29th February in leap years. The validity of the leap year cannot be known until entry of the year is completed. For example, entering **29/02/201** is not rejected because the completion of the year is still uncertain (that is, the final year value has yet to be entered).
- Two-digit year fields must have both digits entered before they are considered complete. Similarly, four-digit year fields must have all four digits entered before they are considered complete.
- For dates with alphanumeric month abbreviations, the first character of the month is always converted to uppercase (for example, **aug** is converted to **Aug**).
- If the mask has a label field in the control, by default the text of that label is sized to fit.

  The next field is displayed directly after the label text, which is drawn transparently on the **JadeEditMask** control at paint time, by using the font of the control.

  Areas of the **JadeEditMask** control that are not text boxes are drawn using the background color of the parent.
- Automatically sized text box fields are created with a size that can accommodate the largest possible text entry in the field. This is necessary because a text field with an edit mask of **'aaa'**, for example, with text of **WWW** takes up considerably more space than a field with **iii** text. In addition, because the prompt character requires more space than the character for which it is prompting, the size can be larger than required.
- Tabbing or setting the focus to a **JadeEditMask** control gives the focus to the first defined text box that can have data entered. For a control with multiple text boxes, the Tab key operates as though those text boxes were inserted between the **JadeEditMask** control and the next control in the tab order of the form (in order of their definition).
- Creating a text box containing all literal values is valid and is treated effectively as read-only. These text boxes, which cannot have values entered, are ignored when focus is assigned and when tabbing between text boxes, unless the control does not contain any text box field that can have data entered.
- For numeric fields, the end of the field is defined as the first non-numeric edit character, an optional numeric value following a mandatory numeric value, or an optional numeric value following a mandatory decimal place. For example, **'#99#9'** defines the **##9** and **#9** numeric fields while **'#99.9#9'** defines the **#99.9** and **#9** numeric fields.
- Do not use right-aligned text box fields for anything other than fields that accept only numeric data, as the caret remains at the same position. If your field has embedded literal characters, the arrow key would need to be used to position the caret at the next entry segment.

- If no mask is defined, the control acts as a normal **TextBox** control.

max

Type: Integer

Availability: Read or write at any time

The **max** property of the **ScrollBar** control contains a scroll bar position maximum **value** property setting when the scroll box is in its lowest position (the vertical scroll bar control) or farthest right position (horizontal scroll bar control). You can specify any valid integer. The default setting is **32,767**.

The Windows environment automatically sets ranges for scroll bars proportional to the contents of forms, combo boxes, and list boxes. For a horizontal scroll bar control, however, you must specify these ranges.

Use the **max** property to set a range appropriate to how the scroll bar control is used; for example, as an input device or as an indicator of speed or quantity. Typically, you set the **max** property in the JADE development environment. You can also set it in logic at run time, if the scrolling range must change dynamically.

The code fragment in the following example shows the use of the **max** property.

```
app.myCompany.allProducts.copy(theArray);
listBoxScrollBar.min         := 1;
listBoxScrollBar.value       := 1;
listBoxScrollBar.max         := theArray.size - listBoxRight.lines - 1;
listBoxScrollBar.largeChange := (theArray.size/20).Integer;
```

Use the **largeChange** property to set the maximum scrolling increments for a scroll bar control.

If the **max** property is set to a value less than the value of the **min** property, the value of the **max** property is then set at the farthest left or highest position of a horizontal or vertical scroll bar, respectively.

maxButton

Type: Boolean

Availability: Read or write at any time

The **maxButton** property specifies whether a form has a **Maximize** button.

The settings of the **Form** class **maxButton** property are listed in the following table.

Value	Description
true	The form has a **Maximize** button (the default).
false	The form does not have a **Maximize** button.

A **Maximize** button enables users to enlarge a form window to full-screen size.

Setting the value of the **maxButton** property to **true** causes a form with the **borderStyle** property set to **BorderStyle_None** (**0**) to adopt a **borderStyle** value of **BorderStyle_Single** (**1**).

The form also displays a caption area, regardless of whether the form caption is empty. This property should not be changed when an MDI child form is maximized.

A **Maximize** button automatically becomes a **Restore** button when a window is maximized. Minimizing or restoring a window automatically changes the **Restore** button back to a **Maximize** button.

Maximizing a form at run time generates a **resize** event. The **windowState** property reflects the current state of the window. If you set the **windowState** property to **WindowState_Maximized** (**2**), the form is maximized independently of whatever settings are in effect for the **maxButton** and **borderStyle** properties.

maximumHeight

Type: Integer

Availability: Read or write at any time

The **maximumHeight** property of the **JadeDockBase** class contains the maximum height (in pixels) of the control. Use this property if you want to prevent users from resizing the control over the specified height. (For details about setting a maximum width, see the **maximumWidth** property.)

The default value of this property is zero (**0**), with values in the range zero (**0**) through **32767** pixels permitted. If the property value is zero (**0**), the property is ignored and has no effect.

The **maximumHeight** property applies only to dock controls that are aligned horizontally (that is, when the **alignContainer** property value is set to **AlignContainer_AllHorizontal**, **AlignContainer_Bottom**, or **AlignContainer_Top**); otherwise the property value is ignored.

The **maximumHeight** property controls the maximum height of the control when:

- The user drags the horizontal resize bar below the control
- Assigning a height to the control from logic
- Assigning a height to the control when affected by a parent height resize
- Assigning a height to the control when affected by a height change of another control that is aligned horizontally

The value of the **maximumHeight** property cannot be less than the value of the **minimumHeight** property unless the **maximumHeight** property is set to the default value of zero (**0**). If the values of the **maximumHeight** and **minimumHeight** properties are the same, no resize bar is displayed below the control, even if the value of the **showResizeBar** property is **true**.

The value of the **maximumHeight** property has no impact on the allowed sizes of the parent form on which it appears.

Applies to Version: 2016.0.01 and higher

maximumWidth

Type: Integer

Availability: Read or write at any time

The **maximumWidth** property of the **JadeDockBase** class contains the maximum width (in pixels) of the control. Use this property if you want to prevent users from resizing the control over the specified width. (For details about setting a maximum height, see the **maximumHeight** property.)

The default value of this property is zero (**0**), with values in the range zero (**0**) through **32767** pixels permitted. If the property value is zero (**0**), the property is ignored and has no effect.

The **maximumWidth** property applies only to dock controls that are aligned vertically (that is, when the **alignContainer** property value is set to **AlignContainer_AllVertical**, **AlignContainer_Left**, or **AlignContainer_ Right**); otherwise the property value is ignored.

The **maximumWidth** property controls the maximum width of the control when:

- The user drags the vertical resize bar at the right of the control
- Assigning a width to the control from logic
- Assigning a width to the control when affected by a parent width resize
- Assigning a width to the control when affected by a width change of another control that is aligned vertically

The value of the **maximumWidth** property cannot be less than the value of the **minimumWidth** property unless the **maximumWidth** property is set to the default value of zero (**0**). If the values of the **maximumWidth** and **minimumWidth** properties are the same, no resize bar is displayed on the right of the control, even if the value of the **showResizeBar** property is **true**.

The value of the **maximumWidth** property has no impact on the allowed sizes of the parent form on which it appears.

Applies to Version: 2016.0.01 and higher

maxLength

Type: Integer

Availability: Read or write at any time

The **maxLength** property contains the maximum length of text that can be entered into a **TextBox** control or the text box portion of a **ComboBox** control, to limit the text entry. (For details about automatically moving focus to the next control in the tab order of a form when a user enters the final character into a text box at the maximum text position, see the **TextBox** class **autoTab** property.)

For a **Table** control, the **maxLength** property contains the amount of text that can be entered into a cell that has the **inputType** property set to **InputType_TextBox** (**2**), **InputType_TextNumeric** (**4**), or **InputType_SignedNumeric** (**6**).

The **Table** control **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property determines whether the **maxLength** property for the current sheet, cell, column, or row is being accessed.

The settings of the **maxLength** property are listed in the following table.

Value	Description
0	No limit on how much text can be entered except for the limit imposed by Windows
n	Maximum number of characters that can be entered in the text box

Setting the value of the **maxLength** property for a text box control is rejected when the **dataType** property of a text box control is set to a numeric type when the current text in the text box does not conform to the rules defined by the current **dataType**, **decimals**, and **maxLength** properties.

The limit is not imposed on changes to the text made by logic.

Note Extra room should be assigned for a text box when the **dataType** property allows decimals (for the '.' character) and a signed numeric (for the '**-**' sign).

An Enter key press counts as two characters.

The **maxLength** property of the **JadeRichText** control contains the maximum number of characters, including spaces, that can be entered or pasted into the control. The default value of zero (**0**) indicates a pre-imposed limit of **65,534** for the maximum number of characters that can be entered or pasted. (Note that an embedded object counts as one character long.)

Note **JadeRichText** controls can contain formatted data and embedded content making it expensive to check whether the amount of data exceeds the value of the **maxLength** property for large amounts of text.

A value of the **maxLength** property less than or equal to **256** limits is an absolute limit.

A value of the **maxLength** property greater than **256** is only an approximate limit. To avoid making an assignment to a fixed-length **String** attribute that raises a 1035 "String too long" exception, first determine the length of the string, as shown in the following code fragment.

```
    num := jadeRichText.text.length;
```

mdiChild

Type: Integer

Availability: Read or write at development time, read-only at run time

The **mdiChild** property of the **Form** class specifies whether a form is displayed as an MDI child form inside an MDI frame form. Use this property when creating a Multiple Document Interface (MDI) application.

At run time, forms with this property set to **MdiChild_IsMdi** (2) are displayed inside an MDI form. An MDI child form can be maximized, minimized, and moved inside the parent MDI form (if the **minButton** and **maxButton** property values are **true**).

The settings of the **mdiChild** property are listed in the following table.

Form Class Constant	Value	Description
MdiChild_UseAppDefault	0	Use the value of the **defaultMdi** property for the application to decide whether a form create results in an MDI form or a standalone form (the default)
MdiChild_NotMdi	1	Form create results in a non-MDI form
MdiChild_IsMdi	2	Form create results in an MDI form

The **mdiChild** property can then be used to provide the user with the choice of running the application as MDI forms when forms that do not specifically have to be MDI forms are assigned the default **mdiChild** property value.

Setting the application preference at startup time then runs those forms as MDI or standalone forms.

Set this property to a non-default value in the JADE development environment only if it is specifically required to run MDI or standalone forms. For example, forms shown modally must have a resulting **mdiChild** property value of **MdiChild_NotMdi** (**1**), otherwise the form could be created as MDI and then the modal status is ignored, as it cannot be converted after creation.

When working with MDI child forms, keep the following in mind.

- At run time, when an MDI child form is given the focus, its caption is combined with that of the parent frame. Its menu also replaces the MDI form menu of the parent.
- In the JADE development environment, an MDI child form is displayed like any other form, as the form is displayed inside the parent form only at run time.

- MDI child forms cannot be modal. The **showModal** method on a form declared as an MDI child form raises an exception. For more details, see "Windows Events and JADE Events", later in this document.
- The Windows environment controls the placement of MDI child forms unless you specifically set the placement in the **load** event of the form.
- The MDI frame form is automatically loaded for the first MDI child. If the MDI frame form is the default (supplied by **jade.exe**), it is unloaded when its last MDI child is unloaded.

mdiClientScrollHorzPos

Type: Integer

Availability: Read or write at run time only

The **mdiClientScrollHorzPos** property of the **Form** class contains the horizontal scroll positions of the MDI client windows of a form. When a form is built as an MDI frame, it also automatically creates a child client window that covers the non-border area of the frame. Child MDI forms are placed inside this client window. (An MDI frame is a special type of form that can contain one or more MDI child forms.)

If an MDI child form does not fit within the bounds of this client window, scroll bars are automatically added to the MDI client window.

The scroll positions are in pixels and are always relative to zero (**0**). The scroll range is set automatically by Windows, so that all parts of all child forms scroll into view.

Notes The zero left position is changed by Windows when the MDI client window is scrolled if there is unused space to the left of the forms displayed. The scroll bars are removed if the displayed forms fit within the visible MDI client window.

Windows may adjust any value that is set (rounded to internal increments).

This property can be accessed from both the MDI Frame and any MDI child forms.

Setting a scroll position to a value less than zero (**0**) results in **0**. Setting a value greater than the maximum scroll range value results in the maximum position being selected.

mdiClientScrollVertPos

Type: Integer

Availability: Read or write at run time only

The **mdiClientScrollVertPos** property of the **Form** class contains the vertical scroll positions of the MDI client windows of a form. When a form is built as an MDI frame, it also automatically creates a child client window that covers the non-border area of the frame. Child MDI forms are placed inside this client window. (An MDI frame is a special type of form that can contain one or more MDI child forms.)

If an MDI child form does not fit within the bounds of this client window, scroll bars are automatically added to the MDI client window.

The scroll positions are in pixels and are always relative to zero. The scroll range is set automatically by Windows, so that all parts of all child forms scroll into view.

Notes The zero (**0**) top position is changed by Windows when the MDI client window is scrolled if there is unused space above the forms displayed. The scroll bars are removed if the displayed forms fit within the visible MDI client window.

Windows may adjust any value that is set (rounded to internal increments).

This property can be accessed from both the MDI Frame and any MDI child forms.

Setting a scroll position to a value less than zero (**0**) results in **0**. Setting a value greater than the maximum scroll range value results in the maximum position being selected.

mdiFrame

Type: Boolean

Availability: Read or write at any time, but changing the value after the form has been built has no impact

The **mdiFrame** property of the **Form** class causes the form to always be built as an MDI frame form when the value is set to **true**, regardless of the setting of the **mdiFrame** property of the **Application** class. An MDI frame is a special type of form that can contain one or more MDI child forms.

By default, MDI child forms are placed in a default MDI frame supplied automatically by JADE.

The **mdiFrame** property of the **Application** class can be used to control which form is the next MDI frame. Any MDI child forms built after this point are placed in that MDI frame. If the MDI frame form is not already active as an MDI frame, it is automatically created and displayed when the MDI child form is created and displayed. The application can have any number of currently active MDI frames.

Notes An MDI frame automatically creates a child client window that covers the non-border area of the frame when the first MDI child form is created. The child MDI forms are placed inside this client window. If the MDI frame is defined with controls, the child client window is automatically placed into an empty area of the form.

An MDI frame form has sizable borders, regardless of the **borderStyle** property value for the form in Painter.

The **moveMdiClient** method for forms enables client windows to be positioned as required; for example, below toolbars and above status lines. Call the **moveMdiClient** method from the **resize** method of the form.

mdiPinned

Type: Boolean

Availability: Read and write at run time

The **mdiPinned** property of the **Form** class specifies whether the MDI child form is pinned. The default value is **false**.

Applies to Version: 2020.0.01 and higher

mediaData

Type: Binary

Availability: Read or write at any time

The **mediaData** property contains the data associated with the current medium of the **MultiMedia** class. The default value is **null**.

If the **mediaName** property is set at development time and represents a file, the **mediaData** binary property holds the contents of that file, and the **mediaName** property is no longer relevant, as the contents of the file have all been copied to the **mediaData** property.

Setting the value of the **mediaData** property to a binary value is equivalent to setting the **mediaName** property to a file containing that binary data. Any existing **mediaData** or **mediaName** property value is discarded. Each available device driver examines the data in turn, until it is recognized as being of a format handled by that driver. It is then loaded ready for playing.

If the data is not recognized or accepted by any device driver, an exception is raised.

If the value of the **mediaName** property represents a device type (for example, **cdaudio**) and not a file, this property returns **null**.

If the value of the **mediaName** property represents a file, the complete file is read and loaded into the **mediaData** binary property and then returned to the JADE logic. You should therefore avoid accessing the **mediaData** property unless necessary, particularly for large files. (For example, if you have a 600M byte audio file, you should use the **mediaName** property so that the complete file does not have to be loaded into the JADE database but can be accessed from the device.)

Caution JADE handles only binary data that has a length less than the maximum database cache size, so an attempt to store large files in the database may fail. To cover situations where the data is copied by logic, ensure that only files of a size less than half the cache size are stored.

mediaName

Type: String

Availability: Read or write at any time

The **mediaName** property contains the name of the medium currently installed in the **MultiMedia** class. The medium name can be the name of a data file or the name of a device, as shown in the code fragments in the following examples.

```
mm.mediaName := "c:\media\avi\intro.avi";
```

```
mm.mediaName := "cdaudio";
```

The default value is null (**""**). See also the **openDialog** method.

The device types that can be accessed are listed in the following table.

Device Type	Description
animation	Animation device
cdaudio	Audio CD player
dat	Digital audio tape player
digitalvideo	Digital video in a window
overlay	Overlay device (analog video in a window)
scanner	Image scanner
sequencer	MIDI sequencer
vcr	Videotape recorder or player

Device Type	Description
videodisc	Videodisc player
waveaudio	Audio device that plays digitized waveform files

The file types that can be handled depend on the software that is installed on the workstation, including the types listed in the following table.

File Type	Description
wav	Sound files
mid	MIDI sequence
avi	Video with or without sound
mp3	MPEG-1 Audio Layer-3
mp4	MPEG Layer-4 Audio
mpg	MPEG video or audio

If the **useDotNetVersion** property is set to **true** from version 2018.0.01 and higher, the value of the **mediaName** property can be a URL; for example, http://hostName/images/introduction.mp4. For details about using the MP4 version of a control, see the **useDotNetVersion** property.

For some types, specific software (device drivers) is required.

Setting the **mediaName** property causes the current medium that is being played in the control to be discarded. The appropriate device driver is selected, and the medium is readied for playing. If the **mediaName** property was set at development time and represents a file, the contents of the file are copied to the **mediaData** property, and the **mediaName** property is no longer relevant.

If the **mediaName** property is set at run time and represents a file, the file is accessed directly by the device driver. The **mediaData** property is not set to the contents of that file unless it is accessed by logic. You should therefore avoid accessing the **mediaData** property unless it is necessary, particularly for large files.

Caution JADE handles only binary data that has a length less than the maximum database cache size, so an attempt to store large files in the database may fail. To cover situations where the data is copied by logic, ensure that only files of a size less than half the cache size are stored.

min

Type: Integer

Availability: Read or write at any time

The **min** property of the **ScrollBar** class contains a scroll bar position minimum **value** property setting when the scroll box is in its highest position (the vertical scroll bar control) or farthest left position (horizontal scroll bar control).

Specify any valid integer. The default setting is zero (**0**).

The Windows environment automatically sets ranges for scroll bars proportional to the contents of forms, combo boxes, and list boxes. For a horizontal scroll bar control, however, you must specify these ranges. Use the **min** property to set a range appropriate to how the scroll bar control is used; for example, as an input device or as an indicator of speed or quantity.

Typically, you set the **min** property in the JADE development environment. You can also set it in logic at run time, if the scrolling range must change dynamically.

Use the **smallChange** property to set the minimum scrolling increments for a scroll bar control.

If the **max** property is set to a value less than the value of the **min** property, the value of the **max** property is then set at the farthest left or highest position of a horizontal or vertical scroll bar, respectively.

minButton

Type: Boolean

Availability: Read or write at any time

The **minButton** property of the **Form** class specifies whether a form has a **Minimize** button. The settings of the **minButton** property are listed in the following table.

Value	Description
true	The form has a **Minimize** button (the default).
false	The form does not have a **Minimize** button.

A **Minimize** button enables users to shrink a form window to an icon.

Setting the value of the **minButton** property to **true** causes a form with the **borderStyle** property set to **BorderStyle_None** (0) to adopt a **borderStyle** value of **BorderStyle_Single** (1). The form also displays a caption area, regardless of whether the form caption is empty.

This property should not be changed when an MDI child form is maximized.

Notes Shrinking a form to an icon at run time generates a **resize** event. The **windowState** property reflects the current state of the window.

If you set the **windowState** property to **WindowState_Maximized** (2), the form is maximized independently of the settings that are in effect for the **maxButton** and **borderStyle** properties.

minimumHeight

Type: Integer

Availability: Read or write at any time

The **minimumHeight** property of the **Form** class contains the minimum height (in pixels) of the form. Use this property if you want to prevent users from resizing the form below the specified height. (For details about setting a minimum width, see the **minimumWidth** property.)

Notes The minimum height is retained even if logic attempts to set the height of the form below the equivalent minimum height.

Windows imposes a minimum height and width so that the form caption buttons are always visible. The effective minimum sizes are therefore the maximum of the size imposed by Windows and the **minimumHeight** and **minimumWidth** values.

The **minimumHeight** property of the **JadeDockBase** class (from version 2016.0.01) contains the minimum height (in pixels) of the control. Use this property if you want to prevent users from resizing the control below the specified height. (For details about setting a minimum width, see the **minimumWidth** property.)

The default value of this property is zero (**0**), with values in the range zero (**0**) through **32767** pixels permitted. If the property value is zero (**0**), the property is ignored and has no effect.

The **minimumHeight** property applies only to dock controls that are aligned horizontally (that is, when the **alignContainer** property value is set to **AlignContainer_AllHorizontal**, **AlignContainer_Bottom**, or **AlignContainer_Top**); otherwise the property value is ignored.

The **minimumHeight** property controls the minimum height of the control when:

- The user drags the horizontal resize bar below the control
- Assigning a height to the control from logic
- Assigning a height to the control when affected by a parent height resize
- Assigning a height to the control when affected by a height change of another control that is aligned horizontally

The value of the **minimumHeight** property cannot be greater than the value of the **maximumHeight** property unless the **maximumHeight** property is set to the default value of zero (**0**). If the values of the **minimumHeight** and **maximumHeight** properties are the same, no resize bar is displayed below the control, even if the value of the **showResizeBar** property is **true**.

The value of the **minimumHeight** property of the **JadeDockBase** class has no impact on the allowed sizes of the parent form on which it appears.

minimumWidth

Type: Integer

Availability: Read or write at any time

The **minimumWidth** property of the **Form** class contains the minimum width (in pixels) of the form. Use this property if you want to prevent users from resizing the form below the specified width. (For details about setting a minimum height, see the **minimumHeight** property.)

Notes The minimum width is retained even if logic attempts to set the width of the form below the equivalent minimum width.

Windows imposes a minimum height and width so that the form caption buttons are always visible. The effective minimum sizes are therefore the maximum of the size imposed by Windows and the **minimumHeight** and **minimumWidth** values.

The **minimumWidth** property of the **JadeDockBase** class (from version 2016.0.01) contains the minimum width (in pixels) of the control. Use this property if you want to prevent users from resizing the control below the specified width. (For details about setting a minimum height, see the **minimumHeight** property.)

The default value of this property is zero (**0**), with values in the range zero (**0**) through **32767** pixels permitted. If the property value is zero (**0**), the property is ignored and has no effect.

The **minimumWidth** property applies only to dock controls that are aligned vertically (that is, when the **alignContainer** property value is set to **AlignContainer_AllVertical**, **AlignContainer_Left**, or **AlignContainer_Right**); otherwise the property value is ignored.

The **minimumWidth** property controls the minimum width of the control when:

- The user drags the vertical resize bar at the right of the control
- Assigning a width to the control from logic
- Assigning a width to the control when affected by a parent width resize
- Assigning a width to the control when affected by a width change of another control that is aligned vertically

The value of the **minimumWidth** property cannot be greater than the value of the **maximumWidth** property unless the **maximumWidth** property is set to the default value of zero (**0**). If the values of the **minimumWidth** and **maximumWidth** properties are the same, no resize bar is displayed on the right of the control, even if the value of the **showResizeBar** property is **true**.

The value of the **minimumWidth** property of the **JadeDockBase** class has no impact on the allowed sizes of the parent form on which it appears.

modalResult

Type: Integer

Availability: Read or write at run time

The **modalResult** property of the **Form** class contains the returned value for the **showModal** method call in runtime forms that are initiated as modal. You can write logic to return a value that informs the calling method of the result of the modal process. This property is set to zero (**0**) for any form when it is created.

The setting of this property has meaning only for a form initiated as modal. The final value of the property when the modal form is unloaded is returned by the **showModal** method call. No value is assigned automatically after creation or (for example), after pressing the **Cancel** button.

modified

Type: Boolean

Availability: Read or write at run time only

The **modified** property of the **JadeTextEdit** control specifies whether the text in the control has been modified. The value of this property is set to **false** when the text is unchanged. When the value is set to **false**, the undo history is not discarded and the next editor change causes the **firstChange** event.

To override the editor state, set this property to **true**. All subsequent reads of the **modified** property then return **true** until the value of the property is set to **false**, the **text** property is set, or the **emptyUndoBuffer** method is called.

mouseCursor

Type: Binary (picture)

Availability: Read or write at any time, but the setting is ignored at development time

The **mouseCursor** property of the **Window** class contains a cursor that is not provided by the system to display when the mouse is over a form or control. This process is achieved by assigning a cursor (by using the **Application** class **loadPicture** method (**app.loadPicture**) to load the **mouseCursor** property and then setting the **mousePointer** property to **MousePointer_Cursor**).

The definition of a mouse cursor includes the position of the hotspot. The tool that you use to create a cursor should include the ability to define the hotspot. Icons used as cursors default to the top left position because an icon has no hotspot.

mousePointer

Type: Integer

Availability: Read or write at any time, but the setting is ignored at development time

The **mousePointer** property of the **Window** class contains the type of mouse pointer that is displayed when the mouse is over a specific part of a form or control at run time. The **mousePointer** property controls the shape of the mouse pointer.

Use this property when you want to indicate changes in functionality as the mouse pointer passes over controls on a form or control. Use the **MousePointer_HourGlass** (11) value to indicate that the user should wait for a process or operation to finish. To restore the previous behavior of the mouse pointer, set the **mousePointer** property value to **MousePointer_Default** (0).

If the **mousePointer** property of the **Application** class is set to a non-zero value, the **mousePointer** property value of the form or control is ignored. The application **mousePointer** property enables the mouse pointer to be set globally for the whole application; for example, as an hourglass.

The settings of the **mousePointer** property are listed in the following table.

Window Class Constant	Value	Description
MousePointer_Default	0	Default value determined by the current window (for example, an arrow or hourglass).
MousePointer_Arrow	1	Arrow (↖).
MousePointer_Cross	2	Cross (+ cross-hair pointer).
MousePointer_IBeam	3	I-Beam (I).
MousePointer_Cursor	4	User-defined cursor not provided by the system. (See the **Window** class **mouseCursor** property.)
MousePointer_Size	5	Size (✥ four-pointed arrow pointing north, south, east, west).
MousePointer_NESW	6	Size NE SW (⤢ double arrow pointing north east and south west).
MousePointer_NS	7	Size N S (↕ double arrow pointing north and south).
MousePointer_NWSE	8	Size NW SE (⤡ double arrow pointing north west and south east).
MousePointer_WE	9	Size W E (↔ double arrow pointing west and east).
MousePointer_UpArrow	10	Up arrow (↑).
MousePointer_HourGlass	11	Hourglass (wait) ⌛.
MousePointer_NoDrop	12	No drop (⃠).
MousePointer_Drag	13	Standard JADE drag cursor (🗀).

Window Class Constant	Value	Description
MousePointer_HorizontalLine	14	Cursor used to drag a horizontal line ( ).
MousePointer_VerticalLine	15	Cursor used to drag a vertical line ( ).
MousePointer_HandPointing	16	Cursor used for hyperlinks ( ).

The following example shows the use of the **mousePointer** property.

```
bCancel_dragDrop(btn:  Button input;
                 win:  Window input;
                 x, y: Real) updating;
begin
    mousePointer       := MousePointer_NoDrop;
    bCancel.bubbleHelp := "Cannot Drop Here";
    doWindowEvents(2000);
    mousePointer       := MousePointer_Arrow;
    bCancel.bubbleHelp := null;
end;
```

multiSelect

Type: Integer

Availability: Read or write at any time

The **multiSelect** property determines whether a user can make multiple selections in a **ListBox** control and the way in which multiple selections can be made.

The settings of the **multiSelect** property are listed in the following table.

ListBox Class Constant	Value	Description
MultiSelect_None	0	Multiple selection is not allowed (the default).
MultiSelect_Simple	1	Simple multiple selection. A click or the space bar selects or deselects an item in the list. (Arrow keys move the preselect focus.)
MultiSelect_Extended	2	Extended multiple selection. Shift+click or Shift+arrow key extends the selection from the previously selected item to the current item. Ctrl+click selects or deselects an item in the list. Ctrl+up arrow, down arrow, Home, End, Page Up, or Page Down key changes the current list entry but does not alter the selected status of any entry. Ctrl+space bar selects or deselects an item in the list.

When multiple items are currently selected, the value of the **listIndex** property is the last of the items selected. One or more items can be selected, with the value of the **listIndex** property being none of those items (for example, when you select an item, press the Shift key and select another item, then press the Ctrl key and remove the selection of one of the previously selected items).

Note You must use the Ctrl key or Shift key to select multiple items in a list box on a Web form, as HTML does not support the **MultiSelect_Simple** functionality. Multiple selections are therefore regarded as extended multiple selections (that is, **MultiSelect_Extended**).

name

Type: String[100]

Availability: Read or write at design time, read-only at run time

The **name** property of the **Window** class contains the name used in logic to identify an application, form, control, or menu item object.

The default name for new objects is the kind of object plus a unique integer. For example, the first new form is **Form1**, and the third text box you create on a form is **textBox3**. For a form defined as a Web page, the **name** property contains the name of the Web page.

Applications and forms are defined in the JADE database as classes, and the first letter is converted to an uppercase character.

Controls and menu items are defined in the JADE database as properties, and the first character of the name is converted to a lowercase character.

A **name** property of an object must start with a letter, with a maximum length of 100 characters for a form and 86 characters for a control. This property can include numbers and underscore characters, but it cannot include punctuation symbols or spaces. Forms cannot have the same name as another form or global object, such as the application name or predefined JADE classes; for example, **Image**.

Subclassed forms cannot have controls or menus with the same name as a control on a superclass of the form.

Note Although JADE uses the **name** property as the default value for the **caption** and **text** properties, changing one property does not affect the others.

nameSeparator

Type: String[1]

Availability: Read or write at any time

The **nameSeparator** property contains the item delimiter string used when accessing the **ListBox** control **itemFullName** method. The default value is the backslash character (\). A **null** value is treated as the default backslash character (\).

Accessing the **itemFullName** method of an entry in the **ListBox** control returns a string consisting of the text of all parents of the entry concatenated with its own text entry. These level names are separated by the value of the **nameSeparator** property.

noPrefix

Type: Boolean

Availability: Read or write at any time

The **noPrefix** property of the **Label** class enables you to control whether the character following a single ampersand (**&**) is underlined.

The default value of **false** indicates that the first character of a label that is proceeded by a single **&** character is underlined. If you set the property value to **true**, the label displays the **&** character but with no underline attribute applied to the following character.

Applies to Version: 2016.0.01 and higher

oleObject

Type: OleObject

Availability: Read or write at any time

The **oleObject** property of the **OleControl** class contains a reference to an OLE object.

Use the **copy** method of the **OleObject** class to copy another object into or out of the control to or from the database. Use the **loadFromDB** method to access the contents of the OLE object.

The code fragment in the following example shows the use of the **oleObject** property.

```
foreach ole in ReviewOLEObj.instances do
    count := 1 + count;
    if count = 1 then
        oleReview1.oleObject.copy(ole);
        oleReview1.loadFromDB;
    elseif count = 2 then
        oleReview2.oleObject.copy(ole);
        oleReview2.loadFromDB;
    elseif count = 3 then
        oleReview3.oleObject.copy(ole);
        oleReview3.loadFromDB;
    endif;
endforeach;
```

See also the **OleObject** class **getData** and **setData** methods.

parameters

Type: String

The **parameters** property of the **WebJavaApplet** class contains parameters for the applet.

Use this property to specify one or more parameters for the compiled Java applet (specified in the **appletName** property) that are to be inserted into the generated HTML for a Web-enabled session at run time. Separate each parameter key and value with a tab; for example:

```
"param1" & Tab & "25,50,fixed,enable" & Tab & "param2" & Tab &
"c:\pics\JadeIcon.png"
```

parent

Type: Window

Availability: Assigned by Painter at development time, read or write at run time

The **parent** property of the **Control** class contains a reference to the direct parent of the control. This direct parent is either the form or another control. This property returns the object of the parent form or control. Using this object can then access the properties of a window. If the parent of the control is the form, the parent is the same object as that returned by the **form** property of the control.

At run time, if the control has an associated window (the normal runtime control situation), the parent can be changed only to another form or control with an associated window. Changing the parent at run time by using this property is not recommended. The following example finds all direct children of **Frame1** and makes them invisible.

```
vars
    ctl  : Control;
    indx : Integer;
begin
    foreach indx in 1 to controlCount do
        ctl := controls(indx);
        if ctl.parent = frame1 then
            ctl.visible := false;
        endif;
    endforeach;
end;
```

parentAspect

Type: Integer

Availability: Read or write at any time

The **parentAspect** property of the **Control** class contains the aspect of the control to its parent. The default value for this property is **ParentAspect_None** (0). For details about obtaining the bottom and right offsets of the control from its parent, see the **Control** class **parentBottomOffset** and **parentRightOffset** properties.

The **parentAspect** property provides the following features for any control other than a **Sheet**, **StatusLine**, **JadeDockBar**, or **JadeDockContainer** control, to enable you to define resize and reposition actions that take effect without having to write any logic.

- Determines whether the control stretches horizontally, vertically, or both horizontally and vertically with the parent as the parent size changes.

  For horizontal stretching, the **left** position remains fixed and the control **width** is changed so that the distance from the right edge of the control to the right of the client area of the parent remains constant. For vertical stretching, the **top** position remains fixed and the control **height** is changed so that the distance from the bottom edge of the control to the bottom of the client area of the parent remains constant.

- Determines whether the control is anchored to the right, bottom, or both to the right and bottom of the parent as the size of the parent changes.

  For horizontal anchoring, the **left** position of the control is changed so that the distance from the right edge of the control to the right of the client area of the parent remains constant. For vertical anchoring, the **top** position of the control is changed so that the distance from the bottom edge of the control to the bottom of the client area of the parent remains constant.

- Determines whether the control is centered horizontally, vertically, or both horizontally and vertically within the client area of the parent as the size of the parent changes.

When the value of the **StatusLine** control **autoSize** property is **true**:

- If a child control **left** property value position is less than zero, the control is moved to be zero (**0**) when the **parentAspect**, **relativeLeft**, and **relativeWidth** property values of the child do not affect the horizontal position (not stretch horizontal, anchor right, and centered horizontal, and the **relativeLeft** and **relativeWidth** property values are **false**).

- If a child control is not fully visible horizontally, the child is right-aligned in the status line control if it can be fully displayed or positioned at zero (**0**) if it cannot when the **parentAspect**, **relativeLeft**, and **relativeWidth** property values of the child do not affect the horizontal position (not stretch horizontal, anchor right, and centered horizontal, and the **relativeLeft** and **relativeWidth** property values are **false**).
- If a child control top position is less than zero (**0**), the control is moved to be zero when the **parentAspect** property value of the child does not affect the vertical position (not stretch vertical, anchor bottom, or centered vertical).
- If a child control is not fully visible vertically, the child is bottom-aligned in the **StatusLine** control when the **parentAspect** property value of the child does not affect the vertical position (not stretch vertical, anchor bottom and centered vertically).
- The values of the **relativeTop** and **relativeHeight** properties of child controls are always set to **false**, as their functionality is not compatible with auto-sizing the height of the **StatusLine** control (as has always been the case).
- All **parentAspect** flag values and **relativeLeft** and **relativeWidth** values are applied.

The height of the **StatusLine** control is then determined by analyzing the child control as follows, to determine the maximum height required. For a child control that does not have a fixed height and has the **parentAspect** property with the:

- **ParentAspect_StretchBottom** flag set, the height required is the **top** position, **height**, and **parentBottomOffset** property values of the child.
- **ParentAspect_AnchorBottom** flag set, the height required is the **height** and **parentBottomOffset** property values of the child; otherwise, the **height** of the child control.

The **Control** class constants are listed in the following table.

Constant	Integer Value
ParentAspect_None	0
ParentAspect_StretchRight	#1
ParentAspect_StretchBottom	#2
ParentAspect_StretchBoth	#3
ParentAspect_AnchorRight	#4
ParentAspect_AnchorBottom	#8
ParentAspect_AnchorBoth	#c
ParentAspect_CenterHorz	#10
ParentAspect_CenterVert	#20
ParentAspect_CenterBoth	#30

The property values are a bit mask of valid combinations. For example, you can anchor a control to the bottom of its parent and also have it stretch horizontally as the parent width varies, by setting the property value to **ParentAspect_StretchRight** + **ParentAspect_AnchorBottom**.

Similarly, you can center a control vertically and have it stretch horizontally as the parent width varies, by setting the property value to **ParentAspect_CenterVert** + **ParentAspect_StretchRight**. However, not all combinations are compatible, and an invalid combination results in an exception being raised at run time.

The following values for **parentAspect** are mutually incompatible and you can include only one.

- **ParentAspect_StretchRight**
- **ParentAspect_AnchorRight**
- **ParentAspect_CenterHorz**

The following values for **parentAspect** are mutually incompatible and you can include only one.

- **ParentAspect_StretchBottom**
- **ParentAspect_AnchorBottom**
- **ParentAspect_CenterVert**

When using the **parentAspect** property, note the following points.

- The property is ignored:
  - If the alignment of the control is already specified by using the **alignContainer** or **alignChildren** property of a **Frame**, **JadeDockBar**, or **JadeDockContainer** control; for example, a frame that has the **alignContainer** property set to **AlignContainer_Width** (3).
  - For direct children of a **JadeDockContainer** control, which always automatically positions its children. Children that have no specific alignment are positioned in rows or columns so that they do not overlap.

    The **JadeDockContainer** also adjusts the position of the children to minimize the height or width of the container, when appropriate.
  - For a **JadeDockBar** control except when the container has the **alignChildren** property set to **AlignChildren_None** (0). This setting means that it is your responsibility to position the children.

    All other settings cause the children to be automatically positioned.
- Some controls have fixed heights and widths. Any stretching is ignored for those cases, as follows.
  - Height
    - A **Button** control that is automatically sized to its **picture** property
    - Any fixed height **CheckBox** control
    - Any fixed height **ComboBox** control
    - A **Label** control that is automatically sized
    - Any fixed height **OptionButton** control
    - A **Picture** control that automatically sizes to its **picture** property; that is, **Stretch_ControlTo** (2) and **Stretch_Proportional** (3)
    - An automatically positioned **Sheet** or **StatusLine** control
  - Width
    - A **Button** control that is automatically sized to its **picture** property
    - A **Label** control that is automatically sized
    - A **Picture** control that automatically sizes to its **picture** property; that is, **Stretch_ControlTo** (2)

and **Stretch_Proportional** (3)

  – An automatically positioned **Sheet** or **StatusLine** control

- The **parentAspect** property values can conflict with the relative property requirements. Where they do, the relative property setting is ignored, as follows.

  - If **ParentAspect_StretchRight** applies, the **relativeLeft** and **relativeWidth** properties are ignored

  - If **ParentAspect_StretchBottom** applies, the **relativeTop** and **relativeHeight** properties are ignored

  - If **ParentAspect_AnchorRight** applies, the **relativeLeft** property is ignored

  - If **ParentAspect_AnchorBottom** applies, the **relativeTop** property is ignored

- For a **ListBox** or **TextBox** control with the **integralHeight** property set to **true**, the actual distance between the bottom edge of the control and the bottom of the client edge of the parent varies, as the height of the control is rounded to the nearest integral line height.

- If logic changes the position or size of the control, the distances to the edges of the parent of the control are re-evaluated and used from then on.

Note You can also use the **Form** class **minimumHeight** and **minimumWidth** properties to prevent the user resizing the form and causing controls to overlap or the form to have no effective height or width.

parentBottomOffset

Type: Integer

Availability: Read or write at any time

The **parentBottomOffset** property of the **Control** class contains the pixel offset of the bottom edge of a control from the bottom of the client area of its parent. By default, the value of this property is automatically calculated.

This value is used primarily when the **parentAspect** property requires the bottom offset. The value is stored persistently at design time so that the correct bottom position of the **parentAspect** property can be restored when the form is loaded and is therefore unaffected by whether skins are now in use, the application three-dimensional border default values have been changed, subclassed forms of different sizes in cases where the control's parent is also affected by the resize, and so on.

You can set the **parentBottomOffset** property value for any control and it is effectively the same as setting the top position of a control to the pixel position (that is, the value of the parent's **clientHeight** property less the value of the **parentBottomOffset** property and the control height in pixels).

Changing the top position of the control updates the value of the **parentBottomOffset** property and setting the value of the **parentBottomOffset** property updates the top position value.

Notes When a control is added by using the **addControl** method of the **Form** class, the values of the **parentBottomOffset** and **parentRightOffset** property values are calculated initially from the resulting size and position of the control within its parent. Setting these values before calling the **addControl** method has no effect.

The **parentBottomOffset** and **parentRightOffset** property values only take effect when a control is created if the control was saved using the painter or if the **loadControl** method of the **Form** class is used (or if dynamically changed via logic).

parentRightOffset

Type: Integer

Availability: Read or write at any time

The **parentRightOffset** property of the **Control** class contains the pixel offset of the right edge of a control from the right side the client area of its parent. By default, the value of this property is automatically calculated.

This value is used primarily when the **parentAspect** property requires the right offset. The value is stored persistently at design time so that the correct right position of the **parentAspect** property can be restored when the form is loaded and is therefore unaffected by whether skins are now in use, the application three-dimensional border default values have been changed, subclassed forms of different sizes in cases where the parent of the control is also affected by the resize, and so on.

You can set the **parentRightOffset** property value for any control and it is effectively the same as setting the left position of a control to the pixel position (that is, the value of the parent's **clientHeight** property less the value of the **parentRightOffset** property and the control height in pixels).

Changing the left position of the control updates the value of the **parentRightOffset** property and setting the value of the **parentRightOffset** property updates the left position value.

Notes When a control is added by using the **addControl** method of the **Form** class, the values of the **parentBottomOffset** and **parentRightOffset** property values are calculated initially from the resulting size and position of the control within its parent. Setting these values before calling the **addControl** method has no effect.

The **parentBottomOffset** and **parentRightOffset** property values only take effect when a control is created if the control was saved using the painter or if the **loadControl** method of the **Form** class is used (or if dynamically changed via logic).

partialTextIndication

Type: Boolean

Availability: Read or write at run time only

The **partialTextIndication** property of the **Table** class specifies whether an indication is displayed when there is insufficient room to show all text of the cell indicated by the **accessMode**, **accessedSheet**, **accessedRow**, **accessedColumn**, or **accessedCell** property. The default value for any new **Table** control that is added is **true**.

If this property is set to **true**, the end of the visible text is replaced by the points of ellipsis symbol (…), to indicate to the user that not all of the text can be made visible. The number of periods (or dots) that is displayed depends on the number of characters that can be displayed within the cell, with at least the first character of the original text being displayed. For example, **The text is too long to fit in the cell** text may be displayed as follows.

```
The text is too l...
```

If this property is set to **false**, the text in a cell is truncated if it does not fit within the cell area.

For details about word wrapping when displaying text in a table cell, see the **wordWrap** property.

partsDone

Type: Integer

The **partsDone** property of the **ProgressBar** control contains information for the **ProgressBar** control about the percentage of the task that is completed.

When you set the **partsDone** property, the **partsDone** mapping method colors the progress bar with the parts that are done when a change in parameter value affects the appearance of the progress bar. For example, if you set the **partsDone** property to **25** when the **partsInJob** property is set to **100**, the progress bar is a quarter colored, with 75 percent of the job remaining to be done.

When the application is running in thin client mode, use the **thinClientUpdateInterval** property to control how often the progress var will be redrawn as the percentage changes.

The following example shows the use of the **partsDone** property.

```
showProgress(runningTotal: Integer;
             totalNumber:  Integer;
             message:      String) updating;
begin
    progressBar.partsInJob := totalNumber;
    progressBar.partsDone  := runningTotal;
end;
```

partsInJob

Type: Integer

The **partsInJob** method of the **ProgressBar** control contains information for the **ProgressBar** control about the number of parts in the job.

When you set the **partsInJob** property followed by the **partsDone** property, the **partsDone** mapping method colors the progress bar with the parts that are done and displays the remainder that is to be done. For example, if you set the **partsDone** property to 25 when the **partsInJob** property is set to 100, the progress bar is a quarter colored, with 75 percent of the job remaining to be done.

When the application is running in thin client mode, use the **thinClientUpdateInterval** property to control how often the progress var will be redrawn as the percentage changes.

The following example shows the use of the **partsInJob** property.

```
showProgress(runningTotal: Integer;
             totalNumber:  Integer;
             message:      String) updating;
begin
    progressBar.partsInJob := totalNumber;
    progressBar.partsDone  := runningTotal;
end;
```

passwordField

Type: Boolean

Availability: Read or write at any time

The **passwordField** property specifies whether the characters typed by a user or placeholder characters are displayed in a **TextBox** control.

If the value of this property is **true**, the specified text is displayed as asterisk (*) characters. Use this property to create a single-line password text box. The **passwordField** property does not affect the entered text, but only the display of that text.

You can paste into a text box when the value of the **passwordField** property is set to **true** but you cannot copy (using the Ctrl+C shortcut keys) or cut (using the Ctrl+X shortcut keys) the password text to the clipboard; such an attempt is rejected with a message beep.

Setting the **passwordField** property value to **true** causes the text box to become a single-line entry text box and disables the **alignment**, **scrollBars**, and **scrollVertical** properties.

picture

Type: Binary

Availability: Read or write at any time, but read or write at run time only for tables

The **picture** property contains a graphic to be displayed in a control or a menu.

The settings of the **picture** property are listed in the following table.

Value	Description
(none)	No picture (the default).
(Bitmap, icon, cursor, JPEG, PNG, GIF, TIFF CCITT, or metafile)	Specifies a graphic. You can load the graphic from the Properties window in the JADE development environment. At run time, you can set this property, by using the **loadPicture** method on a bitmap, icon, metafile, GIF, JPEG, or PNG.

For a **Button** control defined for a Web page, the **picture** property is displayed as an image that can be clicked. For button controls, the picture is placed on the button to the left of the button caption. However, this property is ignored if the **buttonPicture** property of a button is set to a non-zero value.

When an icon image is assigned to button and the **Button** class **autoSize** property is not **AutoSize_Picture**, the size of the icon selected from the assigned icon image is now based on the client height of the button.

If the **autoSize** property is set, the picture is scaled to fit the button. The button picture painting colors the background of the picture to the background color of the button by using the **drawFloodFill** method.

When the button is disabled and an image is defined by the **pictureDisabled** property, this image is displayed instead.

For a **Picture** control on a Web page, the binary value is automatically converted to the Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), or Portable Network Graphics (PNG) image format, depending on the value specified in the **ImageType** parameter in the [WebOptions] section of the JADE initialization file. (The default value is **jpg**.)

When setting the **picture** property from the JADE Painter in the JADE development environment, the graphic is saved and loaded with the form. When you load a graphic at run time, the graphic is not saved with the application.

The graphic can be set by setting the **picture** property of the control to the **picture** property of another control or by using the **loadPicture** method of the **Application** class. For menus, the picture can be a bitmap, icon, cursor, portable network graphics image, or metafile. The picture is drawn at actual size, except for a metafile or on the menu bar of the form, when it is scaled to fit the menu line size.

Menus are drawn in four columns, as follows.

```
checkMark : picture : text : accelerator text
```

The width of each column is defined to be the maximum of all the displayed items in that popup menu.

The following examples show the use of the **picture** property.

```
buttonSelect_click(btn: Button input) updating;
vars
    file : CMDFileOpen;
begin
    pictureEnlarged.picture := null;
    create file;
    file.filter := "Pictures(*.bmp;*.ico)|*.bmp;*.ico";
    if file.open = 0 then
        pictureNormal.picture := app.loadPicture(file.fileName);
    endif;
end;

if logo.picture.length <> 0 then
    if product.logo = null  then
        create l;
    else
        l := product.logo;
    endif;
    l.loadSelf(logo.picture);
endif;
```

For the **Picture** control, the picture can be displayed as defined, it can be stretched to fit the control, or the control can be stretched to fit the picture size, according to the value of the **stretch** property.

For the **Table** control:

- The current **sheet**, **row**, and **column** properties define the picture access.
- If the cell is too small to fit both the text and the picture, the text takes precedence over the picture unless the **stretch** property is set to **Stretch_None_Picture_First** (**2**) or **Stretch_Cell_Picture_First** (**3**), where the picture takes precedence.
- The size of the text is determined by taking the cell size and calculating the required text size using the word wrap option. The space that is left over is used to scale the image proportionally, so that the whole image is displayed.

  If the **stretch** property for the table control is set to **Stretch_None** (**0**), the picture is drawn to actual size. If it does not fit the available space, it is truncated on the right and bottom.

For GIF picture types:

- Only the first image in the file is displayed, by default.
- Only .**gif** files containing images are handled. (Text records within a .**gif** file are ignored.)
- The FreeImage library used by JADE does not support the ability to convert images to .**gif** files.
- Animated GIF handling is supported if the **picture** property of a **Picture** or **JadeMask** control is set to a GIF binary that contains more than one image. That control then runs the animation on a separate thread in a similar way to the **Picture** control **setPicture** method list and play mechanism.
- Use of an animated GIF file is mutually exclusive with the use of the **Picture** control **pictureCount** and **setPicture** methods. Setting the **picture** property to an animated GIF removes any pictures created by using the **setPicture** method.

  Similarly, changing the value of the **pictureCount** property or using the **setPicture** method closes any animated GIF operation (but leaves the value of the **picture** property unchanged).
- An animated GIF file commences animation when the control first paints.
- The animation loops indefinitely.
- Although GIF files have the ability to require user input before continuing from an animation point, this requirement for user input is ignored by JADE.
- Text display records within a GIF file are ignored.

When using an animated GIF picture, note the following.

- Animated .**gif** files are not drawn stretched. The **stretch** property is used to resize the control to the GIF image if required, but the animation is always drawn at its normal size.
- You can use the **play** and **stop** methods with an animated .**gif** file. The .**gif** file then starts playing automatically when the control is first painted, unless the **stop** method has been called.

  After calling the **stop** method for an animated .**gif** file, you must then call the **play** method to start or continue the animation.

pictureClosed

Type: Binary

Availability: Read or write at any time

The **pictureClosed** property contains the qualifying picture image displayed for an entry in a **ListBox** control or a **ComboBox** control. The picture can be set only to a bitmap or an icon image. It cannot be cleared.

Use the **hasPictures** property to control whether the image is displayed. If the picture is larger than the list line size, it is scaled to fit.

pictureCount

Type: Integer

Availability: Run time only

The **pictureCount** property of the **Picture** class contains the maximum index of the picture array associated with a picture control. This array can be used to hold an array of pictures that are selected for display based on JADE logic. Use the **setPicture** method to load pictures into the array.

Setting the **pictureCount** property to the maximum index value before loading a picture is more efficient, as the required array can be preallocated. If the **setPicture** method is called with an index greater than the current value of the **pictureCount** property, the **pictureCount** property is enlarged accordingly.

Use the **pictureIndex** property to control the picture that is currently displayed.

Changing the value of the **pictureCount** property closes any animated GIF operation of a **Picture** or **JadeMask** control (but leaves the value of the **picture** property unchanged).

pictureDisabled

Type: Binary

Availability: Read or write at any time

The **picture** property of the **Picture** control defines the picture that is normally displayed. However, you can give the picture box the appearance of being a button, by assigning pictures to the **pictureDisabled** and **pictureDown** properties.

For **Picture** controls, the **pictureDisabled** property defines the picture displayed when the picture box is disabled. It defaults to the **picture** property. Picture control pictures are scaled according to the **stretch** property of the picture box.

For a **Button** control, the **buttonPicture** or **picture** property defines the picture that is normally displayed on the button, if any. For button controls, the **pictureDisabled** property defines the picture that is displayed if the button is disabled, regardless of which of the **buttonPicture** or **picture** properties is set.

When an icon image is assigned to button and the **Button** class **autoSize** property is not **AutoSize_Picture**, the size of the icon selected from the assigned icon image is now based on the client height of the button.

By default, for **Button** controls, a disabled button grays the text and displays the picture defined by the **buttonPicture** or **picture** property.

pictureDown

Type: Binary

Availability: Read or write at any time

The **picture** property of the **Picture** control contains the picture that is normally displayed. However, you can give the picture box the appearance of being a button, by assigning pictures to the **pictureDisabled** and **pictureDown** properties.

For **Picture** controls, the **pictureDown** property defines the picture displayed when the mouse is pressed on the control. This property defaults to the **picture** property. **Picture** control pictures are scaled according to the **stretch** property of the picture box.

For a **Button** control, the **buttonPicture** or **picture** property defines the picture that is normally displayed on the button, if any. The **pictureDown** property defines the picture that is displayed when the mouse is pressed on the button, regardless of which of the **buttonPicture** or **picture** properties is set.

When an icon image is assigned to button and the **Button** class **autoSize** property is not **AutoSize_Picture**, the size of the icon selected from the assigned icon image is now based on the client height of the button.

By default, for **Button** controls, a disabled button grays the text and displays the picture defined by the **buttonPicture** or **picture** property.

pictureFocus

Type: Binary

Availability: Read or write at any time

The **pictureFocus** property of the **JadeMask** class contains the picture that defines the mask for the control when the control has focus and it is in the up position. If the **pictureFocusDown** property is not set, the JADE mask defaults to the value of the **pictureFocus** property if that is set.

The picture is displayed when the control has focus, it is not disabled, not currently clicked (that is, the left mouse is not down), and the mouse is not over the control.

If these pictures are not defined, the **pictureRollOver** state is displayed when the control has focus or the normal control picture state if that picture is not provided.

pictureFocusDown

Type: Binary

Availability: Read or write at any time

The **pictureFocusDown** property of the **JadeMask** class contains the picture that defines the mask for the control and specifies the logical area of the control. If this property is not set, the JADE mask defaults to the value of the **pictureFocus** property if that is set.

The picture is displayed when the control has focus, it is not disabled, not currently clicked (that is, the left mouse is not down), and the mouse is not over the control.

If these pictures are not defined, the **pictureRollOver** state is displayed when the control has focus or the normal control picture state if that picture is not provided.

pictureIndex

Type: Integer

Availability: Run time only

The **pictureIndex** property of the **Picture** class contains the picture that is displayed in a picture box. By default, this value is zero (**0**); that is, the value of the **picture** property is displayed. If the value of the **pictureIndex** property is greater than zero (**0**), the picture that is added to the picture array by the **setPicture** method is displayed.

The value of the **pictureIndex** property must be zero (**0**) or correspond to a valid array entry.

Note If the left mouse button is down over the picture box and the **pictureDown** property is set, that picture is displayed regardless of the setting of the **pictureIndex** property.

Similarly, if the picture box is disabled and the **pictureDisabled** property is set, this picture is used regardless of the value of the **pictureIndex** or **pictureDown** property.

pictureLeaf

Type: Binary

Availability: Read or write at any time

The **pictureLeaf** property contains the qualifying picture image displayed for an entry in a **ListBox** control or a **ComboBox** control. The picture can be set only to a bitmap or an icon image. It cannot be cleared.

Use the **hasPictures** property to control whether the image is displayed. If the picture is larger than the list line size, it is scaled to fit.

pictureMask

Type: Binary

Availability: Read or write at any time

The **pictureMask** property of the **JadeMask** class contains the picture that defines the mask for the control and specifies the logical area of the control.

Any pixel with the color of the **activeColor** property in the mask picture defines a pixel within the logical control. A pixel of any other color in the mask is considered not to be on the control and the control will not respond while the mouse is over that pixel.

The picture for the mask can be a bitmap, PNG, GIF, JPEG, or TIFF image only.

Note The masking ability is disabled if the value of the **rotation** property is non-zero or the value of the **style** property is **Style_Mask_Color** (**3**).

The mask picture is never shown. (See also the **JadeMask** class **createRegionFromMask** property.)

If the value of the **pictureMask** property is null (**""**), the roll over and roll under effects (controlled by the **pictureRollOver** and **pictureRollUnder** properties) occur when the mouse is over any part of the control.

pictureMinus

Type: Binary

Availability: Read or write at any time

The **pictureMinus** property contains the qualifying picture image displayed for an entry in a **ListBox** control or a **ComboBox** control. The picture can be set only to a bitmap or an icon image. It cannot be cleared.

Use the **hasPictures** property to control whether the image is displayed. If the picture is larger than the list line size, it is scaled to fit.

pictureOpen

Type: Binary

Availability: Read or write at any time

The **pictureOpen** property contains the qualifying picture image displayed for an entry in a **ListBox** control or a **ComboBox** control. The picture can be set only to a bitmap or an icon image. It cannot be cleared.

Use the **hasPictures** property to control whether the image is displayed.

If the picture is larger than the list line size, it is scaled to fit.

picturePlus

Type: Binary

Availability: Read or write at any time

The **picturePlus** property contains the qualifying picture image displayed for an entry in a **ListBox** control or a **ComboBox** control. The picture can be set only to a bitmap or an icon image. It cannot be cleared.

Use the **hasPictures** property to control whether the image is displayed. If the picture is larger than the list line size, it is scaled to fit.

pictureRollOver

Type: Binary

Availability: Read or write at any time

The **pictureRollOver** property of the **JadeMask** class contains the picture that is displayed when the mouse is over the control or the control has focus and the *button* is in the up position (that is, **value** = **false**). See also the **picture**, **pictureDown**, **pictureDisabled**, **pictureRollUnder**, and **pictureMask** properties.

Note If the value of the **pictureIndex** property is not zero (**0**), the **pictureIndex** picture is always displayed unless the control is disabled or the mouse is currently pressed on the control (the picture determined by the value of the **pictureDown** property is displayed).

If the value of the **pictureRollOver** property is **null**, the picture determined by the **picture** property is displayed when the **value** property is **false** and the mouse moves over the control or the control has focus.

pictureRollUnder

Type: Binary

Availability: Read or write at any time

The **pictureRollUnder** property of the **JadeMask** class contains the picture that is displayed when the mouse is over the control or the control has focus and the *button* is in the down position (that is, **value** = **true**).

See also the **picture**, **pictureDown**, **pictureDisabled**, **pictureRollOver**, and **pictureMask** properties.

Note If the value of the **pictureIndex** property is not zero (**0**), the picture determined by the **pictureIndex** property is always displayed unless the control is disabled or the mouse is currently pressed on the control (the picture determined by the value of the **pictureDown** property is displayed).

If the value of the **pictureRollUnder** property is **null**, the **pictureDown** property picture is displayed when the **value** property is **true** and the mouse moves over the control or the control has focus.

position

Type: Integer

Availability: Read or write at any time

The **position** property of the **MultiMedia** class contains the current position of the medium with the content of the device.

The units for the position value depend on the time format of the device. For details, see the **timeFormat** property.

The default value is the current value of the content, which is initially zero (**0**).

The following example shows the use of the **position** property.

```
backtrack_click(btn: Button input) updating;
vars
    track : Integer;
begin
    track           := currentTrack;
    if track > 1 then
        track       := track - 1;
        cd.position := trackPosition(track);
    endif;
end;
```

promptCharacter

Type: Character

Availability: Read or write at any time

The **promptCharacter** property of the **JadeEditMask** class contains the character that is used to fill the character positions of the text in text box fields of the control that can have data entered.

The default value is the _underscore character. The prompt offers the user a visual indication of positions at which text can be entered. When a character is typed in that character position, the prompt character is replaced, regardless of the setting of the **insertMode** property.

Note The control can handle entry of the underscore character that is used as the prompt character and it can distinguish between the two situations when the text is retrieved.

Setting the **promptCharacter** property to **null** causes a blank prompt character to be used, which is necessary for handling any literal in the mask sequence.

readOnly

Type: Boolean

Availability: Read or write at any time

The **readOnly** property specifies whether a control is read-only for user input.

For a **CheckBox** control, if the value of the **readOnly** property is **true**, the control does not respond to mouse or keyboard actions. Its state can be changed only from logic.

For a **TextBox** and **JadeEditMask** controls, the control can be tabbed to, responds to user mouse actions, and text can be selected, but keyboard input is ignored. For a text box or edit mask control on a Web page, the control is converted to static text if the value of this property is **true**.

For a **Table** control, if the value of the **readOnly** property is **true**:

- No key events are processed
- The user can still change cells that have the **inputType** property set
- No select or select clear occurs when a mouse is clicked in a cell
- No focus rectangle is displayed
- Clicking a cell still causes a **rowColumnChg** event

Tip Set the **userInputEnabled** property to **false** if you want the value of the **inputType** or the **cellControl** property ignored for all cells.

The settings of the **readOnly** property are listed in the following table.

Value	Description
false	Control responds to mouse and keyboard actions (the default)
true	Control does not respond to mouse and keyboard actions

Setting the **readOnly** property to **true** enables a check box to be used to indicate the state of an option without the user being able to directly change that option.

The **readOnly** property of the **JadeRichText** control specifies whether the contents of the control can be updated (that is, whether user input is accepted). A double-click action is ignored in a read-only **JadeRichText** control only if an object is selected. Double-clicking a hyperlink in a read-only **JadeRichText** control processes the URL request (that is, the control can be tabbed to, respond to user mouse actions, and text can be selected, but keyboard input is ignored). The default value of **false** indicates that the control responds to both mouse and keyboard actions.

The **readOnly** property of the **JadeTextEdit** control specifies whether the text editor is read-only for user input. The **EVENTTYPE_ALTERREADONLY** notification occurs on each attempt by the user or the application to change the text editor contents.

Assigning to the **JadeTextEdit** class **text** property is permitted when the value of the **readOnly** property is **true** but all other user and programmatic changes are ignored.

relativeHeight

Type: Boolean

Availability: Read or write at any time

The **relativeHeight** property of the **Control** class specifies whether the height of a control is relative to the height of its parent. If this property is set to **true**, the control height is adjusted proportionally when the parent is resized. For example, this means that list boxes can be sized according to the form height.

The settings of the **relativeHeight** property are listed in the following table.

Value	Description
false	If the parent is resized, no action occurs (the default)
true	Resize the control height in proportion to the size change undergone by the parent

Setting the **relativeHeight** property to **true** causes the adjustment of the height for the control when the parent size changes. For example, if all of the relative properties are set to **true**, two adjacent list boxes resize, move, and are still adjacent after the resize of a form. Similarly, if you set the **relativeTop** and **relativeLeft** properties to **true**, a button remains the same size but it is positioned proportionally to the form size.

Note Some controls automatically resize themselves, and setting this property may have no effect. In addition, the **relativeHeight** property is ignored when a control is aligned. (See also the **parentAspect** property.)

The size ratios between the control and its parent are calculated when:

- A form is loaded, if any of the property values are set to **true**.

  The ratios are calculated using the development property values of **left**, **top**, **height**, and **width** of the control and its parent established by the Painter in the JADE development environment.

- The form is running:

  - If any of the **relativeLeft**, **relativeTop**, **relativeHeight**, or **relativeWidth** properties of the control are changed to **true**.

  - If any of the **left**, **top**, **height**, or **width** properties of the control are set and any of the relative properties are set to **true**.

  - If the **move** method of the **Window** class is called for the control and any of the relative properties are set to **true**.

  In these situations, the current size of the control and parent recalculate the size ratios.

If logic changes the position or size of the control, the distances to the edges of the parent of the control are re-evaluated and used from then on (that is, the relative ratios are re-evaluated).

When using the **relativeHeight** property, some controls have fixed heights. Any stretching is ignored for those cases, as follows.

- A **Button** control that is automatically sized to its **picture** property

- Any fixed height **CheckBox** control

- Any fixed height **ComboBox** control

- A **Label** control that is automatically sized

- Any fixed height **OptionButton** control
- A **Picture** control that automatically sizes to its **picture** property; that is, **Stretch_ControlTo** (**2**) and **Stretch_ Proportional** (**3**)
- An automatically positioned **Sheet** control

The following example shows the use of the **relativeHeight** property.

```
setFormProperties(pScaleForm, pRelativeSize: Boolean) updating;
vars
    count : Integer;
begin
    // Set the form's scale property.
    self.scaleForm := pScaleForm;
    // Now set the relative size and position properties of the
    // form's controls.
    foreach count in 1 to self.controlCount do
        self.controls(count).relativeHeight := pRelativeSize;
        self.controls(count).relativeLeft   := pRelativeSize;
        self.controls(count).relativeTop    := pRelativeSize;
        self.controls(count).relativeWidth  := pRelativeSize;
    endforeach;
end;
```

relativeLeft

Type: Boolean

Availability: Read or write at any time

The **relativeLeft** property of the **Control** class specifies whether the left position of a control is relative to the width of its parent.

If this property is set to **true**, the control position is adjusted proportionally when the parent is resized. For example, the left position of a list box can be set relative to the width of its form parent.

The settings of the **relativeLeft** property are listed in the following table.

Value	Description
false	If the parent is resized, no action occurs (the default)
true	Reposition the control left in proportion to the size change undergone by the parent

Setting the **relativeLeft** property to **true** causes the adjustment of the left position for the control when the parent size changes. For example, if all of the relative properties are set to **true**, two adjacent list boxes resize, move, and are still adjacent after the resize of a form. Similarly, if you set the **relativeTop** and **relativeLeft** properties to **true**, a button remains the same size but it is positioned proportionally to the form size.

Notes Some controls automatically resize themselves, and setting this property may have no effect. In addition, the **relativeLeft** property is ignored when a control is aligned. (See also the **parentAspect** property.)

If logic changes the position or size of the control, the distances to the edges of the parent of the control are re-evaluated and used from then on (that is, the relative ratios are re-evaluated).

The size ratios between the control and its parent are calculated when:

- A form is loaded, if any of the property values are set to **true**. The ratios are calculated using the development property values of **left**, **top**, **height**, and **width** of the control and its parent established by the Painter.
- The form is running if:
    - Any of the **relativeLeft**, **relativeTop**, **relativeHeight**, or **relativeWidth** properties of the control are changed to **true**.
    - Any of the **left**, **top**, **height**, or **width** properties of the control are set and any of the relative properties are set to **true**.
    - The **move** method of the **Window** class is called for the control and any of the relative properties are set to **true**.

  In these situations, the current size of the control and parent are used to recalculate the size ratios.

relativeTop

Type: Boolean

Availability: Read or write at any time

The **relativeTop** property of the **Control** class specifies whether the top position of a control is relative to the size of its parent. If this property is set to **true**, the control position is adjusted proportionally when the parent is resized.

The settings of the **relativeTop** property are listed in the following table.

Value	Description
false	If the parent is resized, no action occurs (the default)
true	Reposition the control in proportion to the size change undergone by the parent

Setting the **relativeTop** property to **true** causes the adjustment of the top position when the parent size changes. For example, if all of the relative properties are set to **true**, two adjacent list boxes resize, move, and are still adjacent after the resize of a form.

Similarly, if you set the **relativeTop** and **relativeLeft** properties to **true**, a button remains the same size but it is positioned proportionally to the form size.

Notes Some controls automatically resize themselves, and setting this property may have no effect. In addition, the **relativeTop** property is ignored when a control is aligned. (See also the **parentAspect** property.)

If logic changes the position or size of the control, the distances to the edges of the parent of the control are re-evaluated and used from then on (that is, the relative ratios are re-evaluated).

The size ratios between the control and its parent are calculated when:

- A form is loaded, if any of the property values are set to **true**. The ratios are calculated using the development property values of **left**, **top**, **height**, and **width** of the control and its parent established by the Painter.

- The form is running if:
  - Any of the **relativeLeft**, **relativeTop**, **relativeHeight**, or **relativeWidth** properties of the control are changed to **true**.
  - Any of the **left**, **top**, **height**, or **width** properties of the control are set and any of the relative properties are set to **true**.
  - The **move** method of the **Window** class is called for the control and any of the relative properties are set to **true**.

  In these situations, the current size of the control and parent are used to recalculate the size ratios.

relativeWidth

Type: Boolean

Availability: Read or write at any time

The **relativeWidth** property of the **Control** class specifies whether the width of a control is relative to the size of its parent. If this property is set to **true**, the control width is adjusted proportionally when the parent is resized.

The settings of the **relativeWidth** property are listed in the following table.

Value	Description
false	If the parent is resized, no action occurs (the default)
true	Resize the control height in proportion to the size change undergone by the parent

Setting the **relativeWidth** property to **true** causes the adjustment of the width for the control when the parent size changes. For example, if all of the relative properties are set to **true**, two adjacent list boxes resize, move, and are still adjacent after the resize of a form. Similarly, if you set the **relativeTop** and **relativeLeft** properties to **true**, a button remains the same size but it is positioned proportionally to the form size.

Note Some controls automatically resize themselves, and setting this property may have no effect. In addition, the **relativeWidth** property is ignored when a control is aligned.

The size ratios between the control and its parent are calculated when:

- A form is loaded, if any of the property values are set to **true**. The ratios are calculated using the development property values of **left**, **top**, **height**, and **width** of the control and its parent established by the Painter.
- The form is running if:
  - Any of the **relativeLeft**, **relativeTop**, **relativeHeight**, or **relativeWidth** properties of the control are changed to **true**.
  - Any of the **left**, **top**, **height**, or **width** properties of the control are set and any of the relative properties are set to **true**.
  - The **move** method of the **Window** class is called for the control and any of the relative properties are set to **true**.

  In these situations, the current size of the control and parent are used to recalculate the size ratios.

If logic changes the position or size of the control, the distances to the edges of the parent of the control are re-evaluated and used from then on (that is, the relative ratios are re-evaluated).

When using the **relativeWidth** property, some controls have fixed widths. Any stretching is ignored for those cases, as follows.

- A **Button** control that is automatically sized to its **picture** property
- A **Label** control that is automatically sized
- A **Picture** control that automatically sizes to its **picture** property; that is, **Stretch_ControlTo** (2) and **Stretch_Proportional** (3)
- An automatically positioned **Sheet** control

repeat

Type: Boolean

Availability: Read or write at any time

The **repeat** property of the **MultiMedia** class specifies whether continuous playback mode is set. When this property is set, playing the device by using the **play** method results in a continuous playback loop. Only the **digitalvideo** device supports continuous playback.

rightIndent

Type: Integer

Availability: Read or write at any time

The **rightIndent** property of the **JadeRichText** control contains the distance (in pixels) between the right edge of the control and the right edge of the current selection or text added at the current insertion point.

If multiple paragraphs are selected and each has a different value, the property contains **ParagraphFormat_Undefined** (#80000000). The default value of zero (**0**) indicates that the control is not indented.

rotation

Type: Real

Availability: Read or write at any time

The **rotation** property of the **Picture** control contains the number of radians by which the current picture is rotated about the center point of the control. The default value is zero (**0**).

The picture is:

- Sized according to the size of the picture and the value of the **stretch** property setting (ignoring the value of the **rotation** property).
- Then drawn to that size and rotated about the central point of the control. Any parts of the picture outside the control are clipped.

As a result, the best way to handle the **rotation** property is to:

1. Construct the picture in another **Picture** control to the required size without rotation, with the appropriate **stretch** property value and control size.

2. Use the **Window** class **createPictureAsType** method to obtain the binary picture value from the **Picture** control that you created in the previous step of this instruction.

   If you call the **createPictureAsType** method with a bit value less than the bit value of the original image, color distortion can occur.

3. Resize the destination **Picture** control so that it fits the rotated picture. (Set the destination **stretch** property value to **Stretch_None** and the **picture** property to the binary value returned from **createPictureAsType** method.)

In addition, creating a picture with the **pictureType** method set to **PictureType_Jpeg** results in picture image quality reduction, because the JPEG format is lossy; that is, picture quality is reduced to lessen the size of the resulting image. To retain existing quality and produce a smaller binary image, use a picture type such as **PictureType_Png**, which has a lossless compression of images for greater clarity.

Note Any stretching of an image can cause image distortion.

row

Type: Integer

Availability: Read or write at run time only

The **row** property contains the current row on the current sheet of a **Table** control.

The **column**, **row**, and **sheet** properties define the current cell when accessing certain properties within the table control; for example, the **text**, **picture**, and **selected** properties of a cell.

If the current cell is visible on the table and the **Table** control has focus, the cell has focus rectangle painted on it.

Row access is **1**-relative. Changing the value of the **row** property does not cause the table to be repainted.

The following examples show the use of the **row** property.

```
// If new instance, then add new row to database
if isNewInstance and
    instancesTable.row > instancesTable.fixedRows then
        saveOnDatabase(currentRow);
        instancesTable.setFocus;
    return;
endif;

dragDrop(table: Table input; win: Window input; x: Real; y: Real) updating;
begin
    inheritMethod(table, win, x, y);
    if dragRow > 1 and dragColumn > 0 then
        row := dragRow;
        column := dragColumn;
        if text <> "" then
            calendar.changeType := calendar.ChangeType_Day;
            calendar.date.setDate(text.Integer, calendar.date.month,
                                  calendar.date.year);
        endif;
    endif;
end;
```

rowHeight

Type: Integer array

Availability: Read or write at run time only

The **rowHeight** property enables the size of a row of a **Table** control to be accessed.

The font selected for the table determines the default value of the **rowHeight** property. The user can also change the **rowHeight** property by dragging a fixed row boundary with the mouse.

The **rowHeight** property contains a reference to an array of integer values with the same number of items as the **rows** property.

Setting the height of a row causes a repaint. Setting the **rowHeight** property of a row to zero (**0**) causes it to use the default row height.

The code fragment in the following example shows the use of the **rowHeight** property.

```
while count > 0 do
    tbl.rowHeight[count] := (tbl.height / tbl.rows).rounded;
    count := count - 1;
endwhile;
```

Use the **rowVisible** property to hide a row rather than change its height.

rows

Type: Integer

Availability: Read or write at run time only

The **rows** property contains the number of rows on the current sheet of a **Table** control.

Increasing the number of rows adds empty rows to that existing sheet. Decreasing rows deletes excess rows, discarding any existing data in those rows. Changing the **rows** property value to zero (**0**) empties the sheet of any data.

Note Changing the value of the **rows** property can affect the current values of the **column**, **row**, **topRow**, and **leftColumn** properties.

Rows can also be added by using the **addItem** or **addItemAt** methods. A specific row can also be deleted by using the **removeItem** method.

You can assign a maximum of 32,000 rows to a **Table** control. However, depending on the number of columns that are also assigned to the table, the amount of memory required to handle a large number of rows limits the number of rows that can be handled in practice.

The code fragments in the following examples show the use of the **rows** property.

```
foreach count in 1 to tblPortfolio.rows do
    tblPortfolio.row := count;
    if tblPortfolio.itemObject.Portfolio = saveObject then
        tblPortfolio.selected := true;
        break;
    endif;
endforeach;
```

```
// Remove empty rows so that the sort columns method will work
tbl.column := 1;
row := 1;
while row <= tbl.rows do
    tbl.row := row;
    if tbl.text = "" then
        tbl.deleteRow(row);
        // Move back one row otherwise you are working on the next row!
        row := row - 1;
    endif;
    row := row + 1;
endwhile;
```

rowVisible

Type: Boolean array

Availability: Read or write at run time only

The **rowVisible** property enables a row of a **Table** control to be displayed or hidden.

Setting the **visible** status of a row causes a repaint.

scaleForm

Type: Boolean

Availability: Development only

The **scaleForm** property of the **Form** class specifies whether the form and all of its contents are scaled to match the current font-scaling attribute of the workstation environment (set from the Control Panel **Display** option font size). The default value is **true**.

When this property is set to the default value of **true**, the font-scaling attribute of the workstation is saved with the form. When the form is next loaded, the scaling of the workstation that is running the form is compared. If the font scaling is different, the form and all of its controls are scaled according to the different font scaling environment, thus preserving the layout of the form.

Note Changing this property has no effect after the affected form has been built.

scaleHeight

Type: Real

Availability: Read or write at run time only

The **scaleHeight** property of the **Window** class contains the number of units for the internal vertical measurement of an object when using graphics methods or when positioning child controls. It also determines the units returned during the **mouseDown**, **mouseHover**, **mouseLeave**, **mouseMove**, and **mouseUp** events for a form or control.

Use this property with the **scaleWidth** property to create a custom coordinate scale for drawing or positioning controls. For example, **scaleHeight := 100** defines the internal height of a form or control as 100 units, or one vertical unit as 1/100 of the height.

Use the **scaleMode** property to define a scale based on a standard unit of measurement; for example, twips, points, or pixels.

Setting the **scaleHeight** property interacts with the **scaleMode** property in the following ways.

- Setting any other scale property to any value automatically sets the **scaleMode** to **ScaleMode_User** (1).
- Setting the **scaleMode** property to a number other than **ScaleMode_User** (1) changes the **scaleHeight** and **scaleWidth** properties to the new unit of measurement and sets the **scaleLeft** and **scaleTop** properties to **0**.

Note The **scaleHeight** property is not the same as the **height** property.

scaleLeft

Type: Real

Availability: Read or write at run time only

The **scaleLeft** property of the **Window** class contains the horizontal coordinates for the left edge of an object when using graphics methods or when positioning child controls. It also determines the units returned during the **mouseDown**, **mouseHover**, **mouseLeave**, **mouseUp**, and **mouseMove** events for the form or control.

Use this property with the **scaleTop** property to create a custom coordinate scale for drawing or positioning controls.

Use the **scaleMode** property to define a scale based on a standard unit of measurement; for example, twips, points, or pixels.

Setting the **scaleLeft** property interacts with the **scaleMode** property in the following ways.

- Setting any other scale property to any value automatically sets the **scaleMode** property to **ScaleMode_User** (1).
- Setting the **scaleMode** property to a number other than **ScaleMode_User** (1) changes the **scaleHeight** and **scaleWidth** properties to the new unit of measurement and sets the **scaleLeft** and **scaleTop** properties to **0**.

Note The **scaleLeft** property is not the same as the **left** property.

scaleMode

Type: Integer

Availability: Read or write at run time only

The **scaleMode** property of the **Window** class contains the unit of measurement for coordinates of an object when using graphics methods or when positioning child controls.

It also determines the units returned during the **mouseDown**, **mouseHover**, **mouseLeave**, **mouseMove**, and **mouseUp** events for the form or control.

The settings of the **scaleMode** property are listed in the following table.

Window Class Constant	Value	Description
ScaleMode_Pixels	0	Pixel (smallest unit of monitor or printer resolution). This is the default value.

Window Class Constant	Value	Description
ScaleMode_User	1	Indicates that one or more of the **scaleHeight**, **scaleWidth**, **scaleLeft**, or **scaleTop** properties are set to custom values.
ScaleMode_Twip	2	Twip (1440 twips per logical inch; 567 twips per logical centimeter).
ScaleMode_Point	3	Point (72 points per logical inch).

Using the related **scaleHeight**, **scaleWidth**, **scaleLeft**, and **scaleTop** properties, you can create a custom coordinate system with both positive and negative coordinates. These four scale properties interact with the **scaleMode** property in the following ways.

- Setting the value of any other scale property to any value automatically sets the **scaleMode** property to **ScaleMode_User** (1).
- Setting the **scaleMode** property to a number other than **ScaleMode_User** (1) changes the **scaleHeight** and **scaleWidth** property values to the new unit of measurement and sets the **scaleLeft** and **scaleTop** properties to **0**.

Note If the **scaleMode** property is set to a non-zero value, setting control positions and sizes is expressed in scale mode units. These units are converted (and rounded) internally to pixels, to physically place the control on the screen. When the position or size is accessed, the pixel position or size is converted back to scale mode units, and a slightly different value may result due to the rounding that occurred.

Negative **scaleHeight** and **scaleWidth** property values do not apply to window positions and sizes. A window is always drawn across and down from its left and top positions, regardless of the sign of these scale properties.

scaleTop

Type: Real

Availability: Read or write at run time only

The **scaleTop** property of the **Window** class contains the vertical coordinates for the top edges of an object when using graphics methods or when positioning child controls. It also determines the units returned during the **mouseDown**, **mouseHover**, **mouseLeave**, **mouseUp**, and **mouseMove** events for the form or control. Use this property with the **scaleLeft** property to create a custom coordinate scale for drawing or positioning controls.

Use the **scaleMode** property to define a scale based on a standard unit of measurement; for example, twips, points, or pixels. Setting the **scaleTop** property interacts with the **scaleMode** property in the following ways.

- Setting any other scale property to any value automatically sets the **scaleMode** property to **ScaleMode_User** (1).
- Setting the **scaleMode** property to a number other than **ScaleMode_User** (1) changes the **scaleHeight** and **scaleWidth** properties to the new unit of measurement and sets the **scaleLeft** and **scaleTop** properties to **0**.

Note The **scaleTop** property is not the same as the **top** property.

scaleWidth

Type: Real

Availability: Read or write at run time only

The **scaleWidth** property of the **Window** class contains the number of units for the internal horizontal measurement of an object when using graphics methods or when positioning child controls. It also determines the units returned during the **mouseDown**, **mouseHover**, **mouseLeave**, **mouseMove**, and **mouseUp** events for a form or control.

Use this property with the **scaleHeight** property to create a custom coordinate scale for drawing or positioning controls. For example, **scaleHeight := 100** defines the internal height of a form or control as 100 units, or one vertical unit as 1/100 of the height.

Use the **scaleMode** property to define a scale based on a standard unit of measurement; for example, twips, points, or pixels. Setting the **scaleWidth** property interacts with the **scaleMode** property in the following ways.

- Setting any other scale property to any value automatically sets the **scaleMode** to **ScaleMode_User** (1).
- Setting the **scaleMode** property to a number other than **ScaleMode_User** (1) changes the **scaleHeight** and **scaleWidth** properties to the new unit of measurement and sets the **scaleLeft** and **scaleTop** properties to **0**.

Note The **scaleWidth** property is not the same as the **width** property.

scrollBars

Type: Integer

Availability: Read or write at any time

The **scrollBars** property of the **Form** class determines whether an object has horizontal or vertical scroll bars.

The **scrollBars** property of the **JadeRichText** control determines whether an object has horizontal or vertical scroll bars.

The settings of the **scrollBars** property are listed in the following table.

Window Class Constant	Value	Description
ScrollBars_None	0	None (the default)
ScrollBars_Horizontal	1	Horizontal
ScrollBars_Vertical	2	Vertical
ScrollBars_Both	3	Both

Scroll bars are displayed on an object (except for picture boxes, text boxes, and non-MDI forms that are always present) only if the contents of the object extend beyond the borders of the object. For example, in an MDI form, if part of a child form is hidden behind the border of the parent MDI form, a horizontal scroll bar is displayed.

Similarly, a vertical scroll bar is displayed on a **Table** when it cannot display all of its rows. If the **scrollBars** property is set to **ScrollBars_None** (0), the object does not have scroll bars, regardless of its contents.

For the **ListBox** and **Table** control classes only, the **Window** class provides the constants listed in the following table.

Windows Class Constant	Integer	Description
ScrollBars_PermanentHorizontal	4	Horizontal scroll bar is always shown
ScrollBars_PermanentVertical	5	Vertical scroll bar is always shown
ScrollBars_PermanentBoth	6	Both horizontal and vertical scroll bars are always shown
ScrollBars_HorzPermanentVert	7	Horizontal scroll bar is shown only if required and the vertical scroll bar is always shown
ScrollBars_VertPermanentHorz	8	Vertical scroll bar is shown only if required and the horizontal scroll bar is always shown

These constants enable you to define scroll bars for list boxes and tables that are permanently shown, regardless of whether they are needed or not. If a scroll bar on a list box or table control does not have a scroll range because there are insufficient entries, the scroll bar will be shown as disabled.

Any control that is aligned to its parent by using the **alignContainer** or **alignChildren** property will not scroll and that control will remain in place in its parent when the scroll bar of the parent is shifted. Dock control and status lines, for example, therefore remain visible and unchanged when the scroll bar of the parent is adjusted.

This property has no effect on scroll bars on MDI client windows. Scroll bars are drawn on an MDI client window if it is positioned within the MDI frame so that scroll bars are required, and these cannot be hidden. Scroll bars can be displayed or hidden only on the MDI frame form and the MDI child windows that the MDI client contains.

For a **TextBox** control, if a horizontal scroll bar is set, the **scrollHorizontal** property is automatically set to **true**. Similarly, a vertical scroll bar causes the **scrollVertical** property to be set to **true**.

For **JadeRichText** controls, scroll bars are not visible unless text extends beyond the control client window co-ordinates.

scrollHorizontal

Type: Boolean (**TextBox** class), Integer (**ListBox** class)

Availability: Read or write at any time

The **scrollHorizontal** property of a text box specifies whether the text scrolls horizontally. For a list box, it controls whether a list box control has a horizontal scroll bar added when a list box line item does not horizontally fit in the list box.

The settings for the **scrollHorizontal** property for **TextBox** controls are listed in the following table.

Value	Description
false	None (the default)
true	Text scrolls when the size of the text box is exceeded by the entered text

By default, the text in a text box does not scroll horizontally when the text is about to exceed the horizontal size of the control. If the height of the text box allows for more than one line of text, the text word-wraps onto the next line, until all visible lines are full.

Setting the **scrollHorizontal** property to **true** means that the text scrolls to the right as text is entered and the width of the text exceeds the width of the text area of the text box. If the value of the **scrollHorizontal** property is **true**, no word wrapping occurs, and the Enter key must be pressed to terminate the current line and to start a new line.

Note If the value of the **alignment** property of a text box is **Alignment_Right** or **Alignment_Center**, setting the value of the **scrollHorizontal** property to **true** has no effect.

Setting the **scrollBars** property so that a horizontal or vertical bar is displayed causes the **scrollHorizontal** property to be set to **true**. Setting the **scrollHorizontal** property to **true** does not cause scroll bars to be displayed, as this is achieved by setting the **scrollBars** property.

The settings for the **scrollHorizontal** property for **ListBox** controls are listed in the following table.

ListBox Class Constant	Value	Description
ScrollHorizontal_None	0	No horizontal scroll bar is displayed (the default)
ScrollHorizontal_Auto	1	A horizontal scroll bar is added automatically when a list line item cannot be fully displayed horizontally in the list box

scrollHorzPos

Type: Integer

Availability: Read or write at run time only

The **scrollHorzPos** property contains the position of the horizontal scroll bar. The units are pixels, except for a text box control where the horizontal positions are characters.

The **scrollHorzPos** property applies to **Form**, **Picture**, **TextBox**, **ListBox**, and **JadeRichText** controls. (Use the **topRow** and **leftColumn** properties to set the scroll positions for **Table** controls.)

If the scroll bar is not defined for the form or control, a value of zero (**0**) is returned, and setting the **scrollHorzPos** property has no effect.

Setting the value of the **scrollHorzPos** property when the horizontal scroll bar is defined scrolls the window to that horizontal position relative to the minimum scroll range value. Setting this property to a value outside the scroll range results in an exception being raised.

For the **JadeRichText** control, the **scrollHorzPos** property contains the horizontal position in the virtual text space corresponding to the point displayed on the left hand side of the control. The units are characters. If the scroll bar is not defined for the control, the value of this property defaults to zero (**0**) and setting the **scrollHorzPos** property has no effect.

The **scrolled** event is not generated as a result of setting a new scroll position.

This property can also be used to convert physical window positions into logical positions; as shown in the following example.

```
list1_mouseDown(listbox: ListBox input; button: Integer; shift: Integer;
                x: Real; y: Real) updating;
begin
    xpos := x + list1.scrollHorzPos;      // logical pos for mouse down
end;
```

scrollVertical

Type: Boolean

Availability: Read or write at any time

The **scrollVertical** property of **TextBox** controls specifies whether the text in a text box control scrolls vertically.

The settings for the **scrollVertical** property are listed in the following table.

Value	Description
false	None (the default)
true	Text scrolls when the size of the text box is exceeded by the entered text

By default, the text in a text box does not scroll vertically when the text is about to exceed the vertical size of the control. If the height of the text box allows for more than one line of text, the text word-wraps onto the next line, until all visible lines are full.

Setting the **scrollBars** property so that a vertical scroll bar is displayed causes the **scrollVertical** property to be set to **true**.

Setting the **scrollVertical** property to **true** does not cause scroll bars to be displayed, as this is achieved by setting the **scrollBars** property.

scrollVertPos

Type: Integer

Availability: Read or write at run time only

The **scrollVertPos** property contains the position of the vertical scroll bar.

The units are pixels, except for a text box control where the vertical positions are text lines.

The **scrollVertPos** property applies to **Form**, **Picture**, **TextBox**, and **JadeRichText** controls. (Use the **topIndex** property to set vertical scroll positions for **ListBox** controls and the **topRow** and **leftColumn** properties to set the scroll positions for **Table** controls.)

If the scroll bar is not defined for the form or control, a value of zero (**0**) is returned, and setting the **scrollVertPos** property has no effect. Setting the value of the **scrollVertPos** property when the vertical scroll bar is defined scrolls the window to that vertical position relative to the minimum scroll range value. Setting this property to a value outside the scroll range results in an exception being raised.

The **scrolled** event is not generated as a result of setting a new scroll position.

For the **JadeRichText** control, the **scrollVertPos** property contains the vertical position in the virtual text space corresponding to the point displayed at the top of the control. The units are characters.

If the scroll bar is not defined for the control, the value of this property defaults to zero (**0**) and setting the **scrollVertPos** property has no effect.

secureForm

Type: Boolean

Availability: Read or write at any time

The **secureForm** property of the **Form** class specifies whether Web forms are secure; that is, Uniform Resource Locators (URLs) and page submits use the **https** protocol.

By default, forms are not secure. (The **http** protocol is used.)

securityLevelEnabled

Type: Integer

Availability: Read or write at any time

The **securityLevelEnabled** property of the **Window** class determines whether the form or control is automatically disabled when its form is created and loaded or when this property is changed.

If the value of the **securityLevelEnabled** property of the form or control is greater than the value of the **userSecurityLevel** property of the **Application** class (**app**.**userSecurityLevel**), it is disabled regardless of the value of its **enabled** property when it is created.

You can use the **enabled** property in logic to override this setting.

securityLevelVisible

Type: Integer

Availability: Read or write at any time

The **securityLevelVisible** property of the **Window** class determines whether the form or control is automatically made invisible when its form is created and loaded or when this property is changed.

If the value of the **securityLevelVisible** property of the form or control is greater than the value of the **userSecurityLevel** property of the **Application** class (**app**.**userSecurityLevel**), it is made invisible regardless of the value of its **visible** property when it is created.

You can use the **visible** property in logic to override this setting.

selected

Type: Boolean

Availability: Read or write at run time

The **selected** property specifies whether the current cell on the current sheet of a **Table** control is selected. The **sheet**, **row**, and **column** properties define the current cell. Accessing cells for this status in the fixed column and row area has special meaning if:

- The current row and column are both in the fixed area, no action is taken and the **selected** property returns **false**.
- The current row but not the column is in the fixed area, setting the **selected** property sets all non-fixed cells in that column to the selected status. Getting the status in that situation returns **true** if all non-fixed cells in that column are selected. If they are not all selected, it returns **false**.

- The current column but not the row is in the fixed area, setting the **selected** property sets all non-fixed cells in that row to the selected status. Getting the status in that situation returns **true** if all non-fixed cells in that row are selected. If they are not all selected, it returns **false**.

When the user clicks a non-fixed cell, all other selected cells are set to **false**, and the value of the **selected** property for the cell that is clicked is set to **true**. When the user clicks a non-fixed cell with the Shift key held down, the selected status of that cell is toggled. (The same effect can be achieved by using the arrow keys.)

When the user clicks a fixed column cell, all other selected cells are set to **false**, and the value of the **selected** property for all non-fixed cells in the clicked column is set to **true**. When the user clicks a fixed row cell, all other selected cells are set to **false**, and the value of the **selected** property of all non-fixed cells in the clicked row is set to **true**.

Both the Shift and the Ctrl keys can be used when a cell is selected in a table. Clicking a cell with the Ctrl key down toggles the selection of the clicked cell. Clicking a cell with the Shift key down causes the selection of all cells from the currently selected cell up to the cell that is being clicked when they are in the same row or column. (If they are not in the same row or column, only the cell that is clicked is selected.)

The value of the **selectMode** property also restricts the use of the Shift and Ctrl keys when selecting cells in tables. Use Ctrl+space bar to toggle the **selected** property status of the current cell, unless the style defined in the current **selectMode** property overrides that.

The code fragments in the following examples show the use of the **selected** property.

```
obj := tblPortfolio.itemObject;
if obj = null or not tblPortfolio.selected then
    app.msgBox("You should select a share to sell", "Warning",
               MsgBox_Exclamation_Mark_Icon + MsgBox_OK_Only);
    return;
endif;

if tblPortfolio.selected then
    if tblPortfolio.itemObject <> null then
        saveObject := tblPortfolio.itemObject;
    endif;
endif;
```

selectionStyle

Type: Integer

Availability: Read or write at any time

The **selectionStyle** property determines whether selected text in a **TextBox** or **JadeRichText** control remains highlighted when a control loses and gains the focus.

For the **JadeEditMask** control, this property determines whether a text box portion of the edit mask control that receives focus is selected.

The settings of the **selectionStyle** property for the **TextBox** and **JadeRichText** classes are listed in the following table.

TextBox Class Constant	Value	Description
SelectionStyle_Retain	0	Selected text appears selected when the control loses the focus (the default).

TextBox Class Constant	Value	Description
SelectionStyle_Hide	1	Selected text does not appear selected when the control loses the focus. If the text box gains focus by means other than a mouse click, the selected text display is restored.
SelectionStyle_SelectAll	2	All text in the control is automatically selected when the control gains the focus by means other than a mouse click. The data entry position (the **|** vertical bar) is located after the last character of the text. When focus is lost, the selection display is removed.
SelectionStyle_SelectAllAlways	3	For the **TextBox** control only, the entire text is selected, regardless of how the focus was achieved when the text box gains focus (by logic, the mouse, or keyboard). The data entry position (the **|** vertical bar) is located after the last character of the text. When focus is lost, the selection display is removed.

Use the **selectionStyle** property to indicate which text is selected while another form or control has the focus or when the text box gains the focus by means other than a mouse click. Changing this value at run time causes the text box to be recreated in the new style. The text is retained, but other runtime attributes (for example, the **selText** property) are reset.

The settings of the **selectionStyle** property for the **JadeEditMask** class are listed in the following table.

Constant	Value	Description
SelectionStyle_None	0	Text box portion of the edit mask is not selected (default value).
SelectionStyle_Select	1	Text box portion of the edit mask control that receives focus is selected, except when using the mouse to gain focus.
SelectionStyle_Select_Always	2	Entire text box portion of the edit mask control that receives focus is selected, regardless of how the focus was achieved when the text box gains focus (by logic, the mouse, or keyboard). The data entry position (the **|** vertical bar) is located after the last character of the text. When focus is lost, the selection display is removed

selectMode

Type: Integer

Availability: Read or write at any time

The **selectMode** property of the **Table** class enables you to control the selections that are made automatically when the user clicks a cell of the table using the mouse or the keyboard.

Your logic can set the **selected** property value of any cell, regardless of the value contained in the **selectMode** property.

When you set the **selected** property of a fixed cell from JADE logic, the values of all non-fixed cells in that row or column are also set to that value.

The settings of the **selectMode** property, listed in the following table, have meaning when a cell is clicked with the mouse or keyboard (if the Shift or Ctrl key is not also pressed). The selection status of all cells is first cleared.

Table Class Constant	Value	Description
SelectMode_Default	0	The default value, which specifies that the cell selection is turned on for a non-fixed cell and for a fixed row, all non-fixed cells in the whole row are selected. For a cell in a fixed column, all non-fixed cells in that column are selected.
SelectMode_FixedRow	1	Same as **SelectMode_Default** except that selecting a cell in a fixed row has no effect.
SelectMode_FixedColumn	2	Same as **SelectMode_Default** except that selecting a cell in a fixed column has no effect.
SelectMode_Multiple	3	Multiple cells can be selected in any row or column. Clicking on a fixed cell has no impact on selection.
SelectMode_Row	4	Multiple cells can be selected only in the same row. Clicking on a fixed cell has no impact on selection.
SelectMode_Column	5	Multiple cells can be selected only in the same column. Clicking on a fixed cell has no impact on selection.
SelectMode_Single	6	Only one cell can be selected at a time. Clicking on a fixed cell has no impact on selection.
SelectMode_None	7	Clicking anywhere on the table has no impact on selection.
SelectMode_CurrentRow	8	When the **selectMode** property is set to **SelectMode_CurrentRow**, the current row of the sheet of the table is always considered to be selected. This results in all the cells of the row being shown as selected. The selected status of any cell not in the current row is always **false** and the selected status of any cell in the current row is always **true**. A user cannot affect this status by logic or by a mouse or keyboard action. The only change occurs when a new row is selected by logic or by a user action, where the selection rules then apply to the new current row. Note that any logic attempting to change the selection status of any cell, row, column, or sheet is ignored.
SelectMode_WholeRows	9	When the **selectMode** property is set to **SelectMode_WholeRows**, the selected status of all cells in a row is affected when the user clicks on any cell in the row, the **selected** property of any cell is changed, or the user uses a keyboard arrow key to change rows. This selection style allows multiple rows to be selected at once, by using the keyboard, the mouse (using the Shift or Ctrl key), or logic.
SelectMode_CurrentColumn	10	When the **selectMode** property is set to **SelectMode_CurrentColumn**, the current column of the sheet of the table is always considered to be selected. This results in all the cells of the column being shown as selected.

Table Class Constant	Value	Description
		The selected status of any cell not in the current column is always **false** and the selected status of any cell in the current column is always **true**. A user cannot affect this status by logic or by a mouse or keyboard action. The only change occurs when a new column is selected by logic or by a user action, where the selection rules then apply to the new current column. Note that any logic attempting to change the selection status of any cell, row, column, or sheet is ignored.
SelectMode_WholeColumns	11	When the **selectMode** property is set to **SelectMode_WholeColumns**, the selected status of all cells in a column is affected when the user clicks on any cell in the column, the **selected** property of any cell is changed, or the user uses a keyboard arrow key to change columns. This selection style allows multiple columns to be selected at once, by using the keyboard, the mouse (using the Shift or Ctrl key), or logic.

Clicking a cell with the Ctrl key down toggles the selection of that cell. (See also the **selected** property.)

Clicking a cell with the Shift key down causes the selection of all cells from the currently selected cell up to the cell that is being clicked when they are in the same row or column. (If they are not in the same row or column, only the cell that is clicked is selected.)

The code fragment in the following example shows the use of the **selectMode** property.

```
    table1.selectMode := comboSelectMode.listIndex - 1;
```

selBackColor

Type: Integer

Availability: Read or write at any time (**JadeTextEdit**); read or write at run time only (**JadeRichText**)

The **selBackColor** property of the **JadeRichText** and **JadeTextEdit** controls specifies the background color of text selected in the text editor. The **selBackColor** property is not available at design time for **JadeRichText** controls.

The default background color is the **Window** class system highlight color represented by the **Window** class **Color_Highlight** constant.

The default value of the **JadeTextEdit** class is represented by the **ATTRIB_DEFAULT** (-1) **JadeTextEdit** class constant.

The value of the property in a **JadeRichText** control can be an RGB value in the range **0** through **0xFFFFFF** or it can be one of the **JadeRichText** class constants listed in the following table.

JadeRichText Class Constant	Value	Comment
CharacterFormat_AutoColor	-1	Text is drawn using the text color of the system (the default), with the background of the text drawn with the defined value of the **backColor** property
CharacterFormat_Undefined	#80000000	Selected text contains a mixture of colors (this value cannot be set)

The following code fragments are examples of the **JadeRichText** class **selBackColor** property.

```
rtfControl.selStart := 10;
rtfControl.selLength :=  6;
rtfControl.selBackColor := Red;
// sets the background color of the characters 10 through 15 in the control to Red

rtfControl.selBackColor := Green;
rtfControl.load("Monday ", JadeRichText.Load_Append);
// sets the background color of the appended text 'Monday' to Green
```

selFontBold

Type: Integer

Availability: Not available at design time, read or write at run time

The **selFontBold** property of the **JadeRichText** control specifies whether the font style of the selected text is bold. The settings of the **selFontBold** property are listed in the following table.

JadeRichText Class Constant	Value	Comment
CharacterFormat_NotSet	0	The default value
CharacterFormat_Set	1	
CharacterFormat_Undefined	#80000000	The selected text contains both bold and non-bold characters

selFontItalic

Type: Integer

Availability: Not available at design time, read or write at run time

The **selFontItalic** property of the **JadeRichText** control specifies whether the font style of the selected text is italics. The settings of the **selFontItalic** property are listed in the following table.

JadeRichText Class Constant	Value	Comment
CharacterFormat_NotSet	0	The default value
CharacterFormat_Set	1	
CharacterFormat_Undefined	#80000000	The selected text contains both italic and non-italic characters

selFontName

Type: String

Availability: Not available at design time, read or write at run time

The **selFontName** property of the **JadeRichText** control specifies the name of the font used for the selected text.

The default value is determined by the system. Fonts that are available with JADE vary, according to your system configuration, display devices, and printing devices.

Although you cannot set this property to null, a null value (**""**) indicates that the selected text contains multiple characters with different font names.

selFontSize

Type: Real

Availability: Not available at design time, read or write at run time

The **selFontSize** property of the **JadeRichText** class contains the size of the font used for the selected text. The default value is determined by the system. Fonts that are available with JADE vary, according to your system configuration, display devices, and printing devices.

Although you cannot set this property to **CharacterFormat_Undefined** (#80000000), this value indicates that the selected text contains multiple characters with different font sizes.

selFontStrikethru

Type: Integer

Availability: Not available at design time, read or write at run time

The **selFontStrikethru** property of the **JadeRichText** control specifies whether the strikethrough font attribute is applied to the selected text.

The settings of the **selFontStrikethru** property are listed in the following table.

JadeRichText Class Constant	Value	Comment
CharacterFormat_NotSet	0	The default value
CharacterFormat_Set	1	
CharacterFormat_Undefined	#80000000	Selected text contains both strikethrough and non-strikethrough characters (this value cannot be set)

selFontUnderline

Type: Integer

Availability: Not available at design time, read or write at run time

The **selFontUnderline** property of the **JadeRichText** control specifies whether the underline font attribute is applied to the selected text.

The settings of the **selFontUnderline** property are listed in the following table.

JadeRichText Class Constant	Value	Comment
CharacterFormat_NotSet	0	The default value
CharacterFormat_Set	1	
CharacterFormat_Undefined	#80000000	Selected text contains both underlined and non-underlined characters (this value cannot be set)

selFontUnderlineType

Type: Byte

Availability: Not available at design time, read or write at run time

The **selFontUnderlineType** property of the **JadeRichText** control specifies whether the underline font attribute is applied to the selected text and the form in which the underline is drawn. The use of this property is the same as **selFontUnderline**, except that it also controls the style in which the underline is drawn.

Note Only use one of the **selFontUnderline** and **selFontUnderlineType** properties when defining the underline settings.

The setting of the **selFontUnderlineType** property can be one of the **JadeRichText** class constants listed in the following table.

JadeRichText Class Constant	Value	Comment
Underline_Type_None	0.Byte	No underline is drawn (equivalent to **selFontUnderline := false**)
Underline_Type_Underline	1.Byte	Standard underline (equivalent to **selFontUnderline := true**)
Underline_Type_Dotted	4.Byte	Underline is dotted
Underline_Type_Dash	5.Byte	Underline is dashed
Underline_Type_DashDot	6.Byte	Underline is dash, dot repeated
Underline_Type_DashDotDot	7.Byte	Underline is dash, dot, dot repeated
Underline_Type_Wave	8.Byte	Underline is a wavy line
Underline_Type_Thick	9.Byte	Underline is thick (doubled)
Underline_Type_Invert	254.Byte	Underline is an inverted color line

If the selected text contains more than one underline type, the **selFontUnderlineType** property is set to an undefined value. Other values are undefined but will mostly be drawn as a standard underline.

The following code fragments are examples of the **selFontUnderlineType** property.

```
rtfControl.selStart := 10;
rtfControl.selLength := 6;
rtfControl.selFontUnderlineType := JadeRichText.Underline_Type_Wave;
// underlines the characters 10 through 15 in the control with a wavy line

rtfControl.selFontUnderlineType := JadeRichText.Underline_Type_Dotted;
rtfControl.load("Monday ", JadeRichText.Load_Append);
// underlines the appended text 'Monday' with a dotted line.
```

selForeColor

Type: Integer[4]

Availability: Read or write at run time only

The **selForeColor** property of the **JadeTextEdit** control specifies the color of text selected in the text editor. The default value is **-1**, meaning transparent (that is, the syntax coloring is used).

If you set the value of this property, any selected text is displayed in this color, regardless of the syntax coloring that applies.

selLength

Type: Integer

Availability: Not available at design time, read or write at run time

The **selLength** property contains the number of characters selected in a **ComboBox**, **TextBox**, **JadeRichText**, or **JadeTextEdit** control. Use the **selLength** property for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in a control, or clearing text.

The valid range of settings is zero (**0**) through text length (the total number of characters in the edit area of the control). The result of setting the value of the **selLength** property to a value less than zero (**0**) is undefined.

Changing the value of the **selStart** property changes the selection to an insertion point and sets the value of the **selLength** property to zero (**0**). Setting the **selText** property to a new value sets the **selLength** property to zero (**0**) and replaces the selected text with the new string.

If the value of the **selStart** property is zero (**0**) and the value of the **selLength** property is **-1**, all text in the control is selected. If the value of the **selStart** property is **-1**, any current selection is deselected.

For the **JadeTextEdit** class, the **EVENTTYPE_SELECTIONSTATE** notification occurs when the selection changes from empty or to empty.

selLink

Type: Boolean

Availability: Not available at design time, read or write at run time

The **selLink** property of the **JadeRichText** control specifies whether the selected text is all marked as a URL link. You can use this property to define user-friendly name links. If the property is set to **true**, when the user clicks on the text, the **linkClicked** event method is called for the control and the event receives the text of the link. See also the **autoURLDetect** property, which when **true**, causes this process to occur automatically for text that can be recognized as a URL.

Set the **selLink** property to:

- **true**, to have the selected text drawn as a blue underlined URL. The text color and underlining cannot be controlled.
- **false**, to have the selected text drawn normally.

If the selected text is a mixture of a link and a non-link, this property returns **false**.

The following code fragments are examples of the **selLink** property.

```
rtfControl.selStart := 10;
rtfControl.selLength := 6;
rtfControl.selLink := true;
// causes the characters 10 through 15 to be drawn as a URL

rtfControl.selLink := true;
rtfControl.load("Monday ", JadeRichText.Load_Append);
// causes the appended text 'Monday' to be drawn as an underlined blue URL
```

selStart

Type: Integer

Availability: Not available at design time, read or write at run time

The **selStart** property of the **ComboBox**, **TextBox**, or **JadeRichText** class contains the starting point of selected text and indicates the position of the insertion point if no text is selected.

For the **JadeTextEdit** class, this property contains the zero-based character offset of the first character in the text selected in the control and indicates the position of the insertion point if no text is selected. A value of **-1** sets the property to the **currentPosition** property. If the value is past the end of the control, the value is set to the position of the last character of text.

Use this property for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in a control, or clearing text.

The valid range of settings is zero (**0**) through text length (the total number of characters in the edit area of a control). If the value of the **selStart** property is zero (**0**) and the value of the **selLength** property is **-1**, all text in the control is selected. If the value of the **selStart** property is **-1**, any current selection is deselected. Setting the **selStart** property to a value greater than the text length sets the property to the existing text length.

Changing the value of the **selStart** property changes the selection to an insertion point, causes the text at the **selStart** position to be visible if the user has entered text beyond the length of the text box, and sets the value of the **selLength** property to zero (**0**).

selText

Type: String

Availability: Not available at design time, read or write at run time

The **selText** property contains the string of the currently selected text, consisting of an empty string (**""**) if no characters are selected in a **ComboBox**, **TextBox**, **JadeRichText**, or **JadeTextEdit** control. Use this property for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in a control, or clearing text.

Setting the **selText** property to a new value sets the **selLength** property to zero (**0**) and replaces the selected text with the new string. If there is no text selected in the control, the text is inserted at the current caret position.

For the **JadeRichText** control, the **selText** property contains the string of the currently selected text as plain text format, consisting of an empty string (**""**) if no characters are selected. For the **JadeTextEdit** control, no end-of-line conversion is performed when the **selText** property is set or retrieved.

selTextColor

Type: Integer

Availability: Not available at design time, read or write at run time

The **selTextColor** property of the **JadeRichText** control contains the color of the currently selected text. (For an example of the use of this property, see "JadeRichText Control Method Example", earlier in this document.)

The **selTextColor** property can be an RGB value (in the range 0 through 0xFFFFFF) or it can be one of the values listed in the following table.

JadeRichText Class Constant	Value	Comment
CharacterFormat_AutoColor	-1	Text is drawn using the text color of the system (the default), with the background of the text drawn with the defined value of the **backColor** property
CharacterFormat_Undefined	#80000000	Selected text contains a mixture of colors (this value cannot be set)

selTextRTF

Type: String

Availability: Not available at design time, read or write at run time

The **selTextRTF** property of the **JadeRichText** control contains the selected text in rich text format, consisting of an empty string (**""**) if no characters are selected.

Use this property for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in a control, or clearing text.

Setting the **selTextRTF** property to a new value sets the **selLength** property to zero (**0**) and replaces the selected text with the new string.

sheet

Type: Integer

Availability: Read or write at run time for tables

The **sheet** property contains the index of the current sheet for the **Table** control. This sheet value is used in the access of the table control properties. Changing the **sheet** property does not affect the sheet that is displayed. Using the **topSheet** property can change the sheet that is displayed. Sheet access is **1**-relative.

The following example shows the use of the **sheet** property in a method that updates a table.

```
vars
    count, entry, row, sheet : Integer;
    companiesPerSheet        : Integer;
    company                  : Company;
begin
    // work out number of rows per sheet
    count := app.myMarket.allCompanies.size div 3;
    if (count * 3) < app.myMarket.allCompanies.size then
        companiesPerSheet := count + 1;
    else
        companiesPerSheet := count;
    endif;
    entry            := 1;
    count            := 0;
    tblPrices.sheet := 1;
    if reason = Price_Change and changedCompany <> null then
```

```
            findCompanyInTable(changedCompany, companiesPerSheet, sheet, row);
            return;
        endif;
        foreach company in app.myMarket.allCompanies do
            sharedLock(company);
            tblPrices.row        := tblPrices.row + 1;
            tblPrices.column     := 1;
            tblPrices.text       := company.shortName;
            // set column on item Object to company
            tblPrices.itemObject := company;
            tblPrices.column     := 2;
            tblPrices.text       := company.name;
            tblPrices.column     := 3;
            tblPrices.accessMode := Table.AccessMode_Column;
            tblPrices.alignment  := Table.Alignment_Right;
            tblPrices.text       := company.currentPrice.currencyFormat;
            tblPrices.column     := 4;
            tblPrices.accessMode := Table.AccessMode_Column;
            tblPrices.alignment  := Table.Alignment_Right;
            tblPrices.text       := company.availableShares.String;
            entry                := entry + 1;
            if count = 0 then
               tblPrices.sheetCaption := company.shortName;
            endif;
            count := count + 1;
            if count >= companiesPerSheet then
                count := 0;
                tblPrices.rows := tblPrices.row;
                tblPrices.sheetCaption := tblPrices.sheetCaption &
                          " - " & company.shortName;
                if tblPrices.sheet < 3 then
                    tblPrices.sheet := tblPrices.sheet + 1;
                endif;
            endif;
            unlock(company);
        endforeach;
        if count <> 0 then
            tblPrices.rows := tblPrices.row;
        endif;
    end;
```

sheetCaption

Type: String

Availability: Read or write at run time only for tables

The **sheetCaption** property contains the caption for the current sheet of a **Table** control.

This caption is displayed in the tabs area of the table. It is contained in the portion of the table on which the user clicks to make a specific sheet visible.

The code fragment in the following example shows the use of the **sheetCaption** property.

```
    tblPrices.sheetCaption := company.shortName;
```

sheets

Type: Integer

Availability: Read or write at any time

The **sheets** property contains the number of sheets for a **Table** control. Changing the value of this property adds additional sheets to the table or deletes excess sheets, discarding their contents.

The value of the **sheets** property cannot be set to zero (**0**). New sheets added to the table adopt the property defaults value for the current sheet.

The following example shows the use of the **sheets** property.

```
vars
    count : Integer;
begin
    count := 1;
    while count <= tblPrices.sheets do
        tblPrices.sheet   := count;
        tblPrices.columns := 3;
        tblPrices.rows    := totalRows;
        tblPrices.row     := 1;
        tblPrices.column  := 1;
        tblPrices.text    := "Product";
        tblPrices.column  := 2;
        tblPrices.text    := "Price";
        tblPrices.column  := 3;
        tblPrices.text    := "Available";
        tblPrices.columnWidth[ 1 ] := tblPrices.width.Integer div 6;
        tblPrices.columnWidth[ 2 ] := tblPrices.width.Integer div 3;
        tblPrices.columnWidth[ 3 ] := tblPrices.width.Integer div 6;
        count := count + 1;
    endwhile;
    // allow space for the vertical scroll bar
    tblPrices.width := tblPrices.width + 18;
end;
```

See also the **sheets** method of the **Folder** control.

sheetVisible

Type: Boolean array

Availability: Read or write at run time only

The **sheetVisible** property enables the visibility of a sheet of a **Table** control to be accessed.

Hiding a sheet forces a repositioning of the tabs area of the table. If one sheet only is visible, no tabs are displayed.

Deleting the only visible sheet automatically causes the first sheet in the table to be made visible.

shortName

Type: String [100]

Availability: Read or write at run time only

The **shortName** property contains the short name of the OLE object in an **OleControl** class. The short name defaults to the OLE class or short file name used to create the object.

The **shortName** property allows the object to have an identifying short description assigned to the control and OLE object.

show3D

Type: Integer

Availability: Read or write at any time

The **show3D** property of the **Control** class is used to control whether automatic three-dimensional (3D) effects are added to the appearance of a control. The **show3D** property is independent of controls that have inbuilt 3D effects; for example, **Frame** or **Button** controls.

The **Form** sheet of the Define Application dialog in the JADE development environment enables you to specify the default value of the **show3D** property for new controls. For details about drawing borders on individual controls, see the **borderStyle** property.

The **show3D** property is ignored for **ActiveXControl** and **Ocx** controls so that they do not have two borders: one drawn by JADE and another drawn by the **ActiveXControl** or **Ocx** control itself.

The **show3D** property is set automatically to **Show3D_UseBorderStyle** (3) for the **ActiveXControl**, **BrowseButtons**, **Button**, **CheckBox**, **Folder**, **Frame**, **JadeDockBar**, **JadeDockContainer**, **JadeMask**, **MultiMedia**, **Ocx**, **OptionButton**, **HScroll** and **VScroll**, **Sheet**, and **StatusLine** controls, which ignore the **show3D** property.

The **show3D** property is available only in the JADE development environment for the **ComboBox**, **GroupBox**, **Label**, **ListBox**, **Picture**, **Table**, and **TextBox** controls. You can use the **Control** class **is3D** method to return whether the control was drawn three-dimensionally.

Note For performance reasons, the application default three-dimensional options are not dynamic at run time. The three-dimensional default value for each control class is obtained the first time each class is referenced. Changes to the three-dimensional default values of the application do not take effect until the application is next initiated.

The settings of this property are listed in the following table.

Control Class Constant	Value	Description
Show3D_UseAppDefault	0	Use application-defined default value for each control class
Show3D_Not3D	1	The control is not displayed as 3D, regardless of the default value
Show3D_Use3D	2	The control is displayed with 3D, regardless of the default value
Show3D_UseBorderStyle	3	Use only the **borderStyle** property to determine the border

The 3D effects can be thought of as a border that is two pixels wide.

Note If the **show3D** property is set to **Show3D_UseAppDefault** (0) and the application default setting of the **show3D** property for the control is **true** (that is, the control is selected in the **3D Controls** list box on the **Form** sheet of the Define Application dialog) or if the **show3D** property for the control is set to **Show3D_Use3D**, the effective border is a sunken three-dimensional effect regardless of the setting of the **borderStyle** property. If the **borderStyle** property is set to **BorderStyle_3DSunken** (2) or **BorderStyle_3DRaised** (3), the **show3D** property is reset to **Show3D_UseBorderStyle** (3).

The **Window** class **borderStyle** property is ignored when the **show3D** property is set to **Show3D_Use3D** (2) or **Show3D_UseAppDefault** (0) and the application default is 3D. When painting the control, you should make allowance for the increased border size.

The following example shows the use of the **show3D** property in the constructor for a customized control.

```
create() updating;
begin
    isBuilt     := false;
    changeType  := ChangeType_None;
    borderStyle := BorderStyle_Single;   // Ignored, because of setting
                                         // of the show3D property
    show3D      := Show3D_Use3D;
    transparent := true;
end;
```

showFocus

Type: Boolean

Availability: Read or write at run time only

The **showFocus** property of the **Table** class specifies whether the focus rectangle is shown on the current cell of a table when it has focus. The default value is **true**.

Set the **showFocus** property to **false** if you want the table painted without a focus rectangle displayed.

Notes A table that has the **readOnly** property set to **true** will not display the focus rectangle, regardless of the value of the **showFocus** property.

The focus rectangle is not shown if the current cell is within the fixed rows or columns of the table.

showMdiCloseAllButPinnedMenu

Type: Boolean

Availability: Read or write at any time

The **showMdiCloseAllButPinnedMenu** property of the **Form** class specifies whether the **Close All But Pinned** command is displayed in the popup (context) menu when a user right-clicks on the caption of an MDI child form or the tab associated with the form if the value of the **Application** class **mdiStyle** property is **MdiStyle_Mdi_With_Tabs** (1) or **MdiStyle_Tabs_Only** (2).

The **showMdiCloseAllButPinnedMenu** property (and the corresponding **Close All But Pinned**command) is disabled if the value of the **Application** class **mdiStyle** property has the default value of **MdiStyle_Mdi** (0).

The default value is **false**, which means that the popup menu is not displayed by right-clicking the MDI child caption. When this property is set to **true**, selecting the **Close All But Pinned** command closes all MDI child forms in the current MDI frame that are not pinned and that have the **allowClose** property set to **true**.

Notes The **showMdiCloseAllButPinnedMenu** property is ignored if the form is not an MDI child form or the **allowClose** property is set to **false**.

Right-clicking the MDI child form caption does not display a menu if all of the **showMdiCloseAllButPinnedMenu**, **showMdiCloseAllButThisMenu**, **showMdiCloseMenu**, **showMdiDockMenu**, **showMdiFloatMenu**, and **showMdiPinMenu** properties are set to **false**.

Applies to Version: 2020.0.01 and higher

showMdiCloseAllButThisMenu

Type: Boolean

Availability: Read or write at any time

The **showMdiCloseAllButThisMenu** property of the **Form** class specifies whether the **Close All But This** command is displayed in the popup (context) menu when a user right-clicks on the caption of an MDI child form or the tab associated with the form if the value of the **Application** class **mdiStyle** property is **MdiStyle_Mdi_With_Tabs** (1) or **MdiStyle_Tabs_Only** (2).

The default value is **false**, which means that the popup menu is not displayed by right-clicking the MDI child caption. When this property is set to **true**, selecting the **Close All But This** command closes all other MDI children in the current MDI frame that have the **allowClose** property set to **true**, except for the current form.

Notes The **showMdiCloseAllButThisMenu** property is ignored if the form is not an MDI child form or the **allowClose** property is set to **false**.

Right-clicking the MDI child form caption does not display a menu if all of the **showMdiCloseAllButPinnedMenu**, **showMdiCloseAllButThisMenu**, **showMdiCloseMenu**, **showMdiDockMenu**, **showMdiFloatMenu**, and **showMdiPinMenu** properties are set to **false**.

Applies to Version: 2020.0.01 and higher

showMdiCloseMenu

Type: Boolean

Availability: Read or write at any time

The **showMdiCloseMenu** property of the **Form** class specifies whether the **Close** command is displayed in the popup (context) menu when a user right-clicks on the caption of an MDI child form or the tab associated with the form if the value of the **Application** class **mdiStyle** property is **MdiStyle_Mdi_With_Tabs** (1) or **MdiStyle_Tabs_Only** (2).

The default value is **false**, which means that the popup menu is not displayed by right-clicking the MDI child caption. When this property is set to **true**, selecting the **Close** command closes the MDI child form that has the **allowClose** property set to **true** in the current MDI frame.

Notes The **showMdiCloseMenu** property is ignored if the form is not an MDI child form or the **allowClose** property is set to **false**.

Right-clicking the MDI child form caption does not display a menu if all of the **showMdiCloseMenu**, **showMdiCloseAllButThisMenu**, **showMdiCloseAllButPinnedMenu**, **showMdiDockMenu**, **showMdiFloatMenu**, and **showMdiPinMenu** properties are set to **false**.

Applies to Version: 2020.0.01 and higher

showMdiDockMenu

Type: Boolean

Availability: Read or write at any time

The **showMdiDockMenu** property of the **Form** class specifies whether the **Dock** command is displayed in the popup (context) menu when a user right-clicks on the caption of an MDI child form or the tab associated with the form if the value of the **Application** class **mdiStyle** property is **MdiStyle_Mdi_With_Tabs** (1) or **MdiStyle_Tabs_Only** (2). The default value of this property is **false**.

The **Dock** command is disabled if the MDI child form is already docked in the MDI frame.

Notes The **showMdiDockMenu** property is ignored if the form is not an MDI child form or the **allowClose** property is set to **false**.

Right-clicking the MDI child form caption does not display a menu if all of the **showMdiDockMenu**, **showMdiCloseAllButPinnedMenu**, **showMdiCloseAllButThisMenu**, **showMdiCloseMenu**, **showMdiFloatMenu**, and **showMdiPinMenu** properties are set to **false**.

Applies to Version: 2020.0.01 and higher

showMdiFloatMenu

Type: Boolean

Availability: Read or write at any time

The **showMdiFloatMenu** property of the **Form** class specifies whether the **Float** command is displayed in the popup (context) menu when a user right-clicks on the caption of an MDI child form or the tab associated with the form if the value of the **Application** class **mdiStyle** property is **MdiStyle_Mdi_With_Tabs** (1) or **MdiStyle_Tabs_Only** (2).

The default value is **false**, which means that the popup menu is not displayed by right-clicking the MDI child caption. When this property is set to **true**, selecting the **Float** command floats the current MDI child form that has the **allowClose** property set to **true**; that is, it takes the MDI child form out of the MDI frame and allows it to be moved independently from the MDI frame (for example, on to another monitor on the PC). The floated MDI child form is made a window's child of the frame and will then always be above the MDI frame in z-order.

The **Float** command is disabled if the MDI child form is currently floating.

Notes The **showMdiFloatMenu** property is ignored if the form is not an MDI child form or the **allowClose** property is set to **false**.

Right-clicking the MDI child form caption does not display a menu if all of the **showMdiFloatMenu**, **showMdiCloseAllButPinnedMenu**, **showMdiCloseAllButThisMenu**, **showMdiCloseMenu**, **showMdiDockMenu**, and **showMdiPinMenu** properties are set to **false** (the default).

Applies to Version: 2020.0.01 and higher

showMdiPinMenu

Type: Boolean

Availability: Read or write at any time

The **showMdiPinMenu** property of the **Form** class toggles status of the **Pin** command displayed in the popup (context) menu when a user right-clicks on the caption of an MDI child form or the tab associated with the form if the value of the **Application** class **mdiStyle** property is **MdiStyle_Mdi_With_Tabs** (1) or **MdiStyle_Tabs_Only** (2).

The default value is **false**, which means that the popup menu is not displayed by right-clicking the MDI child caption. When this property is set to **true**, selecting the **Pin** command toggles the pinned status of a MDI child form. All pinned MDI child form tabs are displayed to the left of all non-pinned form tabs. Pinned tabs have a pin icon displayed. A check mark is displayed at the left of the **Pin** command if the tab is pinned. Clicking on the pinned icon in the tab also unpins the tab.

The **showMdiPinMenu** property (and the corresponding **Pin**command) is disabled if the value of the **Application** class **mdiStyle** property has the default value of **MdiStyle_Mdi** (0).

Notes The **showMdiCloseAllButThisMenu** property is ignored if the form is not an MDI child form or the **allowClose** property is set to **false**.

Right-clicking the MDI child form caption does not display a menu if all of the **showMdiCloseAllButPinnedMenu**, **showMdiCloseAllButThisMenu**, **showMdiCloseMenu**, **showMdiDockMenu**, **showMdiFloatMenu**, and **showMdiPinMenu** properties are set to **false**.

Applies to Version: 2020.0.01 and higher

showMenu

Type: Boolean

Availability: Read or write at any time

The **showMenu** property of the **OleControl** class specifies whether the control displays a popup menu when the right mouse button is clicked over the inactive control of the verbs supported by the application that can be selected (usually edit, play, or open). The default value is **true**.

For the **MultiMedia** class, the **showMenu** property specifies whether a menu bar is displayed on the playbar of the control. This menu contains items that allow the volume, size, speed, and configuration to be altered for an MP3 file type. If the **showMenu** property and the **useDotNetVersion** property are set to **true**, the menu displayed when the menu button is clicked shows the view, volume and speed menu items but not the copy, and configure menu items, as well as a **Close** menu item that closes the medium being shown. If the value of the **showPlayBar** property is **false**, this property is ignored.

The default value is **false**See also the **contextMenu** event method.

showMode

Type: Boolean

Availability: Read or write at any time

The **showMode** property of the **MultiMedia** class specifies whether the control includes a caption, whose display includes the current status of the device. The default value is **false**.

See also the **showName** and **showPosition** properties.

showName

Type: Boolean

Availability: Read or write at any time

The **showName** property of the **MultiMedia** class specifies whether the control includes a caption, whose display includes the current name of the medium in the device.

The default value is **false**.

See also the **showMode** and **showPosition** properties.

showOpenMenu

Type: Boolean

Availability: Read or write at any time

The **showOpenMenu** property of the **MultiMedia** class specifies whether a menu item allowing the user to open another file in the control is displayed on the menu button menu of the playbar of the control.

If the value of the **showPlayBar** or **showMenu** property is **false**, this property is ignored.

The default value is **false**.

showPlayBar

Type: Boolean

Availability: Read or write at any time

The **showPlayBar** property of the **MultiMedia** class specifies whether a playbar is included in the control. This playbar enables the user to manually start and stop the playing of the medium in the control.

Note If the playbar is visible and the **useDotNetVersion** property is set to **true**, it is drawn using WPF entities and it has a different appearance from the playbar drawn using an MP3 file type.

The default value is **false**.

showPosition

Type: Boolean

Availability: Read or write at any time

The **showPosition** property of the **MultiMedia** class specifies whether the control includes a caption whose display includes the current position of the medium in the device.

The default value is **false**.

See also the **showMode** and **showName** properties.

showRecord

Type: Boolean

Availability: Read or write at any time

The **showRecord** property of the **MultiMedia** class specifies whether a record button is displayed on the playbar of the control.

If the **useDotNetVersion** property is set to **true**, the **showRecord** property is not available and it generates exception 1068 (*Feature not available in this release*).

The display of a record button is ignored if the device does not support recording or if the value of the **showPlayBar** property is **false**.

The default value is **false**.

showResizeBar

Type: Boolean

Availability: Read or write at any time

The **showResizeBar** property of the **JadeDockBase** class specifies whether resize bars can be added to the **JadeDockBar** or **JadeDockContainer** subclass control. The default value is **true**.

Resize bars allow the user to drag that bar to increase or decrease the size of the control. When the user drags the bar to a new position or resizes a floating **JadeDockBar** or **JadeDockContainer** control, the **userResize** event is called. For details about the rules for displaying a resize bar to the right or on the bottom of **JadeDockBar** and **JadeDockContainer** controls when the value of the **showResizeBar** property is **true**, see "Docking a Control" under "JadeDockBase Class", earlier in this document.

showTaskBarProgress

Type: Boolean

The **showTaskBarProgress** property of the **ProgressBar** control specifies whether the progress bar state is shown on the taskbar icon of the application as well as in the progress bar.

Note This functionality is available only if the application displays an icon on the Windows task bar. It does not apply to icons in the system tray.

The default value of **false** indicates that no progress is displayed on the taskbar icon. Setting the property value to **true** displays progress on the taskbar icon. The **Form** class **setTaskBarProgress** method is automatically called when the value of the **partsDone** property is updated.

When the progress bar value reaches 100 percent, the **Form** class **setTaskBarState** method with a parameter value of **TaskBar_State_NoProgress** is automatically called to hide the taskbar state display.

If the value of the property never reaches 100 percent, it is your responsibility to ensure that the taskbar progress state is hidden, if required.

The taskbar state is hidden when the form is unloaded.

sizeMode

Type: Integer

The **sizeMode** property of the **OleControl** class controls the automatic sizing of the control or image. The sizing values are listed in the following table.

OleControl Class Constant	Value	Description
SizeMode_ClipToControl	0	Show as much of the image as fits in the control (the default).
SizeMode_StretchToControl	1	Stretch the image to the size of the control.
SizeMode_AutoSizeControl	2	Stretch the control to the size of the image (occurs only when not active).
SizeMode_Proportional	3	Stretch the image to the size of the control that has been resized proportional to the dimensions of the object (the control never gets larger).

skinCategoryName

Type: String

Availability: Read or write at any time

The **skinCategoryName** property of the **Window** class contains the string name of the skin category. By default, the value of this property is null (**""**). Use this property to define each form and control with a skin category.

When you click on the **skinCategoryName** property on **Common** page of the Properties dialog in the JADE Painter, a combo box with a list of skin category names is displayed. The list box contains only those category names that are assigned to a skin of the same type or superclass type as the window currently selected in the Painter. The list box also contains a blank entry, to enable you to clear the current skin category name. When the required skin has not been loaded into the current development environment, you can also specify a skin category name that is not displayed in the list.

When an application skin is set, the window uses only a skin that matches the defined category. (For details about using skins to enhance your runtime applications, see Chapter 2 of the *JADE Runtime Application Guide*.)

As the linkage between the window and the skin category is deliberately flexible, you can replace or ignore skins (by calling the **Window** class **ignoreSkin** property) to meet your requirements.

See also **JadeSkinControl**::**applyCondition**, the **JadeSkinWindow** class **mySkinCategory** property, and the **JadeSkinApplication** class **myFormSkins** and **myControlSkins** properties, in Chapter 1.

smallChange

Type: Integer

Availability: Read or write at any time

The **smallChange** property contains the amount of change (in pixels) to the **value** property in a **ScrollBar** control when the user clicks a scroll arrow. You can specify any valid integer. By default, the **smallChange** property value is **1**.

The Windows environment automatically sets proportional scrolling increments for scroll bars on form windows, combo boxes, and list boxes, based on the amount of data in the control. For a scroll bar control, however, you must specify these increments.

Use the **smallChange** property to set scrolling increments appropriate to how the scroll bar is being used. Typically, you set the **smallChange** property in Painter in the JADE development environment. You can also reset it in logic at run time when the scrolling increment must change dynamically.

Use the **max** and **min** properties to set the maximum and minimum ranges of a scroll bar control.

sortAsc

Type: Boolean (list box), Boolean array (table)

Availability: Read or write at run time only

The **sortAsc** property controls whether the sorting of a **ListBox** or **Table** control is ascending or descending. The **sortAsc** property defaults to **true**. If the **sorted** property of the list box is not set, the **sortAsc** property has no meaning.

Each sheet of a table control can have in the range zero through six sorted columns. The sorting sets these properties for the sheet, as required. Each sheet of a **Table** control has an array of six items (the entries are **1**-relative).

The column numbers can be established at any time, and are not validated when set. If the column is invalid, it is ignored when sorted. If a sort column is zero (**0**), the remaining sort column values are ignored. If no sort column is set, no sorting is performed.

Note When the text of a sorted column changes, the automatic sorting of rows occurs only when the **Table** class **addItem** method adds a new row or the **Table** class **resort** method is used.

As the number of rows can initially be set, the table control keeps a record of the highest row that has had text set. The sort involves only the rows up to and including that row.

Fixed rows of a **Table** control are not sorted.

The following examples show the use of the **sortAsc** property.

```
//  toggle sort on columns ...
instancesTable.sortColumn[1] := table.column;
instancesTable.sortAsc[1]    := not instancesTable.sortAsc[1];
instancesTable.resort;

table1_dblClick(table: Table input) updating;
begin
    if table.row = 1 then
        table1.sortColumn[1] := table.column;
```

```
            table1.sortAsc[1]    := true;
            table.resort;
        endif;
    end;
```

The code fragment in the following example sorts the table based on three columns: **4**, **1**, and **7**. Column **4** is sorted in ascending order, column **1** in descending order, column **7** in ascending order, and no columns are sorted by case; that is, they are case-insensitive.

```
table1.sortColumn[1] := 4;     // first sort column is 4
table1.sortAsc[1]    := true;
table1.sortCased[1]  := false;
table1.sortColumn[2] := 1;     // second sort column is 1
table1.sortAsc[2]    := false;
table1.sortCased[2]  := false;
table1.sortColumn[3] := 7;     // third sort column is 7
table1.sortAsc[3]    := true;
table1.sortCased[3]  := false;
```

sortCased

Type: Boolean (list box), Boolean array (table)

Availability: Read or write at run time only

The **sortCased** property controls whether the sorting of a **ComboBox**, **ListBox**, or **Table** control is case-sensitive. The **sortCased** property defaults to **false**.

If the **sorted** property of the list box is not set, the **sortCased** property has no meaning.

Each sheet of a **Table** control can have in the range zero through six sorted columns. The sorting sets these properties for the sheet, as required. Each sheet has an array of six items (the entries are **1**-relative).

The column numbers can be established at any time and are not validated when set. If the column is invalid, it is ignored when sorted.

If a sort column is zero (**0**), the remaining sort column values are ignored. If no sort column is set, no sorting is performed. Fixed rows of a table control are not sorted.

Notes When the text of a sorted column changes, the automatic sorting of rows occurs only when the **Table** class **addItem** method adds a new row or the **Table** class **resort** method is used.

Windows sorts text in tables and list boxes by using the collating sequence of the locale that is in use. Uppercase and lowercase are sorted together into alphabetical order (for example, **A b B c C d D**).

As the number of rows can initially be set, the **Table** control keeps a record of the highest row that has had text set. The sort involves only the rows up to and including that row.

The code fragment in the following example shows the use of the **sortCased** property.

```
col := table2.accessColumn(7);
table2.sortColumn[1] := 7;
col.sortAsc := true;
col.sortCased := true;
```

The following table lists a column of characters and the results of the **sortCased** method.

Column to Sort	Sorted Column
B	a
e	A
d	b
b	B
C	c
a	C
D	d
c	D
A	e

sortColumn

Type: Integer array

Availability: Read or write at run time only

The **sortColumn** property of the **Table** control sets the column number for which the text is to be sorted. Each sheet of a table control can have in the range zero through six sorted columns. The sorting sets these properties for the sheet, as required. Each sheet has an array of six items (the entries are **1**-relative).

The column numbers can be established at any time and are not validated when set. If the column is invalid, it is ignored when sorted.

If a sort column is zero (**0**), the remaining sort column values are ignored. If no sort column is set, no sorting is performed.

Note When the text of a sorted column changes, the automatic sorting of rows occurs only when the **Table** class **addItem** method adds a new row or the **Table** class **resort** method is used. (The **resort** method is ignored if no columns are set; that is, when the **sortColumn** property is set to zero.)

As the number of rows can initially be set, the table control keeps a record of the highest row that has had text set. The sort involves only the rows up to and including that row. Fixed rows are not sorted.

The code fragment in the following example sorts the table based on three columns: **4**, **1**, and **7**. Column **4** is sorted in ascending order, column **1** in descending order, column **7** in ascending order, and no columns are sorted by case; that is, they are case-insensitive.

```
table1.sortColumn[1] := 4;    // first sort column is 4
table1.sortAsc[1]    := true;
table1.sortCased[1]  := false;
table1.sortColumn[2] := 1;    // second sort column is 1
table1.sortAsc[2]    := false;
table1.sortCased[2]  := false;
table1.sortColumn[3] := 7;    // third sort column is 7
table1.sortAsc[3]    := true;
table1.sortCased[3]  := false;
```

sorted

Type: Boolean

Availability: Read or write at any time

The **sorted** property specifies whether the elements of a **ComboBox** or **ListBox** control are automatically sorted alphabetically.

The settings of the **sorted** property are listed in the following table.

Value	List items are …
true	Sorted alphabetically (case-insensitive). This is the default value for a **ComboBox** control.
false	Not sorted alphabetically. This is the default value for a **ListBox** control.

When the **sorted** property is set to **true,** the control manages the entries and the sorting process. The **sortAsc** property controls whether the entries are sorted in ascending or descending order, and the **sortCased** property controls whether the sorting is case-sensitive.

Note Using the **addItem** method to add an element to a specific location in the list may violate the sort order, and subsequent additions may not be correctly sorted.

Setting the **sorted** property for a list box control sets the **hasPictures**, **hasPlusMinus**, and **hasTreeLines** properties to **false**.

sortType

Type: InternalPseudoArrayInteger

Availability: Read or write at run time only

The **sortType** property of the **Table** control specifies the type of data that the cell text represents. Use this property to control how to interpret the cell text when a table is sorted.

Each sheet of a **Table** control can have in the range zero through six sorted columns, and each of these columns has a corresponding **sortType** entry that defaults to normal alphanumeric sorting.

The values for the **sortType** property, represented by **Table** class constants, are listed in the following table.

Table Class Constant	Cell Data Type	Integer Value
SortType_Alpha	Default alphanumeric sorting	0
SortType_Numeric	Numeric sorting (including decimals and negative signs)	1
SortType_Date	Date	2
SortType_Time	Time	3
SortType_TimeStamp	Timestamp	4

The numeric evaluation uses the locale of the user for the separator, decimal place, and negative sign character handling. In addition:

- A leading **-** character is always treated as a negative sign, regardless of the locale definition and placement of the negative sign.

- A numeric value enclosed in parentheses (for example, **(123.45)**) is treated as negative (that is, **-123.45** for the preceding example).
- A numeric value is constructed from the cell text up to the first invalid character or the end of the string. For example, **'1,234.45invalid'** treats the numeric value as **'1,234.45'**. If there is no valid leading numeric portion, the sort uses zero (**0**) for the numeric value of the cell.

Notes When the **sortType** property has a value of **SortType_Time** or **SortType_TimeStamp**, text in the cells is typecast as a **String**. Time values must be in 24-hour format, or in 12-hour format with an **AM** or **PM** suffix. Timestamp entries must be in a format that produces a correct typecast; for example, **dd/MM/yyyy, hh:mm**, where the time part can be in 24-hour format, or in 12-hour format with an **AM** or **PM** suffix.

The conversion from the string to the sort type for sorting **Date** and **TimeStamp** primitive type-based columns is based on the current locale used by the application. (For details about using the current locale of the application for date and time formatting, see the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file.) If the conversion of the date part fails based on the locale format, JADE attempts to determine the format (*d/M/y*, *M/d/y*, or *y/M/d*) based on the type of data (that is, year length = 4 and whether the month is alpha).

If the text field of a cell is not a valid instance of the requested primitive type, the resulting sorting is undefined. The code fragment in the following example shows the use of the **sortType** property.

```
if index = 7 then
    table1.sortColumn[1] := 2;
    table1.sortType[1]   := table1.SortType_Numeric;
    table1.sortAsc[2]    := true;
    table1.sortCased[2]  := true;
elseif index = 8 then
```

You can also set the sort type by using the **JadeTableColumn** class **sortType** property.

speed

Type: Integer

Availability: Read or write at any time

The **speed** property of the **MultiMedia** class contains the playback speed of the device. As not all devices support this facility, its use raises an exception when it is not supported.

The default value is **1000**. A speed of **500** plays at half speed, a speed of **2000** at double speed, and so on.

stretch

Type: Integer

Availability: Read or write at any time for pictures, read or write at run time only for tables

The **stretch** property determines how a picture stretches to fit the size of a **Picture** control. The stretch property settings for a **Picture** control are listed in the following table.

Picture Class Constant	Value	Description
Stretch_None	0	The picture box size does not change to fit the picture size. If the picture is too large, the image is clipped.

Picture Class Constant	Value	Description
Stretch_ToControl	1	The picture is resized to fit the control size. This may cause distortion of the image.
Stretch_ControlTo	2	The control stretches to fit the picture size. If the picture is large, the control may be enlarged so that it overlaps other controls or no longer fits on the form.
Stretch_Proportional	3	The control is sized proportionally to the picture. The picture is then stretched to fit the control. This means that the picture is always drawn in isotropic proportions to its original size. When the control size is determined, the width or the height decreases (not increases) to keep the same proportional view of the picture.
Stretch_Pic_Proportional	4	The picture is sized proportionally to fit the control. The picture is centered horizontally or vertically if the resized picture does not fill the control height or width.
Stretch_CenterPicture	5	The picture is centered on the control.

If the value of the **stretch** property for a **Picture** control is non-zero, resizing the control also resizes the picture it contains. In addition, animated .**gif** picture files are not drawn stretched. The **stretch** property is used to resize the control to the GIF image if required, but the animation is always drawn at its normal size.

For an image containing transparency to display correctly in a **Picture** control or a **JadeMask** control as part of a Web-enabled application:

- Set the **stretch** property of the control to **Stretch_ToControl** (1)
- Do not use the drawing methods of the **Window** class (for example, the **drawLine** method) on the control
- In the case of a **JadeMask** control, do not set the **caption** property

Note Images that are mostly white or black lose their clarity when displayed in a **Picture** control using the **stretch** property and the picture control is smaller than the image. When an image is stretched, JADE sets the Microsoft stretch mode to **COLORONCOLOR**, which means all pixels are treated equally, and Windows drops pixels when the image is reduced in size. The exception is for a 1-bit image, where JADE calculates whether there were more white or black pixels and then sets the Microsoft stretch mode to **BLACKONWHITE** (white pixels are discarded instead of black pixels) or **WHITEONBLACK** (black pixels are discarded instead of white pixels).

From JADE 2018.0.02 and higher, JADE has been changed so that If the image consists of more than 50 percent of white pixels, JADE sets the stretch mode to **BLACKONWHITE**. Similarly, if the image consists of more than 50 percent of black pixels, JADE sets the stretch mode to **WHITEONBLACK**. This change improves the display of small images; for example, plan drawings.

For a **Table** control, the **stretch** property determines whether:

- Pictures placed in the cells of the table are drawn to fit the cell (after the text is displayed).
- Pictures are drawn to their actual size.
- A picture or text in a cell gets preference when there is insufficient space to fully display both the picture and the text.

The settings of the **stretch** property for a **Table** control are listed in the following table.

Table Class Control	Value	Description
Stretch_None	0	All pictures displayed in cells in the table are drawn at actual size. If the picture is too large to fit the available space in the cell, the picture is truncated on the right or the bottom. If there is insufficient room for the picture and the text within a cell, the picture is truncated.
Stretch_Cell (the default value)	1	All pictures are stretched proportional to the size of each cell less the size of the text.
Stretch_None_Picture_First	2	No cell picture is stretched. If there is insufficient room for the picture and the text within a cell, the text is truncated.
Stretch_Cell_Picture_First	3	All pictures are stretched proportional to the size of the cell but no bigger than the actual picture image. The text is then displayed in any remaining space in the cell and may be truncated.

The **stretch** property of the **Table** control applies globally to all cells on all sheets of the table.

The code fragment in the following example shows the use of the **stretch** property.

```
    table1.stretch := comboStretch.listIndex - 1;
```

style

Type: Integer

Availability: Read or write at any time

The **style** property determines the type of **ComboBox** control and the behavior of its list box portion. The settings of the **style** property for **ComboBox** controls are listed in the following table.

ComboBox Class Constant	Value	Description
Style_DropDown	0	Includes a drop-down list and an edit area. The user can select from the list or type in the edit area. Accessing the **text** property returns the contents of the edit area, which does not have to match an entry in the combo box list.
Style_Simple	1	Includes an edit area and a list that is always displayed. The user can select a list entry or type in the edit area. The size of a simple combo box includes both the edit and list portions. By default, a simple combo box is sized so that none of the list is displayed. Increase the **height** property to show more of the list.
Style_DropDownList	2	This style allows only selection from the drop-down list (the default). Accessing the **text** property returns the text of the currently selected entry (same as **itemText[listIndex]**).
Style_DropDownComboList	3	This function is the same as the combo box determined by the **Style_DropDown** (0) constant, except that the entered text is used only to select an entry. Accessing the **text** property returns the currently selected list entry (same as **itemText[listIndex]**).

ComboBox Class Constant	Value	Description
Style_SpinBox	4	This is the same as the **Style_DropDownList** (2) constant, except that the list box portion of the control is never displayed. Instead, the control displays a vertical scroll bar, with which the user can select the required entry.

If the **style** property is changed from or to **Style_DropDown** (1) at run time, the contents of the combo box are lost. See "ComboBox Class Constants", for details about the style constants provided by the **ComboBox** control.

The following guidelines assist in deciding on the setting of the **style** property to use for **ComboBox** controls.

- Use setting **Style_DropDown** (0) or setting **Style_Simple** (1) to give the user both a list of choices and the ability to enter a choice in the edit area. **Style_DropDown** (0) saves space on the form, because the list portion closes when the user makes a selection.
- Use setting **Style_DropDownList** (2) to display a fixed list of choices from which the user can select one entry.

  The list portion closes when the user selects an item.
- Use setting **Style_DropDownComboList** (3) instead of setting **Style_DropDownList** (2) if you also want to provide the user with the ability to partially enter the required entry so that it is automatically located.
- Use setting **Style_SpinBox** (4) if you do not want to have a drop-down list displayed and no text entered from the keyboard.

For a **Button** control, the **style** property specifies the style of push button. The settings of the **style** property for **Button** controls are listed in the following table.

Button Class Constant	Value	Description
Style_Normal	0	Click to press, and pops up when button released (the default). This setting does not affect any other button.
Style_Auto2State	1	Pressing this button causes it to stay down. It also causes all other auto two-state buttons with the same parent to come up. This acts exactly the same as the **OptionButton** control but provides a button appearance. If there is one auto two-state button only in the group, it behaves as it does for the **Style_2State** (2) setting.
Style_2State	2	Pressing the button causes it to go down if it was up, or to come up if it was down. It has no impact on any other button.

For a **Button** control whose **style** property is set to **Style_Auto2State** (1), setting the **value** property to **true** causes other automatic two-state buttons with the same parent to come up.

Setting the **value** property to **false** has no impact on other buttons in the same group.

The settings of the **style** property for **JadeMask** controls are listed in the following table.

JadeMask Class Constant	Value	Description
Style_Normal	0	Click to press, and pops up when the mask control is released (the default). This setting does not affect any other mask control.

JadeMask Class Constant	Value	Description
Style_Auto2State	1	Pressing this mask control causes it to go down and stay down. It also causes all other auto two-state mask controls with the same parent to come up; that is, only one mask control of a group can be down at any one time. This acts the same as the **OptionButton** control but provides a button appearance. If there is one auto two-state mask control only in the group, it behaves as it does for the **Style_2State** (2) setting.
Style_2State	2	Pressing the mask control causes it to go down and stay down if it was up, or to come up and stay up if it was down. It has no impact on any other mask control.
Style_Mask_Color	3	The **currentMaskColor** method returns the color of the pixel in the mask picture at the current mouse position. An example of the use of this style is a control that displays a map of a road network. The picture mask is built with the roads drawn in different colors and the rest of the map being some other color (for example, white). When the user moves the mouse over the control, the **mouseMove** event calls the **currentMaskColor** method. If the returned value is not **white**, the color is used to index the name of the road and this is displayed to the user in bubble help. When using this style, the roll over (**pictureRollOver**) and roll under (**pictureRollUnder**) features do not apply.

For a **JadeMask** control whose **style** property is set to **Style_Auto2State** (1), setting the **value** property to **true** causes other automatic two-state buttons with the same parent to come up. Setting the **value** property to **false** has no impact on other buttons in the same group.

tabActiveColor

Type: Integer

Availability: Read or write at any time

The **tabActiveColor** property of the **Folder** and **Table** classes contains the color that is drawn for the background area of the active tab of multiple sheet folders and tables. (For details about drawing the color of inactive tabs, see the **tabInactiveColor** property.)

The default value of **Color_3Dface** (which is usually light gray) is a Windows-imposed color.

The captions of active and inactive tabs are drawn by using the same font.

Tip Consider using **#e7e7e7** for the **tabActiveColor** property value, as this provides better resolution.

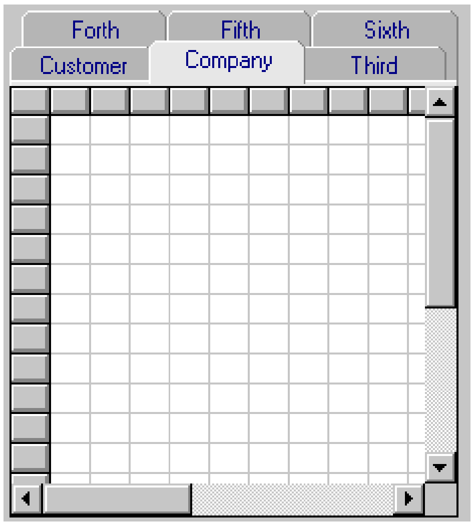
If you explicitly set the value of the **tabActiveColor** property of the active sheet of a folder (that is, you change it from the default **Windows.Color_3Dface** value), that color is always used to display the tab, regardless of the background color of the sheet.

If the value of the **tabActiveColor** property is set to the default **Windows.Color_3DFace** value, the tab is displayed in the same color as the background color of the sheet.

The tabs of folders and tables display a roll-over effect.

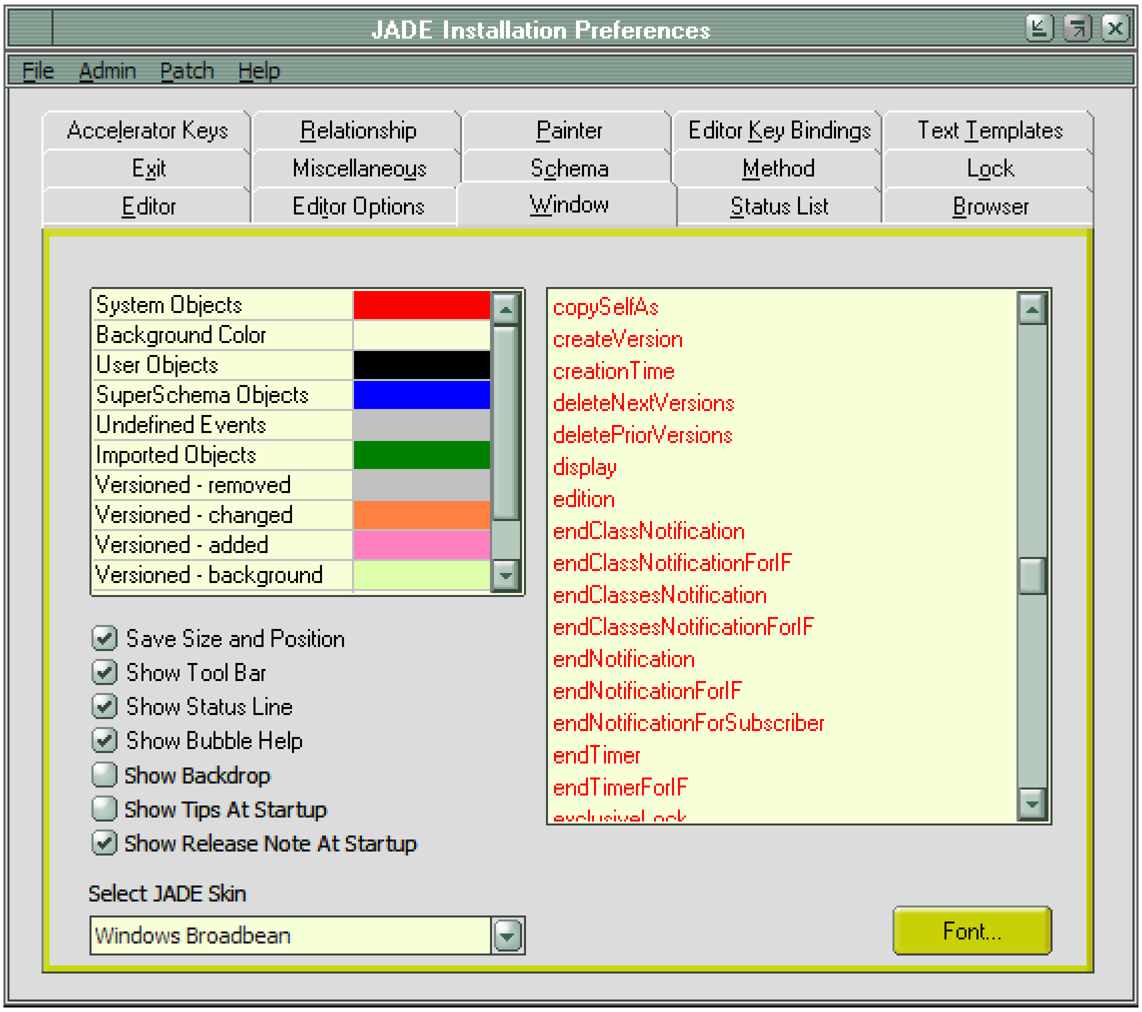
Moving the mouse over an inactive sheet tab causes the tab to be drawn using the Windows Info background color defined for the workstation.

If there are multiple tab rows on a **Table** control, the active tab is drawn three pixels higher, as shown in the following example.



If there are multiple rows on a folder, the active tab is slightly enlarged left and right by three pixels (unless it is on the edge of the table), to further highlight that it is the active tab.

The following is an example of a **Folder** control that has multiple tab rows.



System Objects	🟥
Background Color	⬜
User Objects	⬛
SuperSchema Objects	🟦
Undefined Events	⬜
Imported Objects	🟩
Versioned - removed	⬜
Versioned - changed	🟧
Versioned - added	🟪
Versioned - background	⬜

- [✓] Save Size and Position
- [✓] Show Tool Bar
- [✓] Show Status Line
- [✓] Show Bubble Help
- [ ] Show Backdrop
- [ ] Show Tips At Startup
- [✓] Show Release Note At Startup

Select JADE Skin: Windows Broadbean

copySelfAs
createVersion
creationTime
deleteNextVersions
deletePriorVersions
display
edition
endClassNotification
endClassNotificationForIF
endClassesNotification
endClassesNotificationForIF
endNotification
endNotificationForIF
endNotificationForSubscriber
endTimer
endTimerForIF

If the value of the **borderStyle** property for the active sheet of a **Folder** control is set to any value other than **BorderStyle_None**, the folder does not display a border of its own for the sheet area.

tabInactiveColor

Type: Integer

Availability: Read or write at any time

The **tabInactiveColor** property of the **Folder** and **Table** classes contains the color that is drawn for the background area of inactive tabs of multiple sheet folders and tables. (For details about drawing the color of the active tab, see the **tabActiveColor** property.) The default value of **Color_3Dface** (which is usually light gray) is a Windows-imposed color. The captions of active and inactive tabs are drawn by using the same font.

If you explicitly set the value of the **tabInactiveColor** property of the inactive sheet of a folder (that is, you change it from the default **Windows.Color_3Dface** value), that color is always used to display the tab, regardless of the background color of the sheet.

If the value of the **tabInactiveColor** property is set to the default **Windows.Color_3DFace** value, the tab is displayed in the same color as the background color of the sheet.

The tabs of folders and tables display a roll-over effect. Moving the mouse over an inactive sheet tab causes the tab to be drawn using the Windows Info background color defined for the workstation.

tabIndex

Type: Integer

Availability: Read or write at any time

The **tabIndex** property of the **Control** class contains the tab order of a control within its parent form. It is also used when processing an accelerator key.

If the accelerator is contained in the caption of a control that does not have the **tabStop** property set, the next focus is shifted to the next enabled and visible control in the order of the **tabIndex** property that has the **tabStop** property set.

Use the **tabIndex** property when assigning a visible accelerator (character underlined) in the caption of a label next to a **TextBox** control. The valid range is any integer. If the **tabIndex** property already exists for an existing control, the tab index of that control (and all other controls that are also affected) is increased by one.

By default, JADE assigns a tab order to controls as they are drawn on a form. Each new control is placed last in the tab order. If you change the **tabIndex** property value of a control to adjust the default tab order, JADE automatically renumbers the **tabIndex** property of other controls to reflect insertions and deletions. You can make changes in the JADE development environment by using the Properties window of the JADE Painter, or at run time in logic.

At run time, invisible or disabled controls and controls that cannot receive the focus (**Frame** and **Label** controls) remain in the tab order but are skipped during tabbing. The **tabIndex** property is not affected by the **zOrder** method.

Notes The tab order of a control affects its associated accelerator. If you press the accelerator key for a frame or a label, the focus moves to the next enabled and visible control in the tab order that can receive the focus. An accelerator is added to the caption of a control by placing an ampersand character (**&**) in front of the accelerator character. To activate that accelerator, the user presses Alt and the specified character key.

Controls are pasted on the form in ***controlList*** order (that is, the order in which you add them to the form). The order in this list can change during editing, because parent controls must always precede their children. Controls are copied to the Painter clipboard in **controlList** order. When pasted, each control is added to the form and a new **tabIndex** property is assigned. If the tab index of a control has not been used in the new form (the form on which you paste the control), the pasted control retains its tab index value. A control pasted on to a form can therefore have a tab index order that differs from the one that it had on the original form, although in most cases the tab order corresponds to that of the original form (particularly when pasting controls on to new forms).

For details about the order of controls on touchscreens when accessibility is set, see "Control Order on Touchscreens" under "Changing the Runtime Tab Sequence", in Chapter 5 of the *JADE Development Environment User's Guide.*

tabKey

Type: Integer

Availability: Read or write at any time

The **tabKey** property of the **Table** class implements tabbing within a table and contains the key that is used to tab within cells of a table. By default, the **tabKey** property is set to null (**0**), and movement within the table is controlled only from the keyboard by using the arrow keys.

When you set the **tabKey** property to a key code (accessed from the **keyCode** parameter in the **keyDown** event method), the next visible cell to the right of the current position in the table becomes the current cell when the specified key is pressed. (The **column** property is updated and the **rowColumnChg** event method is called if the **queryRowColChg** event method does not deny the change.)

If the current cell is the last visible cell in the row, pressing the specified tab key moves to the first visible non-fixed cell in the next row unless the current row is the last visible row, in which case the first visible non-fixed row becomes the current row.

If the Shift key is pressed with the key specified for the **tabKey** property, the previous visible cell in the current row becomes the current cell until the first visible non-fixed column is reached. If the first visible non-fixed cell in a row is already the current cell, pressing the Shift key and the key specified in the **tabKey** property moves to the last visible column of the previous row unless the current row is the first visible non-fixed row, in which case the last visible row becomes the current row. The key code can be any key, including the tab key specified by using the **J_key_Tab** global constant in the **KeyCharacterCodes** category. The **Table** control class intercepts the use of this key when the table or any of its children has focus. That key is then used only for tabbing within the table, and any other previous meaning to the table or its children is ignored.

To tab to the next control in the tab sequence when the **tabKey** property is set to the tab key (that is, the **J_key_Tab** global constant in the **KeyCharacterCodes** category, which has an integer value of **09**), use the Ctrl+Tab key combination. The Ctrl+Shift+Tab key combination tabs to the previous control in the tab sequence. This applies only to forms that are not MDI forms.

The code fragment in the following example shows the use of the **tabKey** property.

```
table1.tabKey := J_key_Tab;
```

tabsAlignment

Type: Integer

Availability: Read or write at any time

The **tabsAlignment** property controls the placement of the icon and text in the tab of a **Folder** control. The alignments are listed in the following table.

Folder Class Constant	Value	Description
TabsAlignment_Center	0	Icon is placed to the left of the text, and in the center of the tab area (the default).
TabsAlignment_Left	1	Icon (if present) is placed on the left edge of the tab, followed by the text of the tab.
TabsAlignment_IconLeft	2	Icon is placed at the left edge of the tab, and the text is centered. If the sheet has no icon, the alignment is the same as centered.

tabsFixedWidth

Type: Boolean

Availability: Read or write at any time

The **tabsFixedWidth** property controls the width of tabs of a **Folder** control.

The default value for this property is **false**, indicating that the tabs are given a width so that the icon and the text fit exactly. If the **tabsRaggedRight** property is set to **true**, that is the resulting displayed width. If the **tabsRaggedRight** property is set to **false**, the width of each tab is increased equally in each line, so that the tabs fill the entire available width.

If you set the **tabsFixedWidth** property to **true**, the tabs are all given the width needed to fit the largest icon and text combination. If the **tabsRaggedRight** property is set to **true**, that is the final resulting width.

If the **tabsRaggedRight** property is set to **false**, the width of each tab is increased equally in each line so that the tabs fill the entire available width. The tabs on each line have the same width, but if the tab lines have different numbers of entries, the line with a smaller number of entries has larger resulting tab widths.

tabsHeight

Type: Integer

Availability: Read or write at any time

The **tabsHeight** property contains the height of each line of tabs of a **Folder** control.

Note This property always returns the effective tab height.

The default value for this property is zero (**0**), indicating that the height of the tab lines is set according to the height of the text by using the font of the folder.

As the setting of this property to any positive value is interpreted as the number of pixels to use to display each tab line, use this property to provide a larger tab height so that the tab icons are displayed as a larger size.

Setting this property to a negative value specifies that no tabs are displayed for the folder. The user does not have any means of selecting which folder sheet to display, ensuring that the display of folder tabs is controlled totally by logic.

tabsLines

Type: Integer

Availability: Read or write at any time

The **tabsLines** property controls whether the tabs of a **Folder** control are displayed in multiple or single lines and whether simulated edges of the sheets are displayed.

The values for this property are listed in the following table.

Folder Class Control	Value	Description
TabsLines_MultiLineEdged	0	If required, each displaying a simulated stack of sheet edges (the default). This setting ignores the value of the **tabsRaggedRight** property, treating it as though it was set to **false**.

Folder Class Control	Value	Description
TabsLines_MultiLine	1	Multiple tab lines that do not display a simulated stack of sheet edges.
TabsLines_SingleLine	2	If the tabs cannot be fitted into one line, scroll arrows are displayed that can be clicked to scroll the appropriate tab entry into view.

If the **tabsStyle** property is set to **TabsStyle_Buttons** (1) or to **TabsStyle_RightSloped** (2), a **tabsLines** property setting of **TabsLines_MultiLineEdged** (0) is treated as if it were set to **TabsLines_MultiLine** (1); that is, multiple lines with no simulated sheet edges are displayed.

For the **TabsStyle_RightSloped** (2) setting, **TabsLines_MultiLineEdged** (**0**) draws the second rows increasingly offset from the left and there is no multiple-edged drawing of the sheets themselves.

If the resulting tab area uses more than half of the area of the folder, the tabs are automatically drawn in the **TabsLines_SingleLine** (2) mode.

tabsPosition

Type: Integer

Availability: Read or write at any time

The **tabsPosition** property determines where the folder tabs are positioned for a **Folder** control. By default, the folder tabs are positioned at the top of the folder.

The constant integer values for tab positions are listed in the following table.

Folder Class Constant	Value	Position
TabsPosition_Top	0	Top of folder
TabsPosition_Left	1	Left of folder
TabsPosition_Right	2	Right of folder
TabsPosition_Bottom	3	Bottom of folder

tabsRaggedRight

Type: Boolean

Availability: Read or write at any time

The **tabsRaggedRight** property specifies whether the tabs in a line of a **Folder** control are stretched to fill the available space for the tab line.

The default value for this property is **false**, indicating that the tabs are stretched to fill the available space in the tab line.

If you set the **tabsRaggedRight** property to **true**, the tabs are sized only to fit the icon and text of the sheet. If you set the **tabsFixedWidth** property to **true**, all tabs use the size of the largest required tab.

This property is ignored when the **tabsLines** property is set to **TabsLines_MultiLineEdged** (0).

Note When the lengths of the tab lines are different, setting this property to **true** can result in tabs that may appear rather ugly when used with other property values.

tabsStyle

Type: Integer

Availability: Read or write at any time

The **tabsStyle** property controls whether a **Folder** control is displayed with sculptured tabs or push buttons.

The values for this property are listed in the following table.

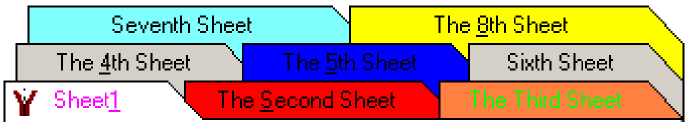
Folder Class Constant	Value	Description
TabsStyle_Tabs	0	Tabs stay in place on each line, but the currently selected tab line moves to the bottom of the line of tabs (the default). No highlight of an extra pixel is drawn around the current sheet.
TabsStyle_Buttons	1	Buttons push down or come up in place. As the buttons are drawn by using common facilities, they look like standard buttons. When the **tabsStyle** property is set to **TabsStyle_Buttons** for a **Folder** control, tabs are drawn using the **Button** control skin if the application has defined a button skin.
TabsStyle_RightSloped	2	Draws tabs with a sloped right-hand border. Tabs drawn with this style differ from the appearance of the **TabsStyle_Tabs** style in the following ways. ■ The tabs remain in the same position, as they do with the **TabsStyle_Buttons** style ■ The current tab is drawn by using the **Color_3DHighlight** color of the **Window** class ■ Moving the mouse over the tab of a sheet that is not the current sheet causes that tab to be drawn by using the **Color_InfoBk** color (bubble help background) ■ The font of the current tab is not drawn bold ■ When the **tabsLines** property is set to **TabsLines_MultiLineEdged** (0), the tab rows are offset from the left (as they are for the **TabsStyle_Tabs** style) but there is no multiple-edged drawing of the sheets themselves

Tab styles have different effects on the appearance of the folder control, as follows.

- Tabs are drawn with slightly rounded corners and have a three-dimensional border around each tab.
- Buttons are rectangular, drawn with three-dimensional effects, and there is no border drawn around the folder.

  The **tabsLines** property setting of **TabsLines_MultiLineEdged** (0) is therefore treated as multiple lines with no simulated sheet edges.
- Tabs are drawn using a rectangle with a sloped top-right border, shown in the following image.

The following example shows folder tabs with right-sloping borders.



tabStop

Type: Boolean

Availability: Read or write at any time

The **tabStop** property of the **Control** class specifies whether a user can use the Tab key to set the focus to a control.

The settings of the **tabStop** property are listed in the following table.

Value	Description
true	Designates the control as a tab stop (the default)
false	Bypasses the control when the user is tabbing, although the control still holds its place in the actual tab order, as determined by the **tabIndex** property

Use the **tabStop** property to add or remove a control from the tab order on a form.

When a folder control gets the focus, the current tab has a focus rectangle drawn around it, and the arrow keys can then be used to move between the sheets of the folder.

tabWidth

Type: Integer

Availability: Read or write at any time

The **tabWidth** property of the **JadeTextEdit** control contains the size of a tab in the text editor.

The tab size calculated as a multiple of the size of a space character in the default style (represented by the **STYLE_DEFAULT** class constant) for the control.

The default tab width is eight (**8**) characters, and you can specify a value in the range zero (**0**) through **100**.

tag

Type: String

Availability: Read or write at any time

The **tag** property of the **Window** class contains a data value to be stored with the form or control. Unlike other properties, the value of the **tag** property is not used by JADE, but you can use this property to identify objects.

The length of the tag string can be in the range **0** through **32,767** bytes. By default, the **tag** property is set to an empty string (**""**).

Use this property to assign an identification string to an object without affecting any of its other property settings or causing side effects.

Use the **tag** property when you need to check the identity of a control or MDI form that is passed as a variable to a method.

targetDevice

Type: Integer

Availability: Read or write at any time

The **targetDevice** property of the **JadeRichText** control contains the device used for "what you see is what you get" (WYSIWYG) printing. The values for this property are listed in the following table.

JadeRichText Class Constant	Integer Value
TargetDevice_Printer	1
TargetDevice_Screen	0

When this property is set **TargetDevice_Printer** (1), the rich text format control layout changes to match how the contents would look if they were printed on the current default printer. In this mode, the **lineWidth** property has no meaning, as the horizontal width of the text is determined by the printer.

text

Type: String

Availability: Read or write at any time for text box, edit mask, and rich text controls, read or write at run time only for tables, list boxes, and combo boxes with the **style** property set to **0** (dropdown) or **1** (simple), read or write at run time only for text edit controls that have the **readOnly** property set to **true**, read-only at run time for all other combo boxes, read or write at run time only for text editor controls

The **text** property contains the text in the edit area of **ComboBox** controls that have the **style** property set to **Style_DropDown** (0) or to **Style_Simple** (1), and the **TextBox** control. For other styles of the **ComboBox** control, this property returns the text of the selected item in the combo box list. The property is read-only in this context.

The **text** property of the **JadeRichText** control contains the text in the control in plain text format.

In the **JadeEditMask** and **TextBox** classes, this property can be translated when the value of the **Schema** class **formsManagement** property is **FormsMngmt_Single_Multi** (2).

For a **ListBox** control, this property contains the text of the selected item in the list box. This property is equivalent to accessing the **itemText** property with an index of **listIndex**. Changing the text of an item in the list box can change its position because of sorting. When the text of an item is altered from the **text** property, the **newIndex** method can be used to obtain its new index value.

For the **Table** control, the property accesses the current cell on the current sheet. The current cell is defined by the **row** and **column** properties. The code fragment in the following example uses concatenation with the **Tab** character to store text in cells to the right of the specified cell.

```
// Set up the column headings
table.row := 1;
table.column := 1;
table.text := "Name" & Tab & "Address" & Tab & "Phone";
```

For a **TextBox** or a **ComboBox** control that has the **style** property set to **Style_DropDown** (**0**) or to **Style_Simple** (1), this property returns the text contained in the edit area of the control.

A numeric text box expects the numeric string to be formatted according to the locale under which the user application is running. Setting the **TextBox** control **text** property validates the numeric value on the basis of that locale definition format. Retrieving the text returns the same string (that is, no locale conversion is performed).

Tip To automatically handle locale formatting in numeric text boxes, use the *getTextAs…* methods, **getTextAsCurrencyDecimal**, **getTextAsCurrencyReal**, **getTextAsDecimal**, **getTextAsInteger**, **getTextAsInteger64**, **getTextAsLongDate**, **getTextAsReal**, **getTextAsShortDate**, and **getTextAsTime**, and their corresponding *setTextFrom…* methods.

For a **ListBox** control or a combo box control with other settings of the **style** property, you can use the **text** property to obtain the text of the currently selected item. For a list box, if no entry is selected (**listIndex = -1**), a null string (**""**) is returned.

Note For a text box control, if the **dataType** property is to be set to a numeric type, setting the text is rejected if that text does not conform to the rules defined by the current **dataType**, **decimals**, and **maxLength** properties.

The **text** property of the **JadeEditMask** class contains the concatenated text of the text box fields of the control, including any literal text. When setting the text value of the control, that text must be valid according to the **mask** property rules. Any prompt characters contained in the text are not treated as prompt characters.

Note Any spaces in character positions other than edit mask types **C**, **c**, or literal positions are treated as prompt characters. For example, **' 2/ 3/2001'** for a mask of a **dd/MM/yyy** field is treated as **_2/_3/2001**.

Accessing the **text** property value returns a value consisting of the concatenated text of each of the text box fields, with each prompt character replaced by a space. For example, the text **21/10/2001** returns **'21/10/2001'** and the text **21/__/__** returns **'21/ / '**. Setting the text of the control to the returned value is always accepted. Setting the text to **null** (**""**) clears all characters that can be entered, leaving only any literals in the text and the prompt characters.

As dates are in locale order and abbreviated months are in locale format, you must convert them by using locale-aware routines. (For details, see "Date Type", in Chapter 1 of the *JADE Encyclopaedia of Primitive Types*, and "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.) Similarly, locale-equivalent characters are expected and returned for decimal places, thousand separators, negative signs, currency symbols, date delimiters, and time delimiters. Conversion must therefore use locale-aware routines.

Setting the text value with text that has a length less than the expected field length pads that text with an empty version of the expected text field. For example, setting a date field of **dd/MM/yyyy** to **'21'** results in the displayed text field of **21/__/____**.

The **text** property of the **JadeTextEdit** control can be set when the **readOnly** property of the **JadeTextEdit** control is set to **true**. When the **text** property is set, the undo buffer is cleared and the **modified** property is set to **false**. No end-of-line conversion is performed when the text is set or retrieved. You can call the **convertEndOfLines** method to force all line endings to a required value.

The following example shows the use of the **text** property.

```
userNotify(eventType: Integer;
           theObject: Object;
           eventTag:  Integer;
           userInfo:  Any) updating;
begin
    // The userNotify method is executed when a notification that was
    // registered for a user event is received.  The notification is
    // identified using the eventType parameter and an appropriate
```

```
        // message is displayed in a text box.
        if eventType = 16 then
            textBox2.text := "User Class Notification Received";
            textBox7.text := userInfo.String;
        elseif eventType = 17 then
            textBox4.text := "User Notification Received";
            textBox8.text := userInfo.String;
        endif;
    end;
```

textOffset

Type: Integer

Availability: Read or write at any time

The **textOffset** property of the **Text Box** class contains the pixel offset of the left and right margins for text displayed in a text box, which can improve the appearance of the displayed text.

The default value is zero (**0**). An exception is raised if the value is set to a negative value.

When the value is greater than zero, the text in the text box is displayed the specified number of pixels from the left and right of the text box client area, to create a left and right margin where text is not displayed.

Note The specified value is ignored if the effective text area is less than 5 pixels wide (that is, the text is too small for the margin to be applied).

Applies to Version: 2016.0.01 and higher

textRTF

Type: String

Availability: Read or write at any time

The **textRTF** property of the **JadeRichText** control contains the text in the control in rich text format.

textUser

Type: String

Availability: Read or write at run time only

The **textUser** property of the **JadeEditMask** control contains the concatenated user text of the text box fields of the control, excluding any literal text.

When setting the text value of the control, that text must be valid according to the **mask** property rules.

Accessing the **textUser** property value returns the concatenated text of each text box field with all literal characters removed. This text is stripped back to the last non-prompt character, with each prior prompt character replaced by a space. For example:

- **21/10/2001** returns **'21102001'**
- **21/__/__** returns **'21'**
- **21/__/3_** returns **'21 3'**

For a right-aligned field, accessing the **textUser** property returns the text right-aligned with blanks on the front of the string for any un-entered data. For example, the returned string for an edit mask of **'#####9'** with a value of **_____9** is **' 9'**.

Tip To automatically handle locale formatting, use the *getTextAs…* methods **getTextAsDate**, **getTextAsDecimal**, **getTextAsInteger**, **getTextAsInteger64**, **getTextAsReal** and **getTextAsTime** methods and their corresponding *setTextFrom…* methods.

A **JadeEditMask** control expects the data string to be formatted according to the locale that the control is using (for details, see the **languageId** property). Setting the **text** or **textUser** property validates the text on the basis of that locale definition format for numbers, dates, times, and so on. Retrieving the **text** or **textUser** property strings returns the same string (that is, no locale conversion is performed).

Notes Setting the **textUser** property of a control to the returned value must be valid according to the **mask** property rules.

Any spaces in character positions other than edit types **C**, **c**, or literal positions are treated as prompt characters. For example, **2 32001** in a **dd/MM/yyy** field is treated as **'_2/_3/2001'**.

As dates are in locale order and abbreviated months are in locale format, you must convert them by using locale-aware routines. (For details, see "Date Type", in Chapter 1 of the *JADE Encyclopaedia of Primitive Types*, and "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.) Similarly, locale-equivalent characters are expected and returned for decimal places, thousand separators, negative signs, currency symbols, date delimiters, and time delimiters. Conversion must therefore use locale-aware routines.

thinClientUpdateInterval

Type: Integer

The **thinClientUpdateInterval** property of the **ProgressBar** control specifies (in milliseconds) how often the progress bar is redrawn when the percentage changes when running the application in thin client mode. The default value of **1000** milliseconds (1 second) specifies that the control is updated only when the percentage changes and at least one second has elapsed since it was last updated or the 100 percent mark has been reached.

If you set the value of the **thinClientUpdateInterval** property to zero (**0**) when the application is running in thin client mode, the progress bar is updated for every required percentage change specified by the **partsDone** property.

Tip This automatic thin client optimization prevents an unnecessary number of messages being sent over the TCP connection when the progress updates at a fast rate.

timeFormat

Type: String

Availability: Read or write at any time

The **timeFormat** property of the **MultiMedia** class contains the time format of the device. The time format defines the units for obtaining and controlling the positioning of the media. This time format is then used by the **position** property, the **stepRelative** and **playFromTo** methods, and applies to the position sent to the **notifyPosition** event.

If the **useDotNetVersion** property is set to **true**, the **timeFormat** property is not available and it has a fixed value of null (**""**).

The time formats that are available for each device and the default values differ. For example, the default value for **digitalvideo** is **frames**, and for **waveaudio** it is **milliseconds**. For the list of valid formats for each device, see the appropriate device documentation.

The following example shows the use of the **timeFormat** property.

```
optionMillisecond_change(optionbutton: OptionButton input) updating;
begin
    if optionbutton.value then
        multimedia.timeFormat := "milliseconds";
    endif;
end;
```

timerPeriod

Type: Integer

Availability: Read or write at any time

The **timerPeriod** property of the **MultiMedia** class contains the timer period used for the **notifyMode**, **notifyMedia**, and **notifyPosition** events.

The timer period, expressed in milliseconds, controls the time that elapses before the control is notified of a change in operating mode, media, or position. The default value is **500**. The minimum value is **30**.

top

Type: Real

Availability: Read or write at any time

The **top** property of the **Window** class contains the distance between the internal top edge of an object and the top edge of the client area of the container (the non-border area). When the parent is a **Form** class or a **BaseControl** or **Picture** control, the top position is also offset by the amount that the parent is scrolled.

The **top** property for a form is always expressed in pixels.

Note If the value of the **top** property plus the value of the **height** property is greater than **32,767** pixels, the resulting window extents may be unpredictable.

For controls, the **top** property is expressed in units controlled by the **scaleMode** property of the parent of the control. The default value of the **scaleMode** property is pixels. (See also the **parentAspect** property.)

The value for the **top** property changes as the object is moved by the user or by logic.

The code fragments in the following examples show the use of the **top** property.

```
if scrollBar = ScrollBars_Vertical then
    toolbar.top  := scrollVertPos;
else
    toolbar.left := scrollHorzPos;
endif;

startName.top := theTable.top + theTable.height + 10;
```

When multiple monitors are running on a workstation and a form is saved in the JADE Painter, the values of the **left** and **top** properties are converted to be relative to the top and left of the primary monitor.

topIndex

Type: Integer

Availability: Not available at design time, read or write at run time only

The **topIndex** property contains the item in a **ListBox** control that is the first item displayed in the list. The default value is **1**, or the first item in the list.

Use this property to scroll through a list box without selecting an item. The item appears at the topmost position if there are enough items below it to fill the visible portion of the list. The value of the **topIndex** property can be modified to ensure that a full page of entries is displayed. If the setting would result in empty lines being displayed at the end of the list and there are prior entries, the value of the **topIndex** property is decreased until a full set of entries is visible.

Setting the **topIndex** property causes the list box to be filled with entries up to that point when using the **listCollection** method and the **topIndex** property value is greater than the entries that are obtained so far.

The following example shows the use of the **topIndex** property.

```
textBoxLeftStart_change(textbox: TextBox input) updating;
vars
    count : Integer;
begin
    count := listBoxLeft.findString(1, textBoxLeftStart.text);
    if count <> -1 then
        listBoxLeft.topIndex := count;
    endif;
end;
```

topLevelMenuItems

Type: MenuItemArray

Availability: Read at run time only

The **topLevelMenuItems** property of the **Form** class contains a reference to an array of all of the top-level menu items on the form (that is, those that appear on the form menu bar). The collection is ordered according to the defined menu item list.

Applies to Version: 2016.0.01 and higher

topRow

Type: Integer

Availability: Read or write at run time

The **topRow** property contains the row that is displayed at the top edge of the non-fixed area of the current sheet of a **Table** control. This value may be decreased automatically by the control if lower values can still display the remainder of the rows.

The **topRow** property is set to the first non-fixed row in the following situations.

- If the display does not require a scroll bar
- When using attached collections

Changing the **row** or **column** property does not change the rows or columns that are displayed, as these are changed only by the **leftColumn** and **topRow** properties.

The following example shows the use of the **topRow** property.

```
convertPositionToRow(yPos: Real): Integer updating;
// Returns the table row whose top and height positions cover the
// y coordinate that is passed
vars
    iy          : Integer;
    totalHeight : Integer;
begin
    iy := theTable.topRow;
    totalHeight := theTable.rowHeight[1];    // the fixed row
    while iy <= theTable.rows do
        totalHeight := totalHeight + theTable.rowHeight[iy];
        if yPos <= totalHeight.Real then
            return(iy);
        endif;
        iy := iy + 1;
    endwhile;
    return(0);
end;
```

topSheet

Type: Integer (**Table** class), Sheet (**Folder** class)

Availability: Read or write at run time only

The **topSheet** property contains the sheet of a **Folder** control or **Table** control that is currently visible (the topmost).

For a **Folder** control:

- The **topSheet** property returns the control object of the sheet that is currently on display.
- Setting the **topSheet** property to a sheet of the folder causes that sheet to become the currently displayed sheet. This sheet must be visible.
- If the value of the **topSheet** property is changed, focus is moved to the folder only when the control with focus is a child of the folder. If the control with focus is not a child of the folder, focus is not moved.
- An alternative method is to use the **showMySheet** and **isMySheetVisible** methods.
- Setting the focus to a control causes the sheet on which the control resides to be brought to the top.

For a **Table** control:

- The **topSheet** property returns the index (**1**-relative) of the currently displayed sheet.
- Setting the **topSheet** property to an integer index value causes the sheet with that index to become the currently displayed sheet.

The following example shows the use of the **topSheet** property.

```
buttonOK_click(btn: Button input) updating;
begin
    if folder1.topSheet.name = 'customerEdit' then
```

```
            beginTransaction;
            myCustomer.update(textBoxName.text, textBoxAddress.text);
            commitTransaction;
        elseif folder1.topSheet.name = 'employeeEdit' then
            beginTransaction;
            myEmployee.update(textBoxName.text, textBoxTaxNumber.text);
            commitTransaction;
        endif;
    end;
```

transparent

Type: Boolean

Availability: Read or write at any time

The **transparent** property applies to the **CheckBox**, **Frame**, **GroupBox**, **Label**, **OptionButton**, **Picture**, and **WebHTML** controls.

Setting the **transparent** property to **true** causes the control to be placed above all other sibling controls and the controls or form underneath to be visible. The background area of the control is not painted. The control still responds to all mouse actions over the transparent areas.

The uses of this property are:

- With the **backBrush** property of a form; for example, painting just the text of a label over the **backBrush** bitmap without erasing the background area of the label.
- Placing hotspots on a picture over certain positions that respond to mouse actions (including changing the mouse cursor).
- Placing a control over the top of all other controls for use with a painter-type environment. The transparent control takes all the mouse and keyboard actions, while allowing the controls being manipulated to be visible without them receiving the focus.

Notes This property requires more overhead, as the transparent area receives multiple paints, and changing the transparent control requires additional access to determine which windows are affected.

When controls are removed from a transparent parent in the JADE Painter (either by being dragged off the parent or deleted), the Painter must refresh the transparent control, as it is not refreshed by Windows.

The **transparent** property for a Web picture on a **Label** control is set to **true** when the hotspot is created.

This property applies only to HTML on Web pages accessed using Internet Explorer 4.0 (or higher).

transparentColor

Type: Integer (color)

Availability: Read or write at run time only (**Picture**) or in a subclass at design time (**JadeMask**)

The **transparentColor** property contains the color that is transparent for an image in a **Picture** control; that is, the image is displayed so that the window image of the parent shows through the image anywhere that this color appears in the image.

The **transparentColor** property takes effect only when the **transparentColor** property is not set to black (**0**) and the **rotation** property is set to zero (**0**). The **transparentColor** property default value of black (**0**) indicates that this property is ignored.

The code fragment in the following example shows the use of the **transparentColor** property.

```
transparentColor := 192 + 192*256 + 192*256*256;
```

useDotNetVersion

Type: Boolean

Availability: Read or write at any time

The **useDotNetVersion** property of the **MultiMedia** class specifies whether the control uses .NET, providing access to new style media files such as an MPEG Layer-4 Audio (MP4) file. When the value of this property is:

- **false** (the default value), the control operates the same as it did in earlier versions of JADE; that is, it uses the older-style MCI control
- **true**, the control creates a Microsoft WPF MediaElement control that is used to play the media files

When using the .NET control style (that is, this property is set to **true**), the JADE **MultiMedia** control mostly operates in the same way as the older style, with the following exceptions.

- If the play bar of the control is shown, it is drawn using WPF entities and it has a different appearance.
- If the value of the **showMenu** property is **true**, the menu that is displayed when the menu button is clicked shows the **View**, **Volume**, and **Speed** menu commands but not the **Copy**, **Configure**, and **Command** menu items. In addition, the menu provides the **Close** command, which closes the medium that is being shown.
- The **mediaName** property can be a URL (for example, http://hostName/images/introduction.mp4).
- The .NET style *may* not handle older style media files.
- The following **MultiMedia** class properties and methods are not available, and generate exception 1068 (*Feature not available in this release*).
  - eject
  - newFile
  - playFromTo
  - playReverse
  - record
  - save
  - sendString
  - showRecord
  - timeFormat
- The methods and property listed in the following table return fixed values.

Method or Property	Fixed Value
canEject	false
canPlay	true
canRecord	false
canSave	false
timeFormat	null
usesFiles	true

- The **openDialog** method includes MP4-type files in the files that are displayed.
- Media attribute values position (obtained by calling the **getEndPosition**or **getStartPosition** method) and length (obtained by calling the **getLength** method) are not available until the medium has been opened and the **notifyMedia** or **notifyMode** event has been received.

Applies to Version: 2016.0.01 and higher

usePresentationClient

Type: Boolean

Availability: Read or write at run time only

The **usePresentationClient** property of the **ActiveXControl** class, which is used in conjunction with the **makeAutomationObject** method, specifies whether the ActiveX control is executed on the presentation client or application server.

As this property is set to **true** by default, set it to **false** if you want your ActiveX control to run on the application server when an automation object is created by using the **makeAutomationObject** method.

userInputEnabled

Type: Boolean

Availability: Read or write at run time only

The **userInputEnabled** property of the **Table** class provides you with the ability to enable or disable all **inputType** and **cellControl** property actions that are currently set. The **userInputEnabled** property enables you to establish the input facilities for the table and disable then until some other precondition on the form is completed.

By default, the **userInputEnabled** parameter is set to **true**, and when the user moves to a cell that has an **inputType** or **cellControl** property in effect, the appropriate control is positioned and then made available for user interaction.

When the **userInputEnabled** property is set to **false**, the **inputType** or the **cellControl** property value is ignored for all cells.

As the **readOnly** property allows **inputType** or **cellControl** property actions when the **readOnly** property is set to **false**, it does not cover all required user situations that can be handled by this **userInputEnabled** property.

userObject

Type: Object

Availability: Read or write at run time only

The **userObject** property contains a reference to an object that you can associate with any object of the **Window** class (a form or control). This is a runtime-only property, which is not used by any JADE process. It is defined only for your convenience.

The default value for the **userObject** property is **null**.

userScript

Type: String

Availability: Read or write at run time only

The **userScript** property of the **Window** class contains a string that is used to set up scripts to be included as part of the HTML generation. Set this property dynamically at run time, by specifying the string in the code of your Web application.

The script is included in the HTML generate exactly as it is, and no validation is performed. It is therefore your responsibility to validate that the script will run on all Web browser platforms in which the application is expected to be deployed.

Scripts added to the **Form** object in your JADE code are generated before the HTML **<BODY>** tag. Scripts added to **Control** objects are generated before the control tag. See the **Window** class **addWebEventMapping** method for details about writing a function and including it as part of your user script to be invoked when a specified event occurs.

useTabs

Type: Boolean

Availability: Read or write at any time

The **useTabs** property of the **JadeTextEdit** control specifies whether tabs are used in the text editor to indent the lines to the next indent position.

The default value is **true**; that is, indentation is created by inserting a tab character to the next indent position. Set this property to **false** if you want indentation created by using space characters.

To replace all indentation in the text editor control with spaces only or with a combination of tabs and spaces, call the **convertIndentWhitespace** method.

value

Type: Integer (scroll bars), Boolean (check box, button, JADE mask, and option button)

Availability: Read or write at any time

The **value** property determines the state of a **CheckBox** control as checked or unchecked. The default value is zero (**0**), or **false**.

For **Button** and **JadeMask** controls, the **value** property specifies whether the state of the button or mask control is up (**false**) or down (**true**). Setting the **value** property is ignored for a normal push button or mask control with a **style** property of **Style_Normal** (**0**). For a two-state or an automatic two-state button or mask control, setting this property pushes or raises the button or mask control. For an automatic two-state button or mask control, setting the **value** property to **true** also causes other automatic two-state buttons or mask controls with the same parent to come up. For automatic two-state buttons or mask controls, setting the **value** property to **false** has no impact on other buttons or mask controls in the same group.

For **OptionButton** controls, setting the **value** property to **true** causes other option buttons in the same group to be set to **false**. Setting the **value** property to **false** has no impact on other option buttons in the same group.

For horizontal and vertical **ScrollBar** controls, the **value** property determines the current position of the scroll bar. The value that is returned is always between the values for the **max** and **min** properties.

The settings for the **value** property are listed in the following table.

Control	Description
CheckBox	**false** is unchecked (default), **true** is checked
Button and JadeMask	**false** indicates the button is up, **true** indicates the button is down
ScrollBar	Set values between the **min** and **max** properties, to position the scroll box
OptionButton	**true** indicates the button is selected, **false** (the default) indicates the button is not selected

The following example shows the use of the **value** property.

```
tranState_click(checkbox: CheckBox input) updating;
begin
    if tranState.value then
        if not process.isInTransactionState then
            beginTransaction;
            sl1.caption := $S_In_Transaction_State;
        endif;
    else
        if process.isInTransactionState then
            commitTransaction;
            sl1.caption := $S_Not_In_Transaction_State;
        endif;
    endif;
end;
```

verticalSpace

Type: Integer

The **verticalSpace** property of the **WebJavaApplet** class contains the number of pixels above and below the applet on your Web page.

viewEndOfLine

Type: Boolean

Availability: Read or write at any time

The **viewEndOfLine** property of the JadeTextEdit control specifies whether end-of-line characters are displayed in the text editor.

This property is set to **false** by default; that is, end-of-line characters are not displayed.

viewLineNumbers

Type: Boolean

Availability: Read or write at any time

The **viewLineNumbers** property of the JadeTextEdit control specifies whether line numbers and the line number margin (margin 0)are displayed.

This property is set to **false** by default; that is, line numbers are not displayed.

viewWhitespace

Type: Integer

Availability: Read or write at any time

The **viewWhitespace** property of the JadeTextEdit control specifies how space and tab characters are visibly displayed in the text editor.

The **viewWhitespace** property can be set to one of the **JadeTextEdit** class constants listed in the following table.

Class Constant	Value	White space…
SCWS_INVISIBLE	0	Is displayed as an empty background color (the default value)
SCWS_INVISIBLEAFTERINDENT	2	Used for indentation as an empty background color but after the first visible character, it is shown as dots and arrows (for spaces and tabs, respectively)
SCWS_VISIBLEALWAYS	1	Space and tab characters are always drawn as centered dots and arrows, respectively

visible

Type: Boolean

Availability: Read or write at any time

The **visible** property of the Window class specifies whether an object is visible or hidden. When the **visible** property is set to **true** (the default), the object is visible. Set this property to **false** to hide the object.

To hide a control at startup, set the **visible** property to **false** in the JADE development environment. Setting this property in logic enables you to hide and later redisplay a control at run time in response to a specific event.

The **visible** property, when set in the JADE development environment, has no effect for a form. The **show** or **showModal** method causes the form to become visible. Setting the **visible** property to **true** causes the **load** event for the form to be executed if it has not already been executed.

As sibling controls that have the **alignContainer** property set do not occupy the same area on their parent, use the **visible** property to control the window that is currently displayed in this situation.

When a window is invisible, it does not receive any mouse, keyboard, focus, activation, or drag and drop events. The window still receives notification events and forms still receive **load**, **queryUnload**, and **unload** events.

When the **visible** property for a **Button**, **CheckBox**, **OptionButton**, **Picture**, **Label**, **TextBox**, Web label, or Web picture control defined for a Web page is set to **false**, the input type for the control is **hidden**. For other controls that are supported on Web pages, the control is not included in the HTML generation when this property is set to **false**.

The code fragment in the following example shows the use of the **visible** property.

```
// show hourglass etc when loading data (unless during initial form load)
if self.visible then               // visible unless loading
    app.mousePointer    := MousePointer_HourGlass;
    statusLine1.caption := 'Reading...';
endif;
```

Note The **visible** property cannot be changed for a form created with the **createPrintForm** method of the **GUIClass** class.

volume

Type: Integer

Availability: Read or write at any time

The **volume** property of the **MultiMedia** class contains the playback volume of the device. As not all devices support this facility, its use raises an exception when it is not supported.

The default value of volume is **1000**.

A volume of **500** plays at half volume, a volume of **2000** plays at double volume, and so on.

wantReturn

Type: Boolean

Availability: Read or write at any time

The **wantReturn** property of the **TextBox** and **JadeRichText** class specifies whether carriage returns are passed to the text box or rich text control.

When this property is set to the default value of **false**, entering a carriage return while the text box or rich text control has focus and the form has a default button causes the default button to get focus and be clicked. If the form does not have a default button, the carriage return is passed to the text box or rich text control.

When the **wantReturn** property is set to **true** and the text box or rich text control has focus, a carriage return is always passed to the text box or rich text control and any default button is unaffected.

Note This property is ignored in web-enabled forms.

Applies to Version: 2016.0.01 and higher

webBrowserAutoRefreshInterval

Type: Integer

The **webBrowserAutoRefreshInterval** property of the **Form** class specifies the number of seconds after which the Web page is automatically refreshed. The default value of zero (**0**) indicates that the Web page does not refresh automatically.

To generate the required **<META>** tag for the automatic refreshing of the Web page, set this property to the appropriate non-zero value. For details about handling Web pages after the specified interval has been reached, see the **Form** class **webBrowserAutoRefreshURL** property.

webBrowserAutoRefreshURL

Type: String

When the value of the **webBrowserAutoRefreshInterval** property of the **Form** class is not zero (**0**), use the **Form** class **webBrowserAutoRefreshURL** property to specify the URL to invoke when the automatic refresh interval is reached.

The default **webBrowserAutoRefreshURL** property value of null (**""**) indicates that the JADE application is returned to when the specified number of seconds is reached. If this property contains a null value and a default button has been set up for the Web page, the **click** method for this button is then called.

Note If there is no default button, the same page is displayed continuously unless your JADE code handles the incoming request.

webBrowserDisableBackButton

Type: Boolean

The **webBrowserDisableBackButton** property of the **Form** class specifies whether the previous Web page is displayed when the user clicks the Web browser **Back** button. By default, this property is set to **false**.

Set the **webBrowserDisableBackButton** property to **true** if you want the currently displayed Web page refreshed instead of displaying the previous page when the user clicks the Web browser **Back** button.

Note As it is not possible to disable the **Back** button or the menu item from scripting, use this property if you want to prevent the user from going to a previous Web page.

webEncodingType

Type: Character

The **webEncodingType** property of the **Form** class specifies the content type used to submit the Web form to the JADE application.

You can set the content type to one of the values listed in the following table.

Character Value	Description
None	Application / *x-www-form-urlencoded* (the default value)
M	Multipart / *form-data* (when the returned page includes submitted files)

Any value other than **M** (that is, multipart) is treated as the application type default. You can set the encoding type only in your JADE code. This cannot be set by using the JADE Painter.

webFileName

Type: String

Availability: Read or write at any time

The **webFileName** property of the **Picture** class has two purposes. It enables you to specify a file name (for example, "**image.jpg"** or "**mypic.png"**) for:

- An image that is created by the Web framework (rather than the automatically generated file name)
- A file that already exists

Note A firewall would prevent an existing file from being used, so to use an existing file requires that the **firewall** parameter in the [WebOptions] section of the JADE initialization file is set to **false**.

Using an existing file for static images can greatly improve performance.

For the **Form** class, this property contains the name of the background image that is to be displayed on the Web page.

For the **Button** class, this property contains the name of the image that is to be displayed on a button on the Web page.

webInputType

Type: Character

Availability: Read or write at any time

The **webInputType** property of a **TextBox** control contains the type of input that is accepted by a text box on a Web page.

You can assign the **webInputType** property to each text box on a Web page.

The types of input that can be accepted are listed in the following table.

TextBox Class Constant	Value	Description
Web_InputType_File	'F'	Uploading a file from a Web session
Web_InputType_Hidden	'H'	Specifies whether the text box is displayed on a Web page (see the **enabled** property)
Web_InputType_Password	'P'	Specifies whether characters are displayed in a text box (see the **passwordField** property)
Web_InputType_Text	'T'	Single line text box (the default)
Web_InputType_TextArea	'A'	Multiple line text box

If your JADE application accepts file input in text boxes on a Web page (by using the **webInputType** property of a **TextBox** control with the **Web_InputType_File** setting to upload a file from a Web session), you can use the **fileTransferDirectory** parameter in the [Jadehttp Files] section of the **jadehttp.ini** file to specify the directory to which the file is written. This parameter controls the directory in which any files transferred using the HTML **InputType=file** option are placed. By default, any transferred files are placed in the same directory as the **jadehttp** library.

The format of the **text** property value of the text box control is:

```
<source-file-name>;<destination-file-path><destination-file-name>
```

The *source-file-name* value is the name (excluding the path) of the originating file on the client workstation from which the file was loaded (that is, the workstation that is running the Web browser).

A semicolon character (**;**) separates this and the *destination-file-path* and *destination-file-name* values, which are the full path to which the file is written (uploaded) and the name of that file; for example:

```
UsefulStuff.doc;d:\jade\bin\txf188.tmp
```

In this example, accessing the **txf188.tmp** file in the specified directory opens a document file that contains the information in the **UsefulStuff.doc** file uploaded via the Web browser.

To provide increased security for applications running in HTML thin client mode if you use the **webInputType** property of a **TextBox** control with the **Web_InputType_File** setting to upload a file from a Web session, you must process each text file that is transferred in the event that resulted in the file upload (for example, in the **click** method of a **Completed** button).

Caution To prevent malicious use of files uploaded to Web-enabled applications, the files are removed as soon as the event that resulted in their upload has completed. You should therefore process the file immediately or move it into a directory that is not available from the Web if you require that file for future processing.

width

Type: Real

Availability: Read or write at any time

The **width** property of the **Window** class contains the dimensions of an object.

Width measurements are calculated by using the following.

- For a **Form**, the external width of the form, including the borders and title bar
- For a **Control**, the external width of the control

For a form, the **width** property is always in pixels.

Note If the value of the **width** property plus the value of the **left** property is greater than 32,767 pixels, the resulting window extents may be unpredictable.

For a control, the **scaleMode** property units of the parent control determine the width. The **scaleMode** property defaults to pixels. For a form or control, the value for the **width** property changes as the object is sized by the user or by logic.

The maximum limit for all objects is system-dependent.

Windows limits forms and controls to a maximum width of 32,767 pixels. Setting a value larger than the maximum results in a value of 32,767 pixels being used.

The code fragments in the following examples show the use of the **width** property.

```
tblPortfolio.columnWidth[1] := tblPortfolio.width.Integer div 3 - 12;

height := width / 1.5;             // Make control a rectangle

btn.width := clientWidth / ButtonWidthRatio;
```

See also the **parentAspect** property.

windowState

Type: Integer

Availability: Read or write at any time

The **windowState** property of the **Form** class contains the visual state of a form window at run time.

The settings of the **windowState** property are listed in the following table.

Form Class Constant	Value	Description
WindowState_Normal	0	Normal (the default)
WindowState_Minimized	1	Shrunk to an icon
WindowState_Maximized	2	Enlarged to maximum size

Minimizing a form causes a **resize** event. The size of the form and its controls reflects the minimized state.

The following example shows the use of the **windowState** property.

```
resize() updating;
begin
    if windowState <> WindowState_Minimized then // restore after minimize
        caption := "CD Player";
    endif;
end;
```

wordWrap

Type: Boolean

Availability: Read or write at any time (labels, frames, or status lines) or at run time only (tables)

The **wordWrap** property specifies whether text displayed in a caption for a **Label**, **Frame**, or **StatusLine** control advances to the next line of the control when the current line is filled.

The wrapping is performed based on complete words.

The settings of the **wordWrap** property are listed in the following table.

Setting	Description
true	The text wraps when the text overflows a line. The horizontal size does not change.
false	The text is output on a single line and may overflow the control.

A **Label** control can be automatically sized to fit the text by setting the **autoSize** property to **true**. The setting of the **wordWrap** property then determines whether the resize is performed horizontally or vertically.

For **Table** controls, the **wordWrap** property specifies whether the text of a cell is displayed using word wrap when the width of the cell is less than the length of the text. Accessing this property on a table is affected by the current value of the **Table** class **accessMode** property. The **wordWrap** property can also be accessed by the **accessSheet**, **accessRow**, **accessColumn**, and **accessCell** methods.

The default value for any new **Table** control that is added is **false**. When the value of the **wordWrap** property is set to **false** for a table, the text of a cell is displayed as a single line unless the text contains carriage return characters, which start a new line. In addition, when the **inputType** property of a cell has a value of **InputType_TextBox**, that text box is displayed in the cell at data entry time with the **scrollHorizontal** property set to **true**, indicating that the text scrolls horizontally as required to access all of the text.

If the **wordWrap** property for a table is set to **true**, the text of a cell is displayed using word wrap so that if the text is wider than the cell, a new line is started when the text exceeds the cell width, breaking on word boundaries.

In addition, when the **inputType** property of a cell is set to **InputType_TextBox**, that text box is displayed in the cell at data entry time with the **scrollHorizontal** property set to **false**, indicating that the text also uses word wrap during entry. If word wrapping occurs, the height of the cell may need adjusting to fully display the text.

When the **Form** class **generateHTML** method is called to generate an HTML string or HTML is automatically generated for forms in a Web-enabled application, the HTML is generated without word wrapping when the **wordWrap** property is set to **false**. Set this property to **true** if you want an HTML string in a table cell generated with word wrapping.

When the **wordWrap** property for a **Table** is set to **true**, the effect of setting the **autoSize** property is as follows.

- If the column width is *not* set by logic or by the user, the height of the cell is affected if the value of the **widthPercent** property of the column is greater than zero (**0**) or the **autoSize** property is set to **AutoSize_Row** (**1**) and the row height has not been specifically set (otherwise a non-word wrap display is assumed).

- If the column width is set by logic or by the user, the height of the cell is affected if the **autoSize** property is set to **AutoSize_Row** (1), **AutoSize_Both** (3), or **AutoSize_BothColumnMinimum** (4) and the row height has not been specifically set.

For details about displaying an indication when there is insufficient room to show all text of a cell, see the **Table** class **partialTextIndication** property.

wrapIndent

Type: Integer

Availability: Read or write at any time

The **wrapIndent** property of the **JadeTextEdit** control contains the number of spaces by which continuation lines of wrapped lines are indented when the **wrapMode** property is set to **SC_WRAP_WORD**.

When continuation lines of wrapped lines are indented, the display of a visible edge marker on continuation lines is offset the corresponding number of characters to the right.

By default, continuation lines of wrapped lines are not indented.

The valid range is zero (**0**) through **90**.

wrapMode

Type: Integer

Availability: Read or write at any time

The **wrapMode** property of the **JadeTextEdit** control contains the way in which lines of text that exceed the text editor line length are wrapped to fit within the client area.

The **wrapMode** property can be set to one of the **JadeTextEdit** class constants listed in the following table.

Class Constant	Value	Description
SC_WRAP_NONE	0	Disables line wrapping (the default value)
SC_WRAP_WORD	1	Enables line wrapping

When line wrapping is enabled, lines wider than the client window width continue on the following lines. Lines are broken after space or tab characters or between characters of different text styles. However, if a word in one style is wider than the window, the break occurs after the last character that completely fits on the line. When wrap mode is enabled, the horizontal scroll bar is not displayed.

When wrapping is enabled, the Home and End keys move the caret to the start and end of a text line, respectively, as opposed to a display line.

New line characters are not inserted in the text at the wrapping point. Wrapping brings into view the right-hand side of long lines that are normally outside the display area.

Use the **edgeMode** property if you want an indication when a line reaches a specified length; for example, as a format convention that limits lines to 80 characters.

wrapVisualFlags

Type: Integer

Availability: Read or write at any time

The **wrapVisualFlags** property of the **JadeTextEdit** control contains the way in which visual flags are displayed to indicate that a line of text is wrapped when the **wrapMode** property is set to **SC_WRAP_WORD**.

When word wrapping is enabled, small arrows are used as visual flags at end of a continuation line of a wrapped line or at the beginning of the next continuation line. When visual flags are displayed, the flag at the beginning of the next continuation line is indented by one character.

The **wrapVisualFlags** property can be set to one of the **JadeTextEdit** class constants listed in the following table.

Class Constant	Value	Visual flags …
SC_WRAPVISUALFLAG_END	1	Are displayed at the end of a continuation line of a wrapped line
SC_WRAPVISUALFLAG_END_BY_TXT	3	At the end of continuation lines are drawn near to the text rather than near to the border of the text editor
SC_WRAPVISUALFLAG_NONE	0	Are not displayed in wrapped lines (the default value)

Class Constant	Value	Visual flags …
SC_WRAPVISUALFLAG_START	2	Are displayed at the start of a continuation line of a wrapped line and the continuation line is indented by one character to accommodate the visual flag
SC_WRAPVISUALFLAG_START_BY_TXT	4	At the start of continuation lines, are drawn near to the text rather than near to the border of the text editor

xaml

Type: String

Availability: Read or write at any time

The **xaml** property of the **JadeXamlControl** class contains the Windows Presentation Foundation (WPF) definition of the content of the control. The definition is written in the Extensible Application Markup Language (XAML).

The content is compiled and added as a child of the WPF DockPanel parent of the control. The content can be a single or composite WPF entity. If the control has an existing definition, setting the **xaml** property discards the existing definition and replaces it with the new definition.

The value of the **xaml** property must include an XML Name Space (XMLNS) definition that defines the name space necessary for the definition to be successfully compiled according to the WPF installation of the user.

Note Currently Windows does not support XAML that includes event definitions.

The following example shows how the **xaml** property is set at runtime.

```
xamlControl.xaml :=
    '<Canvas Name="SimpleExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
        <Button Canvas.Top="20" Canvas.Left="50" Content="Hello World" />
    </Canvas>';
```

zoom

Type: Integer

Availability: Read or write at any time

The **zoom** property of the **MultiMedia** class contains the zoom factor of the video image. As not all devices support this facility, its use raises an exception when it is not supported. The zoom default value is **100** percent.

A zoom factor of 50 percent halves the size of the control and a factor of 200 percent doubles the control size, and so on. Setting the **zoom** property to zero (**0**) when the value of the **autoSize** property is **false** stretches the media image to the current size of the client area of the control.

MultiMedia controls are sized as follows.

- When the value of the **autoSize** property is **true**, the size of the control is set to the size of the displayed image (multiplied by the value of the **zoom** property divided by 100) plus the size of any playbar and caption. If the device or file being played does not involve a playback image, the control is not visible. If the value of the **zoom** property is zero (**0**), 100 is used.

- When the value of the **autoSize** property is **false** and the value of the **zoom** property is not zero, the control size is not affected by the size of the displayed image. Only as much of the displayed image that fits in the client area of the control is displayed.
- When the value of the **autoSize** property is **false** and the value of the **zoom** property is zero (**0**), the displayed image is stretched to fit the client area of the control.

For the **JadeRichText** class, the **zoom** property contains the factor by which the contents of the control are zoomed. This is a scale factor in the range 2 percent through 6,400 percent. A value of zero (**0**) indicates no zooming. The zoom default value is 100 percent; that is, this property is set to zero (**0**). A zoom factor of 50 percent halves the size of the control and a factor of 200 percent doubles the control size, and so on. For a rich text control, the **zoom** property requires a scalable font; for example, the Microsoft Sans Serif font is not a True Type font and therefore it cannot be scaled.

The **zoom** property of the **JadeTextEdit** control, which is read and write at run time only, contains the current zoom factor.

The zoom factor, which defaults to zero (**0**), enables you to increase or decrease the font size of text in the text editor in steps of one point. The point size is added to or removed from the font size. Although you can specify a value in the range **-10** points through **150** points, the displayed point size never decreases below two points.

The numeric keypad plus (**+**), minus (**-**), and divide (**/**) character keys in conjunction with the Ctrl key set the zoom value up by one, down by one, and to zero (**0**), respectively.

Window, Form, and Control Methods

This section describes the methods defined in the following classes.

- **Window**
- **Form**
- **Control** class and subclasses

You can access all GUI properties and methods (which are marked as **clientExecution** methods) from a server method except for anything that brings up a modal-type dialog (that is, the common dialog class methods, the **app.msgBox**, and the **showModal** and **popupMenu** methods in the **Form** class). The other exceptions to this are the **app.doWindowEvents**, **app.checkPictureFile**, and **app.loadPicture** methods, which are executed relative to the server.

Caution Use of GUI methods and properties is *very* expensive in a server method. A **clientExecution** method requires that all transient objects passed to the server are passed back with the client execution (and passed back to the server after the client execution is complete).

Accessing GUI properties and methods within a server execution therefore should be done only in exceptional circumstances.

aboutBox

Signature `aboutBox();`

The **aboutBox** method of the **Window** class and **Control** class initiates the About box of the application by default. However, you can reimplement this method for subclass controls to allow a specific About box to be displayed, depending on the window or control that is involved.

An exception is raised if this method is invoked from a server method.

The **aboutBox** method is called when a user clicks on the **About Box** button in the Properties form of the JADE Painter.

Although the **aboutBox** method is primarily for ActiveX control implementations, it can be used for local control subclasses.

accessCell

Signature
```
accessCell(row:    Integer;
           column: Integer): JadeTableCell;
```

The **accessCell** method of the **Table** class returns a reference to the **JadeTableCell** object for the requested row and column (specified in the **row** and **column** parameters) for the current **topSheet** of the table.

Accessing a cell using this method or the **JadeTableSheet** class **accessCell** method sets the corresponding **Table** class **accessedCell** property to the returned cell so that it can be used for subsequent access.

The following code fragments show the use of the **accessCell** method.

```
table1.accessCell(2,3).inputType := Table.InputType_TextBox;
table1.accessedCell.foreColor    := Red;
```

```
table1.accessSheet(2).accessCell(1,4).text := "Company";
table1.accessedCell.alignment              := Table.Alignment_Right_Middle;
```

Storing a reference to a returned cell causes problems unless you take a copy of that cell, as shown in the following example in which both **cell1** and **cell2** refer to the same object, which is referencing **cell(3, 4)**.

```
cell1      := table1.accessCell(2, 3);
cell2      := table1.accessCell(3, 4);
cell1.text := "abc";
```

In the following example, **cell1** has been cloned and still refers to **cell(2, 3)**.

```
cell1      := table1.accessCell(2, 3).cloneSelf(true);
// the cloned cell must be deleted by your logic
cell2      := table1.accessCell(3, 4);
cell1.text := "abc";
```

Note Your logic must delete cloned cells.

See also the **Table** class **accessColumn**, **accessRow**, and **accessSheet** methods.

accessColumn

Signature `accessColumn(column: Integer): JadeTableColumn;`

The **accessColumn** method of the **Table** class returns a reference to the **JadeTableColumn** object for the requested column specified in the **column** parameter for the current **topSheet** of the table.

Accessing a column using this method or the **JadeTableSheet** class **accessColumn** method sets the corresponding **Table** class **accessedColumn** property to the returned column so that it can be used for subsequent access.

The code fragment in the following example shows the use of the **accessColumn** method.

```
table1.accessColumn(2).alignment := 2;
table1.accessedColumn.backColor  := Blue;
```

Storing a reference to a returned column causes problems unless you take a copy of that column.

Note Your logic must delete cloned columns.

See also the **Table** class **accessCell**, **accessRow**, and **accessSheet** methods.

accessRow

Signature `accessRow(row: Integer): JadeTableRow;`

The **accessRow** method of the **Table** class returns a reference to the **JadeTableRow** object for the requested row (specified in the **row** parameter) for the current **topSheet** of the table. Accessing a row using this method or the **JadeTableSheet** class **accessRow** method sets the corresponding **Table** class **accessedRow** property to the returned row so that it can be used for subsequent access.

Storing a reference to a returned row causes problems unless you take a copy of that row.

Note Your logic must delete cloned rows.

The code fragment in the following example shows the use of the **accessRow** method.

```
table1.accessRow(3).visible  := true;
table1.accessedRow.backColor := Blue;
```

See also the **Table** class **accessCell**, **accessColumn**, and **accessSheet** methods.

accessSheet

Signature `accessSheet(sheet: Integer): JadeTableSheet;`

The **accessSheet** method of the **Table** class returns a reference to the **JadeTableSheet** object for the requested sheet (specified in the **sheet** parameter) of the table. Accessing a sheet using this method **accessCell** sets the corresponding **Table** class **accessedSheet** property to the returned sheet so that it can be used for subsequent access.

Storing a reference to a returned sheet causes problems unless you take a copy of that sheet.

Note Your logic must delete cloned sheets.

The code fragment in the following example shows the use of the **accessSheet** method.

```
table1.accessSheet(1).visible := false;
table1.accessSheet(2).visible := true;
table1.accessSheet(2).accessCell(1,4).text := "Company";
table1.accessedCell.alignment              := Table.Alignment_Right_Middle;
```

See also the **Table** class **accessCell**, **accessColumn**, and **accessRow** methods.

activeChild

Signature `activeChild(): Form;`

The **activeChild** method of the **Form** class returns a reference to the current active MDI child for an MDI frame form. If there is no active child or the calling form is not an MDI frame, the **activeChild** method returns **null**. One child form only can be active at any time for an MDI frame.

The code fragment in the following example shows the use of the **activeChild** method.

```
statusLine.caption := activeChild.name;
```

addControl

Signature `addControl(c: Control);`

The **addControl** method of the **Form** class dynamically adds the control referenced in the **c** parameter to a form at run time.

If the **c** parameter references a persistent object, the call is rejected.

Note If you use this method to add a control to a form in the JADE Painter, see also the **flagControlForSave** method.

The following example shows the use of the **addControl** method.

```
vars
    btn : Button;
begin
    create btn transient;          // create the control
    btn.name    := "testButton";   // set the name
    btn.parent  := testFrame;      // parent of button is a frame called
                                   // test frame
    btn.caption := "Test Button";  // set the caption
    self.addControl(btn);          // add the button to the form
end;
```

When you create the control, the constructor for the control class sets properties for the control to their default values. In particular:

- The **tabIndex** property is set to zero (**0**).
- The control name is set to the control class name; for example, **button**.
- The control **text** or **caption** property is set to the control name; for example, **button**.
- The **top** and **left** properties are always set to zero (**0**), but default values are assigned to **width** and **height**.
- The parent defaults to the form at run time.

  In JADE Painter at design time, if you make the parent of the control the form, the control appears not to have been drawn. The Painter overlays a frame on the form, which is the parent object for any controls that are painted directly onto the form. The constructor of the control should therefore set its parent to a specific object other than the form. (This applies only to transient controls that are created from within another custom control.)

  If you do not set the parent of the control in the JADE Painter, the control is added to the Painter form rather than to the form that you are editing.

If you want to set the parent property to a control rather than the form, do this before calling the **addControl** method. This value defaults to the form (during processing of the **addControl** method).

The font of the control is not set. The default application font (see the **Application** class **fontName** property) is used when the **addControl** method is called. Any properties of the control that can be set in Painter can be set before calling the **addControl** method.

When a form is created with **scaleForm** set to **true**, dynamically added controls are also scaled when the current dpi is different from the dpi used to paint the form. Your logic should create the controls in the size relative to the dpi setting under which the form was created.

Properties that can be set only at run time (for example, the **listIndex** property of the **ListBox** control class) cannot be set before the **addControl** method is called. Methods for the control cannot be invoked until after the **addControl** method.

The added control generates event method calls using the name of the control; for example, **name_click**. Any required methods must be predefined in the form where the control is added (for details, see the **setEventMapping** method).

Use the **delete** instruction to remove a control from a form and destroy it. If **delete** is not called, the control is destroyed when the form is unloaded.

Note The **addControl** method differs from the **loadControl** method, in that the **loadControl** method creates a copy of an existing control and adds it to a form at run time.

addItem

Signature `addItem(str: String): Integer;`

The **addItem** method adds a new item to a **ComboBox** or **ListBox** control or adds a new row to a **Table** control at run time. Use the **str** parameter to specify the string expression to add to the control. For table controls only, use the tab character (character code **09**) to separate multiple strings that you want inserted into each column of a newly added row.

The **addItem** method places the item in the list box, combo box, or table specified by the control. The item is added at the correct sorted position (if the **sorted** property value is set to **true**) or to the end of the list (if the **sorted** property value is **false**).

The position at which the item was added is returned. This value is the same as that contained in the **newIndex** method.

Note Adding an entry can result in the value of the **topIndex** and **listIndex** properties for a list box being changed because of the addition.

For the **ListBox** class, which uses a hierarchy, entries added to the end of the list of entries have an **itemLevel** property value of **1** and are automatically visible within the hierarchy.

For the **Table** control, the following applies.

- A new row is added to the table.
- The first cell in the row is filled with the text that is passed.

  If the text contains tab characters, each tab character is assumed to be the end of the text for a cell, and the next cell is then filled with the remainder of the text, and so on. If all cells are filled and there is more text, that text is discarded.

If the sheet of the table is sorted, the location of the added row depends on the sorted position of that text. The **addItem** method returns the row position. This situation could result in the values of the **row**, **column**, **leftColumn**, and **topRow** properties changing.

When the text of a sorted column changes, the automatic sorting of rows occurs only when the **Table** class **addItem** method adds a new row or the **Table** class **resort** method is used.

The following example shows the use of the **addItem** method.

```
loadListBox() updating;
vars
    prod : Product;
begin
    foreach prod in app.myCompany.allProducts do
        listInstances.addItem(prod.display);
    endforeach;
end;
```

The code fragments in the following examples show the use of the **addItem** method for a table.

```
while count < 1 do
    app.printer.print(printTest.frame1);
```

```
        printTest.table1.addItem("a" & Tab & "b");
        count := count + 1;
    endwhile;

    tblPortfolio.row := tblPortfolio.addItem(portfolio.myCompany.name);

    // add a new row that has two columns for company name and address line 1
    table1.accessSheet(2).addItem(coy.name & Tab & coy.address1);
```

addItemAt

Signature

```
addItemAt(str:   String;
          index: Integer);
```

The **addItemAt** method adds a specified item index to a **ComboBox** or **ListBox** control or adds a new row to a **Table** control at run time.

The parameters of the **addItemAt** method are listed in the following table.

Parameter	Description
str	The string expression to add to the control. For **Table** controls only, use the tab character (character code **09**) to separate multiple strings that you want inserted into each column of a newly added row.
index	An integer representing the position within the control where the new item or row is placed. For the first item in a list box or combo box, or for the first row in a table control, the index is **1**.

The **addItemAt** method places the item at that specified position within the control. If the **sorted** property value is set to **true**, this may upset the sorting process.

Note Adding an entry can result in the value of the **topIndex** and **listIndex** properties for a list box being changed because of the addition. For a **Table** control, the **row**, **column**, **leftColumn**, and **topRow** properties can change as a result of the addition.

For the **ListBox** class, which uses a hierarchy, the following applies.

- Entries added to the end of the list of entries have an **itemLevel** property value of **1** and are automatically visible within the hierarchy.
- Entries inserted within the list adopt the level of the entry at which the insertion occurs. This means that they become leaves, having no subitems. They adopt the same visibility as the entry at which the insertion occurs.

For the **Table** class, the following applies.

- A new row is added to the table.
- The first cell in the row is filled with the text that is passed. If the text contains tab characters, each tab character is assumed to be the end of the text for a cell, and the next cell is then filled with the remainder of the text, and so on. If all cells are filled and there is more text, that text is discarded.

No sorting is performed, but the insertion of a row may still affect other property values.

addText

Signature
```
addText(options: Integer;
        text:    String);
```

The **addText** method of the **JadeTextEdit** class adds the string specified in the **text** parameter to the control content by using the options specified in the **options** parameter. End-of-line conversion is not performed.

The addition is a single undo action.

The **options** parameter can be one of the **JadeTextEdit** class constants listed in the following table.

Class Constant	Value	Description
ADDTXT_ADD	2	Inserts the text after the current position, which remains unchanged
ADDTXT_APPEND	3	Adds the text to the end of the existing text and moves the caret to the end of the new text and into view
ADDTXT_INSERT	1	Inserts the text before the current position and moves the caret to the end of the added text but does not force it into view
ADDTXT_INSERTREPLACESEL	4	Clears the current selection (if any) and then inserts the text specified in the **text** parameter before the current caret position, moving the caret to the end of the new text

addWebEventMapping

Signature
```
addWebEventMapping(eventName:      String;
                   scriptFunction: String): Boolean;
```

The **addWebEventMapping** method of the **Window** class adds a function to be invoked when the event specified in the **eventName** parameter occurs.

Use the **scriptFunction** parameter to specify the function that is to be invoked. For example, the following code fragment processes a **lostFocus** event on a **TextBox** control on the client Web browser.

```
textbox.addWebEventMapping("onLostFocus", "processLostFocus(this, 'lostFocus')");
```

The code fragment in the above example generates the following HTML fragment, which has been simplified for clarity.

```
<input type=text name=textbox onLostFocus="processLostFocus(this, 'lostFocus');">
```

In the HTML fragment shown in the above example, the **processLostFocus** script function is called when the text box loses focus. It is your responsibility to write this function and include this as part of the **userScript** property when you dynamically specify it in your code. As your function is not validated, it is your responsibility to ensure that it works as intended.

This method returns **false** if the function name is the same as the name that JADE generates automatically, and the mapping is not added to the control.

Tip In general, do not use functions that begin with **_ jade**.

The code fragment in the following example displays an alert message box but does not submit the form.

```
btnOK.addWebEventMapping("onClick", "alert('enter value'); return false;");
```

allowWebPrinting

Signature `allowWebPrinting(allow: Boolean);`

The **allowWebPrinting** method of the **Form** class enables you to set the **allow** parameter to **true** so that Microsoft Internet Explorer 4 and higher Web browsers generate slightly different code. (This parameter is set to **false**, by default.)

Setting the **allow** parameter to **true** for these Web browsers enables the correct printing of the contents of the Web page when the **Print** command is selected in the Web browser File menu. The requirement of this setting is content-dependent (for example, sometimes Internet Explorer 5.5 or higher may not print segments or pages of the displayed Web page because of the way that Internet Explorer handles the style sheet settings).

alwaysOnTop

Signature `alwaysOnTop(onTop: Boolean);`

The **alwaysOnTop** method of the **Form** class enables you to place a visible form above all other forms on the desktop when called with the **onTop** parameter set to **true**.

The form remains on top even after another form is activated. To restore the normal behavior, call the **alwaysOnTop** method with the **onTop** parameter set to **false**. (This parameter is set to **false**, by default.)

Note Calling the **alwaysOnTop** method for an MDI child form has no effect.

animateWindow

Signature
```
animateWindow(millisecs:   Integer
              animateType: Integer) clientExecution;
```

The **animateWindow** method of the **Window** class enables you to specify special animation effects (roll, slide, collapse, or expand) when showing or hiding a form or . This allows for the display of an animated informational popup window without having to write a significant amount of logic, for example.

Calling the method is equivalent to toggling the **visible** property of a window where the resulting show or hide of the window or is animated. When the window or is not visible, calling the **animateWindow** method makes it visible. If the window or is visible, calling the **animateWindow** method makes it invisible.

The **animateWindow** method call on a window does not result in any animation being shown if the form is maximized, modal, or an MDI child.

The **Window** class implementation shows the border and caption of a form, and animates the contents inside it. This does not always produce the best appearance, and Windows does not always draw the animation correctly. If the form:

- Has no border and caption, the animation should work correctly and be more appealing.
- Is skinned, the effects apply to the whole form, including the border area, and the animation is drawn correctly.

If the form has not been shown, calling the **animateWindow** method causes the **load** method to be executed.

Use the **millisecs** parameter to specify the duration of the animation, in milliseconds. Typically, an animation takes 200 milliseconds to play.

The **animateType** parameter can be one of the **Window** or class constants listed in the following table.

Class Constant	Value	Description
AnimateWindow_Flags_Activate	#20000	Causes the form to be activated when it is shown. If it is not set, the form is shown without it gaining focus. In addition, when not set, the form retains its current **zOrder** position if it is redisplayed using the **animateWindow** method.
AnimateWindow_Flags_Blend	#80000	Causes the form to fade in or out. (This setting cannot be used with other animation effect options.)
AnimateWindow_Flags_BottomToTop	#8	Animates the window from bottom to top. This flag can be used with roll or slide animation. It is ignored when used with **AnimateWindow_Flags_Center**.
AnimateWindow_Flags_Center	#10	Makes the window appear to collapse inward when the window is hidden or expand outward when the window is shown.
AnimateWindow_Flags_LeftToRight	#1	Animates the window from left to right. This flag can be used with roll or slide animation. It is ignored when used with **AnimateWindow_Center**.
AnimateWindow_Flags_RightToLeft	#2	Animates the window from right to left. This flag can be used with roll or slide animation. It is ignored when used with **AnimateWindow_Flags_Center**.
AnimateWindow_Flags_Slide	#40000	Uses slide animation. By default, roll animation is used. This flag is ignored when used with **AnimateWindow_Center**.
AnimateWindow_Flags_TopToBottom	#4	Animates the window from top to bottom. This flag can be used with roll or slide animation. It is ignored when used with **AnimateWindow_Flags_Center**.

The following code fragment examples demonstrate the use of the **animateWindow** method. In the following example, the method specifies that the control will slide the control into view from right to left when made visible and slide the form out of view right to left when made invisible.

```
frame1.animateWindow(200, Window.AnimateWindow_Flag_Slide +
                          Window.AnimateWindow_Flag_RightToLeft);
```

In the following example, the method specifies that the form will roll into view from the top left corner when made visible and roll the form out of view towards the top left corner when made invisible.

```
frame1.animateWindow(200, Window.AnimateWindow_Flag_RightToLeft +
                          Window.AnimateWindow_Flag_TopToBottom);
```

To animate the show of a form or , position the form or with its final position and size while invisible, and then call the **animateWindow** method to animate the window into that final position and size.

Notes If the **animateType** parameter is not valid, the form or is made visible or invisible without any animation.

The **animateWindow** method is available only in GUI applications and for JADE forms (that is, it is not available for Web forms).

An invalid parameter exception is generated if the value of the **millisecs** parameter is less than zero (**0**) or greater than **60,000**.

The animation may not work effectively if other windows are moved at the same time. This applies to forms or that are docked or aligned to their parents.

Applies to Version: 2016.0.01

append

Signature `append(text: String);`

The **append** method of the **JadeRichText** class loads the text specified in the **text** parameter into the control of the receiver, appending it to the end of the current contents. For an example of the use of this method, see "JadeRichText Control Method Example", earlier in this document.

applySettings

Signature `applySettings(): Integer;`

The **applySettings** method of the **JadeTextEdit** class searches the application and global settings tables and applies entries that are associated with the current language.

This method returns zero (**0**) if the settings were successfully applied or it returns a JADE error code if the action was unsuccessful. (For details about the causes and actions of JADE error codes, see the appropriate error code in the **JADEMsgs.pdf** file.)

The code fragment in the following example shows the use of the **applySettings** method.

```
jte.language := JadeTextEdit.SCLEX_CPP;
jte.applySettings();
jte.restyleText();
```

This example changes the current language to the C/C++ family, applies the appropriate settings (for example, text style and keyword lists), and forces the current text to be restyled for the settings.

See also the **Application** class **getJadeTextEditGlobalSettings**, **getJadeTextEditOneSetting**, and **updateJadeTextEditAppSettings** methods and the **JadeTextEdit** class **updateAppSettings** method.

applyVerb

Signature `applyVerb(verb: String);`

The **applyVerb** method of the **OleControl** class calls the application with the specified verb. The standard verbs are listed in the following table. (The OLE server that is being used determines the list of available verbs.)

Verb	Action
primary	Executes the primary verb, in conjunction with the Ctrl key, defined by the application. (Usually **edit** or **play**.)

Verb	Action
show	Displays the object.
open	Runs the application in normal mode (not in-place).
hide	Hides the object.

beginNotfiyAutomationEvent

Signature
```
beginNotifyAutomationEvent(receiver:          Object;
                           eventClassRefName: String);
```

The **beginNotifyAutomationEvent** method of the **ActiveXControl** class registers the receiver to be notified when a specified event occurs on an ActiveX control object when it has been created as an automation object. (For details, see the **makeAutomationObject** method.)

The control object that invokes the **beginNotifyAutomationEvent** is referred to as the *subscriber*.

An object that subscribes to an automation notification is notified when the nominated event occurs for that object.

The parameters for this method are listed in the following table.

Parameter	Description
receiver	The object that is to receive the event notification
eventClassRefName	The name of the reference (an instance of the **IDispatch** subclass) that implements the notification events

A method implemented by the **eventClassRefName** parameter is executed each time its corresponding automation event occurs.

This event notification continues until the **ActiveXControl** object is deleted or until the **endNotifyAutomationEvent** method is called. The **endNotifyAutomationEvent** method has the same signature as the **beginNotifyAutomationEvent** method.

Caution There may be an impact on performance, particularly in JADE thin client mode or on a slow communications link, if you register for large numbers of automation events or events that are triggered often.

bindKeyToCommand

Signature
```
bindKeyToCommand(keycode: Integer;
                 command: Integer);
```

The **bindKeyToCommand** method of the **JadeTextEdit** class enables you to associate the key combination specified in the **keycode** parameter to the action specified in the **command** parameter, which can include one of the **JadeTextEdit** class constants listed in the following table.

SCI_BACKTAB	SCI_FINDAGAIN
SCI_FINDNEXT	SCI_FINDPREV
SCI_GONEXT_JADE_BRKPNT	SCI_GONEXT_JADE_LINEMARK
SCI_GOPRIOR_JADE_BRKPNT	SCI_GOPRIOR_JADE_LINEMARK

SCI_TOGGLEFOLDERHERE	SCI_TOGGLE_JADE_BREAKPOINT
SCI_TAB	SCI_TOGGLE_JADE_DEBUG
SCI_TOGGLE_JADE_LINEMARK	SCI_ZOOMIN
SCI_ZOOMOUT	

See http://scintilla.sourceforge.net/ScintillaDoc.html, for details about additional commands that are available in Scintilla.

Note The bound key action has the highest priority when the **JadeTextEdit** control has focus. Bound key actions override menu accelerator and menu shortcut event methods.

Although you can use the **keycode** parameter to specify any key combination, you can also use the JADE global constants in the **KeyCharacterCodes** category; for example, **J_key_F2** or **J_key_F12**.

In addition, you can use the **JadeTextEdit** class **KEYMOD_ALT**, **KEYMOD_CTRL**, and **KEYMOD_SHIFT** constants. For example, **J_key_F1** + **KEYMOD_SHIFT** + **KEYMOD_CTRL** indicates the Ctrl+Shift+F1 key combination.

The code fragment in the following example shows the use of the **bindKeyToCommand** method.

```
// Toggle folding
jte.bindKeyToCommand('R'.Integer + JadeTextEdit.KEYMOD_CTRL,
          JadeTextEdit.SCI_TOGGLEFOLDHERE);
// Find next/previous
jte.bindKeyToCommand(J_key_F4, SCI_FINDNEXT);
jte.bindKeyToCommand(J_key_F4 + JadeTextEdit.KEYMOD_SHIFT, SCI_FINDPREV);
// Toggle and go to linemark
jte.setLinemarkAttributes(JadeTextEdit.MARKER_JAD_LINEMARK, 0,
          jteSource.rgb(255,180,180), jteSource.rgb(128,255,255));
jte.bindKeyToCommand(J_key_F2 + JadeTextEdit.KEYMOD_CTRL,
          JadeTextEdit.SCI_TOGGLE_JADE_LINEMARK);
jte.bindKeyToCommand(J_key_F2, JadeTextEdit.SCI_GONEXT_JADE_LINEMARK);
jte.bindKeyToCommand(J_key_F2 + JadeTextEdit.KEYMOD_SHIFT,
          JadeTextEdit.SCI_GOPRIOR_JADE_LINEMARK);
```

bindKeyToNotification

Signature
```
bindKeyToNotification(keycode:  Integer;
                      eventTag: Integer);
```

The **bindKeyToNotification** method of the **JadeTextEdit** class assigns the key combination specified in the **keycode** parameter to the notification message specified in the **eventTag** parameter. When the key combination is pressed, the notification is generated and a user notification occurs; that is, the **userNotify** event is called. (For details about receiving notifications, see "Receiving User Notifications", in Chapter 2 of the *JADE Developer's Reference*.)

Note The bound key action has the highest priority when the **JadeTextEdit** control has focus. Bound key actions override menu accelerator and menu shortcut event methods.

Although you can use the **keycode** parameter to specify any key combination, you can also use the JADE global constants in the **KeyCharacterCodes** category; for example, **J_key_F2** or **J_key_F12**.

In addition, you can use the JadeTextEdit class **KEYMOD_ALT**, **KEYMOD_CTRL**, and **KEYMOD_SHIFT** constants. For example, **J_key_F3** + **KEYMOD_SHIFT** + **KEYMOD_CTRL** indicates the Ctrl+Shift+F3 key combination.

The **eventTag** parameter is a user-defined integer value (for example, an index into an array) that identifies a notification subscription that is passed to the notification callback method. The notification is the JadeTextEdit class **EVENTTYPE_BOUNDKEY** constant.

The code fragment in the following example shows the use of the **bindKeyToNotification** method.

```
// Breakpoint linemark
jteSource.bindKeyToNotification(J_key_1 + JadeTextEdit.KEYMOD_CTRL, 1001);
```

The method in the following example shows a userNotify event for a bound key notification.

```
handleUserNotify(textedit:  JadeTextEdit input;
                 eventType: Integer;
                 theObject: Object;
                 eventTag:  Integer;
                 userInfo:  Any);
vars
begin
    if eventType = JadeTextEdit.EVENTTYPE_BOUNDKEY then
        if eventTag = 1001 then // Ctrl+1
            self.doCtrl_1();
        endif;
    endif;
end;
```

bindKeyToText

Signature
```
bindKeyToText(keycode:  Integer;
              text:     String);
```

The **bindKeyToText** method of the JadeTextEdit class assigns the key combination specified in the **keycode** parameter to the string specified in the **text** parameter; for example, JADE accelerator keys.

Note The bound key action has the highest priority when the **JadeTextEdit** control has focus. Bound key actions override menu accelerator and menu shortcut event methods.

Although you can use the **keycode** parameter to specify any key combination, you can also use the JADE global constants in the KeyCharacterCodes category; for example, **J_key_A**.

In addition, you can use the JadeTextEdit class **KEYMOD_ALT**, **KEYMOD_CTRL**, and **KEYMOD_SHIFT** constants. For example, **J_key_A** + **KEYMOD_SHIFT** + **KEYMOD_CTRL** indicates the Ctrl+Shift+A key combination.

The code fragment in the following example shows the use of the **bindKeyToText** method.

```
// Bind the accelerator keys
jte.bindKeyToText('A'.Integer + JadeTextEdit.KEYMOD_SHIFT +
                  JadeTextEdit.KEYMOD_CTRL, "abortTransaction;");
jte.bindKeyToText('B'.Integer + JadeTextEdit.KEYMOD_SHIFT +
                  JadeTextEdit.KEYMOD_CTRL, "beginTransaction;");
jte.bindKeyToText('C'.Integer + JadeTextEdit.KEYMOD_SHIFT +
                  JadeTextEdit.KEYMOD_CTRL, "commitTransaction;");
```

```
jte.bindKeyToText('E'.Integer + JadeTextEdit.KEYMOD_SHIFT +
                  JadeTextEdit.KEYMOD_CTRL, "endforeach;");
```

callMethod

Signature
```
callMethod(controlName: String;
           memberName:  String;
           paramList:   ParamListType);
```

The **callMethod** method of the **JadeXamlControl** class enables you to execute a Windows Presentation Foundation (WPF) method on an entity of the XAML control.

The parameters are combined to form a sequence of accesses to the WPF entities involved. The JADE method parameters are a mixture of property names, method names, and WPF method parameters, as described in the following table.

Parameter	Description
controlName	Name of the WPF FrameworkElement involved. If the name is null or equal to the control name, the search for the **memberName** starts with the parent control; otherwise the search starts with the first child element with the specified name. The search succeeds when the entity or one of its children is found to have the specified **memberName** value. An exception is raised if the **controlName** or **memberName** is not found.
memberName	Name of the first method or property being accessed.
paramList	Remaining property, methods, and parameters being used in sequence.

The code fragments in the following examples show the use of these parameters.

```
jadeXamlCtl.callMethod(null, "BringIntoView");
     // brings the base XAML control into view

jadeXamlCtl.callMethod("list", "BringIntoView");
     // brings the child control named "list" into view

jadeXamlCtl.callMethod("list", "SelectedItem", "BringIntoView");
     // brings the currently selected item of a list box into view by
     // executing the WPF sequence: list.SelectedItem.BringIntoView

JadeXamlCtl.callMethod("list", "Items", "GetItemAt", 2, "BringIntoView")
     // brings the second item of the Items 'collection' property of the
     // "list" ListBox item to be brought into view by executing the
     // WPF sequence:  list.Items.GetItemAt(2).BringIntoView.
```

Note the following restrictions.

- Only JADE primitives types are supported as parameters to WPF method calls.
- Access to static WPF properties and methods is not supported.
- Any return object from the final method is ignored.
- Parameter types must match the same basic type; that is, an **Integer** parameter must be passed as an **Integer**, a floating point or real number as a **Real**, a byte as a **Byte**, and so on.

- For a presentation client, calls to this method are buffered (the application server does not wait for a reply). If a call fails, an exception is raised. However, the current line of logic listed as causing the exception does not indicate where the method was called.

canBeChildOf

Signature

```
canBeChildOf(proposedParent:  Window;
             rejectionReason: String output): Boolean;
```

The **canBeChildOf** method of the **Control** class is used by the Painter to determine whether the control can be placed on the form or control specified in the **proposedParent** parameter.

If a value of **false** is returned, a message can be returned in the **rejectionReason** parameter. This message is displayed on the status line of the JADE Painter when dragging an existing control or trying to create a new control on top of a control that does not support this dragged or new control; for example, it controls the fact that a **Sheet** control can be placed only on a **Folder** control.

At the **Control** class level, this method always returns the same as the **canControlHaveChildren** method on the parent. It is reimplemented in the **Sheet** and **WebHotSpot** classes to ensure that sheets are placed only on folders and that Web hotspots are placed only on **Picture** controls.

canControlHaveChildren

Signature

```
canControlHaveChildren(): Boolean;
```

The **canControlHaveChildren** method of the **Control** class returns whether a control is permitted to be the parent of other controls.

The **canControlHaveChildren** method is defined for the following reasons.

- The Painter needs to determine the controls that can be parents.
- For subclasses of the **BaseControl** class, it can be overridden to determine the behavior of the developer-defined subclassed control in regards to being a parent.

The default value returned by the method is determined by the control. The **BaseControl**, **Frame**, **GroupBox**, **JadeDockBar**, **JadeDockContainer**, **JadeMask**, **Picture**, **Sheet**, and **StatusLine** controls return **true**; all other controls return **false**.

canEject

Signature

```
canEject(): Boolean;
```

The **canEject** method of the **MultiMedia** class returns whether the device can eject its media.

If the **useDotNetVersion** property is set to **true**, the **canEject** method returns a fixed value of **false**.

The method in the following example shows the use of the **canEject** method.

```
eject_click(btn: Button input) updating;
begin
    if cd.canEject then
        cd.stop;
        cd.eject;
    endif;
end;
```

canHaveAsChild

Signature
```
canHaveAsChild(proposedChild:   Control;
               rejectionReason: String output): Boolean;
```

The **canHaveAsChild** method of the **Control** class is used by the Painter to determine whether the control specified in the **proposedChild** parameter can be placed on the control. If this method returns **false**, a message can be returned in the **rejectionReason** parameter.

This message is displayed on the status line of the JADE Painter when dragging an existing control or trying to create a new control on top of a control that does not support this dragged or new control; for example, it controls the fact that a **Folder** control can have only **Sheet** controls as children.

At the **Control** class level, this method always returns the same as the **canControlHaveChildren** method. It is reimplemented in the **Folder** class to ensure that only sheets are placed only on folders.

canPaste_

Signature `canPaste_(): Boolean;`

The **canPaste_** method of the **JadeRichText** class programmatically returns whether there is content such as text or an image in the Windows clipboard that can be pasted into the **JadeRichText** control. (You can also obtain this status by calling the **JadeRichText** class **getRedoAndUndoState** method.)

Applies to Version: 2020.0.01 and higher

canPlay

Signature `canPlay(): Boolean;`

The **canPlay** method of the **MultiMedia** class returns whether the device can play its media.

If the **useDotNetVersion** property is set to **true**, the **canPlay** method returns a fixed value of **true**.

See also the **play** and **playFromTo** methods.

canRecord

Signature `canRecord(): Boolean;`

The **canRecord** method of the **MultiMedia** class returns whether the device supports recording.

If the **useDotNetVersion** property is set to **true**, the **canRecord** method returns a fixed value of **false**.

See also the **record**, **newFile**, and **save** methods.

canSave

Signature `canSave(): Boolean;`

The **canSave** method of the **MultiMedia** class returns whether the device supports the saving of data.

If the **useDotNetVersion** property is set to **true**, the **canSave** method returns a fixed value of **false**.

See also the **save** method.

captureMouse

Signature `captureMouse();`

The **captureMouse** method of the **Window** class sets the mouse capture to the specified window for the application. When a window has captured the mouse, all mouse input is directed to that window, regardless of whether the cursor is positioned within the borders of that window. The mouse can be captured by only one window at a time by each application.

Use the **releaseMouse** method to release the mouse when the window no longer requires all mouse input.

centreWindow

Signature `centreWindow();`

The **centreWindow** method of the **Window**, **Form**, and **JadeDockBase** classes centers an MDI child form that is being opened, and positions it in the middle of the client area of its parent MDI frame. A non-MDI form is centered within the monitor on which the form resides. The **centreWindow** of the **Control** class centers the control in the middle of its parent. The method in the following example shows the use of the **centreWindow** method.

```
load() updating;
begin
    centreWindow;
    caption              := process.signOnUserCode;
    connectionName.text  := app.computerName;
    sendIt.value         := true;
    create tcp;
end;
```

changeKeywords

Signature
```
changeKeywords(action:      Integer;
               keywordList: Integer;
               keywords:    String);
```

The **changeKeywords** method of the **JadeTextEdit** class modifies one or more of the current keyword lists. The keyword lists are used by the current language lexical analyzer to classify the tokens found in the text. For the JADE language, this includes keywords, class names, constant names, and so on.

The value of the **action** parameter can be one of the **JadeTextEdit** class constants listed in the following table.

Class Constant	Value	Description
KEYWORDS_ADD	2	Adds the keywords specified in the **keywords** parameter to the list specified in the **keywordList** parameter.
KEYWORDS_DELETE	3	Deletes the words specified in the **keywords** parameter from the list specified in the **keywordList** parameter.
KEYWORDS_SET	1	Clears the list specified in the **keywordList** parameter then sets it to the words specified in the **keywords** parameter.
KEYWORDS_TOLANGDEF	4	Sets each keyword list to the default set of words for the current programming language. The value of the **keywordList** parameter must be zero (**0**).

Keyword list numbers are language-specific. JADE language keyword list numbers are the **JadeTextEdit** class constants listed in the following table.

Constant	Value	Constant	Value
KWL_JADE_GLOBALCONSTANTS	5	KWL_JADE_IMPORTEDCLASSES	8
KWL_JADE_INTERFACES	9	KWL_JADE_KEYWORDS	1
KWL_JADE_METHODWORDS	2	KWL_JADE_PACKAGES	7
KWL_JADE_SYSTEMCLASSES	4	KWL_JADE_SYSTEMVARS	3
KWL_JADE_USERCLASSES	6		

Use the **keywords** parameter to specify keywords separated by spaces, tabs, **Cr**, **Lf**, or any combination of these separators.

The code fragments in the following examples show the use of the **changeKeywords** method.

```
jteSource.changeKeywords(JadeTextEdit.KEYWORDS_SET, 5, "");

jteSource.changeKeywords(JadeTextEdit.KEYWORDS_ADD, 8, "nine");

jteSource.changeKeywords(JadeTextEdit.KEYWORDS_DELETE, 8, "one");

jteSource.changeKeywords(JadeTextEdit.KEYWORDS_ADD, 8,
                              "five six seven two");

jteSource.changeKeywords(JadeTextEdit.KEYWORDS_DELETE, 8, "two seven nine");
```

clear

Signature `clear();`

The **clear** method clears the contents of a **ListBox** or **ComboBox** control or the contents of the current sheet of a **Table** control.

The **listCount** method of a list box or combo box control then returns **0**, following the **clear** method instruction.

For a table control, the **clear** method removes the contents of each cell of the current sheet of the table, including all cell properties (**itemBackColor**, **itemForeColor**, **itemText**, and so on). The number of rows and columns is retained, together with any **row** or **column** property values, which are reset to **1**.

Notes To clear the contents of an entire sheet, delete the sheet and then add it again (when the table has more than one sheet) or set the number of rows and columns to zero (**0**). The **clear** method detaches a collection from a control by clearing the **displayCollection** and the contents of the control.

The **displayCollection** method is not available in tables on forms in Web-enabled applications, as the HTML framework cannot predict the final size of your table and adjust the HTML page accordingly.

If you want to ensure that all possible entries in the table are displayed on a Web page, use some other "virtual window" for the Web generation or populate the table with all entries from the underlying collection if you know the number of rows will not cause excessive page size.

The methods in the following examples show the use of the **clear** method.

```
btnProcessLocks_click(btn: Button input) updating;
vars
```

```
    procs : ProcessDict;
    proc  : Process;
begin
    create procs transient;
    system.getObjectLockProcesses(obj, procs, 100);
    listBoxProcesses.clear;
    foreach proc in procs do
        listBoxProcesses.addItem(proc.userCode);
    endforeach;
end;

btnEdit_click(button: Button) updating;
begin
    if form.myCustomer <> null then
        listBoxCust.clear;
        loadListBox;
        btnEdit.enabled   := false;
        btnDelete.enabled := false;
    endif;
end;
```

clearAllSelected

Signature `clearAllSelected();`

The **clearAllSelected** method of the **Table** class clears all the **selected** properties of all cells for the current sheet of the table.

The **clearAllSelected** method of the **ListBox** class clears all selected items in the list box. This method has no effect if the **multiSelect** property is set to **MultiSelect_None**, as the **listIndex** item is always selected.

The method in the following example shows the use of the **clearAllSelected** method.

```
cdtable_rowColumnChg(table: Table input) updating;
vars
    indx  : Integer;
    count : Integer;
begin
    if table.row = 1 and table.column > 1 then
        table.sortColumn[1] := table.column;
        table.resort;
        count := table.rows - 1;
        indx  := 1;
        while indx <= count do
            table.setCellText(indx + 1, 1, indx.String);
            indx := indx + 1;
        endwhile;
        table.row    := 1;
        table.column := 1;
        table.clearAllSelected;
    endif;
end;
```

clearAllStyles

Signature `clearAllStyles();`

The **clearAllStyles** method of the **JadeTextEdit** class clears all text style information (for example, font, color, and so on) previously defined for the text editor (for example, when changing the programming language from text to JADE) and resets the text style values to the default values defined by the control properties.

The following actions are performed.

1. Initializes the default text style (**STYLE_DEFAULT**).
2. Sets the default text style attributes to the values of the **fontName**, **fontSize**, **fontItalic**, **fontBold**, **fontUnderline**, **foreColor**, and **backColor** properties for the control.
3. Copies the default style to all other styles.
4. Sets the line number margin style (**STYLE_LINENUMBER**) foreground color to the light gray RGB value (that is, to **#C0C0C0**).
5. Sets brace highlighting style (**STYLE_BRACELIGHT**) foreground color to the bright blue RGB value (that is, to **#0000FF**).
6. Sets the unmatched braces style (**STYLE_BRACEBAD**) foreground color to the red RGB value (that is, to **#FF0000**).
7. Sets the indent guides style(**STYLE_INDENTGUIDE**) foreground color to the light gray RGB value (that is, to **#C0C0C0**) and background color to the white RGB value (that is, to **#FFFFFF**).

For details about setting the individual attributes of a text style, see the **setStyleAttributes** method.

clearHTML

Signature `clearHTML();`

The **clearHTML** method of the **Frame** class clears all previously generated HTML code from the frame.

clearUndoBuffer

Signature `clearUndoBuffer();`

The **clearUndoBuffer** method of the **JadeRichText** class clears information from the undo buffer.

clearWebEventMappings

Signature `clearWebEventMappings();`

The **clearWebEventMappings** method of the **Window** class removes all Web event mappings for the receiver. For example, the following code fragment removes all of the event mappings for the **textBox1** control.

```
textBox1.clearWebEventMappings;
```

clientHeight

Signature `clientHeight(): Integer;`

The **clientHeight** method of the **Control** class returns the height of the client area of a control in pixels.

The client area of a control is the area inside the border area or scroll bars where controls can be placed. The position of child controls is relative to the top left of this client area.

The code fragment in the following example shows the use of the **clientHeight** method.

```
btn.height := clientHeight / (numOfRows + 1);
```

See also the **clientHeight** property of the **Form** class.

clientWidth

Signature `clientWidth(): Integer;`

The **clientWidth** method of the **Control** class returns the width of the client area of a control in pixels.

The client area of a control is the area inside the border area or scroll bars where controls can be placed. The position of child controls is relative to the top left of this client area.

The code fragment in the following example shows the use of the **clientWidth** method.

```
btn.width := clientWidth / ButtonWidthRatio;
```

See also the **clientWidth** property of the **Form** class.

cloneSelf

Signature `cloneSelf(transient: Boolean): SelfType;`

The **cloneSelf** method of the **OleControl** class creates a new instance of the same type as the receiver and copies the attributes of the receiver (including the contents of primitive arrays).

close

Signature `close();`

The **close** method of the **OleControl** class closes the presentation of the object. The control retains the saved image, but the control appears to be empty.

The **close** method of the **MultiMedia** class closes the associated device or a file associated with the control. The **mediaName** property will be empty, and any video image will be cleared.

You can then associate another file or device with the control by setting a new value for the **mediaName** property.

closeDropDown

Signature `closeDropDown();`

The **closeDropDown** method of the **ComboBox** class closes (hides) the presentation of the drop-down list of the combo box control. (Use the **isDroppedDown** method to determine if the drop-down list is currently open.)

colorAs6Hex

Signature `colorAs6Hex(col: Integer): String;`

The **colorAs6Hex** method of the **JadeTextEdit** class returns a six-character hexadecimal string in the RGB format (padded with leading zeros) of the color specified in the **col** parameter. The returned string is in the RGB format or it is **"000000"** (black) if the specified color is less than zero (**0**) or greater than **#FFFFFF** (white).

JADE uses the RGB scheme for colors. Using the appropriate RGB value can set each property. The valid range for a normal RGB color is zero (**0**) through **16,777,215** (#FFFFFF). The high byte of a number in this range equals **0**; the lower three bytes (from least- to most-significant byte) determine the amount of red, green, and blue, respectively.

The red, green, and blue components are each represented by a number in the range 0 through 255 **(#FF**). If the high byte is **128**, JADE uses the system colors, as defined in the Control Panel of the user.

To determine the **Integer** value of a color from the RGB values:

```
int:= RedValue + (GreenValue * 256) + (BlueValue * 256 * 256);
```

You can use this method when creating text to be passed to the **updateAppSettings** method.

configureFor_Jade

Signature `configureFor_Jade();`

The **configureFor_Jade** method of the **JadeTextEdit** class is an example method that performs basic configuration when preparing the **JadeTextEdit** control to display JADE method text. This method sets the language to JADE, clears the text styles, sets the default style and copies it to all styles, and then sets up the JADE-specific text style attributes (primarily the **foreColor** property value). It sets the keyword lists to the built-in JADE words and then calls the **restyleText** method so that the current text (if any) is displayed using the new configuration.

The following is an example of the **configureFor_Jade** method that you can call from your applications.

```
configureFor_Jade() updating, clientExecution;
vars
begin
    self.language := SCLEX_JADE;
    self.clearAllStyles();
    self.setStyleAttributes(STYLE_DEFAULT, self.fontName,
                  self.fontSize.Integer, rgb(0,0,0), rgb(255,255,232),
                  self.fontBold.Integer, self.fontItalic.Integer,
                  self.fontUnderline.Integer, ATTRIB_FALSE);
    self.copyDefaultToAllStyles();
    self.setStyleAttributes(SCE_JAD_SINGLECOLOR, "", ATTRIB_NOCHANGE,
                  rgb(000,000,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_DEFAULT, "", ATTRIB_NOCHANGE,
                  rgb(000,000,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_PUNCTUATION, "", ATTRIB_NOCHANGE,
                  rgb(000,000,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_COMMENT, "", ATTRIB_NOCHANGE,
                  rgb(128,128,128), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_COMMENTLINE, "", ATTRIB_NOCHANGE,
                  rgb(128,128,128), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_STRING1, "", ATTRIB_NOCHANGE,
                  rgb(255,000,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_STRING2, "", ATTRIB_NOCHANGE,
                  rgb(255,000,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_NUMBER, "", ATTRIB_NOCHANGE,
                  rgb(255,000,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_IDENTIFIER, "", ATTRIB_NOCHANGE,
                  rgb(000,000,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_KEYWORD, "", ATTRIB_NOCHANGE,
                  rgb(000,000,255), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_METHODWORD, "", ATTRIB_NOCHANGE,
                  rgb(000,000,255), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_SYSTEMVAR, "", ATTRIB_NOCHANGE,
                  rgb(128,000,128), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_SYSTEMCLASS, "", ATTRIB_NOCHANGE,
                  rgb(000,128,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_USERCLASS, "", ATTRIB_NOCHANGE,
                  rgb(000,128,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_GLOBALCONST, "", ATTRIB_NOCHANGE,
                  rgb(128,000,128), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_PACKAGE, "", ATTRIB_NOCHANGE,
                  rgb(000,128,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_PACKAGECLASS, "", ATTRIB_NOCHANGE,
                  rgb(000,128,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.setStyleAttributes(SCE_JAD_INTERFACE, "", ATTRIB_NOCHANGE,
                  rgb(000,128,000), ATTRIB_NOCHANGE, ATTRIB_NOCHANGE,
                  ATTRIB_NOCHANGE, ATTRIB_NOCHANGE, ATTRIB_NOCHANGE);
    self.changeKeywords(KEYWORDS_TOLANGDEF, 0, "");
    self.restyleText();
end;
```

configureFor_Text

Signature `configureFor_Text();`

The **configureFor_Text** method of the **JadeTextEdit** class is an example method that performs basic configuration when preparing the text editor to display plain text. This method clears the text styles and then sets up the text style attributes and keywords for plain text.

The following is an example of the **configureFor_Text** method that you can call from your applications.

```
configureFor_Text() updating, clientExecution;
vars
    ii : Integer;
begin
    self.language := SCLEX_TEXT;
    self.clearAllStyles();
    self.setStyleAttributes(STYLE_DEFAULT, self.fontName,
               self.fontSize.Integer, self.foreColor, self.backColor,
               self.fontBold.Integer, self.fontItalic.Integer,
               self.fontUnderline.Integer, ATTRIB_FALSE);
    self.copyDefaultToAllStyles();
    self.changeKeywords(KEYWORDS_TOLANGDEF, 0, "");
    self.restyleText();
end;
```

controlCount

Signature `controlCount(): Integer;`

The **controlCount** method of the **Form** class returns the number of controls on the form.

The following example examines all controls on a form.

```
vars
    cntrl : Control;
    indx  : Integer;
begin
    foreach indx in 1 to controlCount do
        cntrl := controls(indx);
        if cntrl.name = "Text1" then
            ...
        endif;
    endforeach;
end;
```

controlNamePrefix

Signature `controlNamePrefix(): String;`

The **controlNamePrefix** method of the **Window** class is a prototype method that you can reimplement in your own schemas if you want to prefix your own control names. You can define a prefix for any **Control** subclass (for example, a prefix of **lbx_** for a **ListBox** control) so that when you add a control to a form in Painter, JADE inserts the appropriate prefix when prompting you for the name of the control.

To prefix a control, you must first define (reimplement) a **controlNamePrefix** method for the appropriate control subclass.

The following example shows the reimplementation of this method defined for the **Button** class in a user-defined schema.

```
controlNamePrefix(): String;
vars
begin
```

```
        return "btn_";
    end;
```

The **controlNamePrefix** method is used only by the JADE Painter. No check for a control name prefix is made when you use the Class Browser or logic to add controls.

If you do not define a valid **controlNamePrefix** method for a control class, the JADE Painter uses the default control naming rules, which may be overridden.

If the control class has a **caption** property, the initial value for the caption is the control name (with a capitalized first letter and without the control name prefix applied) if you accept the default name when prompted to do so, or the actual text (with a capitalized first letter) that you specified as part of the name. For example, if the Painter detects you have changed the control name (for example, from **btn_Button1** to **btn_OK**), the **caption** property is made equal to the name with the prefix removed so that the caption displays only the value that you specified (in this example, **OK**).

The **name** property for the control displays the full name (**btn_OK** in this example) in the Properties dialog.

Notes The **controlNamePrefix** method can be defined once only for any class in a schema branch. You therefore cannot override a prefix if the control is in a superschema and it has already implemented the **controlNamePrefix** method. In this case, set the superschema to the current schema and then change the method.

Control prefixes are not applied when you create a form using the Form Wizard, which prefixes all controls with **ctl_**. (The type of control is not known until the form is built after you have specified the name of the control.)

controls

Signature `controls(controlNumber: Integer): Control;`

The **controls** method of the **Form** class enables logic to access the controls on an active form at run time. This method returns a reference to the active control object specified in the **controlNumber** parameter or **null** if there is no specified control.

The method returns an object of type **Control**, which enables the properties of a control to be accessed.

To access a property specific to a type of control, the object must be converted to a control of the appropriate type, as shown in the following example that examines all controls on a form.

```
    vars
        cntrl    : Control;
        indx     : Integer;
        textBox1 : TextBox;
    begin
        foreach indx in 1 to controlCount do
            cntrl := controls(indx);
            if cntrl.isKindOf(TextBox) then
                textBox1 := cntrl.TextBox;
                if textBox1.maxLength > 60 then
                    ...
                endif;
            endif;
        endforeach;
    end;
```

convertEndOfLines

Signature `convertEndOfLines(eolType: Integer);`

The **convertEndOfLines** method of the **JadeTextEdit** class changes the line endings in the text to the value requested by the **eolType** parameter.

The value of the **eolType** parameter can be one of the **JadeTextEdit** class constants listed in the following table.

Class Constant	Integer Value	Description
SC_EOL_CR	1	Carriage return character
SC_EOL_CRLF	0	Carriage return and line feed characters
SC_EOL_LF	2	Line feed character

convertFormPosition

Signature
```
convertFormPosition(x: Real io;
                    y: Real io);
```

The **convertFormPosition** method of the **Control** class converts the horizontal and vertical positions specified in the **x** and **y** parameters, respectively, into coordinates relative to the control.

The horizontal and vertical positions are relative to the form of the control (in pixels).

convertIndentWhitespace

Signature `convertIndentWhitespace(toTabs: Boolean): Integer;`

The **convertIndentWhitespace** method of the **JadeTextEdit** class changes all of the indentation whitespace to the style requested by the **toTabs** parameter. The indentation whitespace is the tab and space characters that precede the first visible character in each line.

The existing indent whitespace length for each line is calculated. Each tab character is counted using the current tab width. A new sequence of indent whitespace is created that has the same effective length as the original.

If the value of the **toTabs** parameter is **true**, the new indent white space is converted to a series of tab characters followed by any additional space characters to make up the required indentation length.

If the value of the **toTabs** parameter is **false**, the new indent whitespace is built using space characters only.

If the existing indent whitespace does not match the new whitespace, the change is applied.

This method returns the number of lines that are changed.

If any changes occur, the conversion is performed as a single undo action.

copyDefaultToAllStyles

Signature `copyDefaultToAllStyles();`

The **copyDefaultToAllStyles** method of the **JadeTextEdit** class copies the default text style and all of its attributes to all other text styles in the text editor.

Use this method to construct a consistent set of text styles before customizing individual styles to match language requirements by calling the **setStyleAttributes** method.

copyToClipboard

Signature `copyToClipboard(): Integer;`

The **copyToClipboard** method of the **JadeTextEdit** class copies the text selected in the text editor to the system clipboard and returns **1**.

Any end-of-line sequences in the selection are converted to the platform-native sequence in the text copied to the clipboard.

cutToClipboard

Signature `cutToClipboard(): Integer;`

The **cutToClipboard** method of the **JadeTextEdit** class cuts the selected text from the text editor and moves it to the system clipboard. This method returns **1**. The selection is cleared.

Any end-of-line sequences in the selection are converted to the platform-native sequence in the text copied to the clipboard.

create

Signature `create();`

The **create** method of the **Form** class calls a constructor that builds a description of the form as transient JADE objects using the JADE development definition of the form and then builds the physical windows involved. This form description includes all of the controls and menus.

The **create** method for a form is the same as for any JADE class instance. The form is not displayed until a **show** method or a **showModal** statement is executed or the **visible** property of the form is set to **true**.

The **create** method for a control creates a control object but no associated windows object until the **addControl** method is called.

The following example of the **create** method for an **ActiveXControl** assumes that you have imported the Microsoft **SysInfo** control (that is, **sysInfo.ocx**) as an ActiveX control. This example creates a JADE control instance and calls the **makeAutomationObject** method instead of adding the control to a form.

```
createSysInfoAsAutoObject();
vars
    actX : SysInfo;
begin
    create actX transient;
    actX.makeAutomationObject;
    write actX.oSVersion;
    write actX.oSBuild;
epilog
    delete actX;
end;
```

The **create** method of the **ProgressBar** subclass of the **Label** control class creates a progress bar, hides the **caption** property, and sets the **borderStyle** property to **1** (fixed single).

The **create** method for a **Label** class Web picture sets the **transparent** property to **true** and the **caption** property to null (**""**) when the hotspot is created.

createEventNameMap

Signature `createEventNameMap();`

The **createEventNameMap** method of the **JadeDotNetVisualComponent** class is a method that is reimplemented in subclasses of **JadeDotNetVisualComponent** by the .NET Import wizard to establish a mapping between the names of .NET events and the names of the JADE methods to be invoked.

You would not normally need to change the code generated for this method.

createPicture

Signature

```
createPicture(entireWindow:      Boolean;
              includeChildren:   Boolean;
              numberOfColorBits: Integer): Binary;
```

The **createPicture** method of the **Window** and **Control** class creates a bitmap for the receiver form or control, respectively.

Use the **entireWindow** parameter to specify that the created picture includes the non-client area of the form or control, the **includeChildren** parameter to specify that the created picture includes child controls (that is, controls that are placed on the form or control are included in the picture), and the **numberOfColorBits** parameter to specify the number of color bits for the picture. (The valid numbers of color bits that you can specify are **1**, **4**, **8**, or **24**.)

The **createPicture** method creates an image by drawing the window (and its children, if required) and replaying any drawing commands saved while the **autoRedraw** property is set to **true**. The process does not scrape the image off the screen.

The code fragments in the following examples show the use of the **createPicture** method.

```
setBackDrop(logo.createPicture(false, true, 24), 1, -1);

// create a picture and display the common File Save dialog to list all
// available Jpeg files for picture conversion
createPicture(true, true, 24).convertToFile("", Window.PictureType_Jpeg);
```

When you call this method for the **Window** class on an MDI frame that has MDI children, the frame and all of its children are included in the image. In addition, you can use this method to capture the entire image of a **Form** and its controls.

The created image is a reflection of how the window, which can include child windows, would be displayed. To recreate an image of a different size, set the image into a picture control with the **stretch** property set to **Stretch_ToControl** (1) and call **createPicture(false, false, 24)**.

The **createPicture** method for **Picture** and **JadeMask** Web images attempts to preserve any image transparency where possible. This applies only if all of the following are true.

- The value of the **entireWindow** parameter of the **createPicture** method is **false**.
- The value of the **includeChildren** parameter of the **createPicture** method is **false**.
- This image type is **BMP**, **PNG**, **JPEG**, **TIFF**, or **GIF**.
- The control does not implement a **paint** event.

- For a **JadeMask** control, the value of the **text** property is null.
- Either the image size is the same size as the client areas of the control or the image is stretched to fit the control.
- If the image is stretched, the value of the **transparentColor** property of the control must be set to the default **Black** color (**transparentColor** is ignored if the image is the correct size and all of the above are true).

If all of the above are true and the image size is:

- The same size as the client area of the control, a copy of the original image is returned by the **createPicture** method.
- Stretched, the image is scaled to fit the control and it is returned as a PNG image, thus preserving any transparency effects.

If any of the above is false, the **createPicture** method constructs the required image by drawing all of the control components into a new image. This results in any transparency effects being lost.

Note If you rely on the value of the **Picture** class **transparentColor** property for the images, images may have a black background.

See also the **Control** class **createPictureIndirect** method, which improves JADE thin client mode performance as it does not have to pass large binary large objects (blobs) back and forth between the presentation client and the application server.

createPictureAsType

Signature
```
createPictureAsType(entireWindow:      Boolean;
                    includeChildren:   Boolean;
                    numberOfColorBits: Integer;
                    imageType:         Integer): Binary;
```

The **createPictureAsType** method of the **Window** class creates an image of the specified image type for the presentation client receiver form or control, respectively.

This method effectively combines the following instructions into a single instruction and therefore requires only one message to be sent from the application server to the presentation client instead of two, thus reducing the network traffic for a presentation client. You can also reduce the size of the returned image significantly if you specify the **imageType** parameter as **PictureType_Png** instead of **PictureType_Bitmap**.

```
bin := window.createPicture(false, true, 24);

bin := bin.convertPicture(PictureType_Png);
```

Use the **entireWindow** parameter to specify that the created picture includes the non-client area of the form or control, the **includeChildren** parameter to specify that the created picture includes child controls (that is, controls that are placed on the form or control are included in the picture), and the **numberOfColorBits** parameter to specify the number of color bits for the picture. (The valid numbers of color bits that you can specify are **1**, **4**, **8**, or **24**.)

The **createPictureAsType** method creates an image by drawing the window (and its children, if required) and replaying any drawing commands saved while the **autoRedraw** property is set to **true**. The process does not scrape the image off the screen.

When you call this method for the **Window** class on an MDI frame that has MDI children, the frame and all of its children are included in the image. In addition, you can use this method to capture the entire image of a **Form** and its controls.

The created image is a reflection of how the window, which can include child windows, would be displayed. To recreate an image of a different size, set the image into a picture control with the **stretch** property set to **Stretch_ToControl** (1) and call **createPictureAsType(false, false, 24, PictureType_Png)**.

For **Picture** and **JadeMask** Web images, the **createPictureAsType** method attempts to preserve any image transparency, where possible. This applies only if all of the following are true.

- The value of the **entireWindow** parameter of the **createPictureAsType** method is **false**.
- The value of the **includeChildren** parameter of the **createPictureAsType** method is **false**.
- This image type is **BMP**, **PNG**, **JPEG**, **TIFF**, or **GIF**.
- The control does not implement a **paint** event.
- For a **JadeMask** control, the value of the **text** property is null.
- Either the image size is the same size as the client areas of the control or the image is stretched to fit the control.
- If the image is stretched, the value of the **transparentColor** property of the control must be set to the default **Black** color (**transparentColor** is ignored if the image is the correct size and all of the above are true).

If all of the above are true and the image size is:

- The same size as the client area of the control, a copy of the original image is returned by the **createPictureAsType** method.
- Stretched, the image is scaled to fit the control and it is returned as a PNG image, thus preserving any transparency effects.

If any of the above is false, the **createPictureAsType** method constructs the required image by drawing all of the control components into a new image. This results in any transparency effects being lost.

Note If you rely on the value of the **Picture** class **transparentColor** property for the images, images may have a black background.

Use the **imageType** parameter to specify the type of image to be generated. For a presentation client, the image type can be one of **PictureType_Bitmap**, **PictureType_Png**, **PictureType_Jpeg**, **PictureType_Jpeg2000**, or **PictureType_Tiff**.

See also the **Control** class **createPictureIndirect** method, which improves JADE thin client mode performance as it does not have to pass large binary large objects (blobs) back and forth between the presentation client and the application server.

createPictureIndirect

Signature
```
createPictureIndirect(entireWindow:    Boolean;
                      includeChildren: Boolean): Binary;
```

The **createPictureIndirect** method of the **Control** class creates a short binary that contains an instruction concerning the window to be copied. When the binary is assigned to a **Picture** control property, the picture is created using the current image of the requested window.

Note This method, which achieves the same as the **Control** class **createPicture** method (that is, it creates a bitmap for the receiver control), improves JADE thin client mode performance as it does not have to pass large binary large objects (blobs) back and forth between the presentation client and the application server.

The parameters for this method are listed in the following table.

Parameter	Description
entireWindow	If **true**, copies the entire window. If **false**, copies only the client area of the control.
includeChildren	If **true**, includes any children in the created image. If **false**, no children are included in the created image.

The **createPictureIndirect** method creates an image by drawing the control (and its children, if required) and replaying any drawing commands saved while the **autoRedraw** property is set to **true**. The process does not scrape the image off the screen.

The code fragment in the following example dynamically creates a color image to be placed in a list box.

```
picture1.backColor          := Red;
picture1.borderStyle        := 1;
listBox1.itemPicture[indx] := picture1.createPictureIndirect(true, false);
```

If you use the **createPicture** method in JADE thin client mode, the picture image would be created on the presentation client and brought back to the application server. Assigning the picture then causes the image to be sent back to the presentation client and to be cached, which could involve significant delays and overheads for a reasonable-size image.

Using the **createPictureIndirect** method, a small binary is sent instead and the image data itself is not transported between the presentation client and the application server.

Note Use of the **createPictureIndirect** method also means that actual image data cannot be accessed by using the **picture** property.

currentMaskColor

Signature `currentMaskColor(): Integer;`

The **currentMaskColor** method of the **JadeMask** class returns the color of the pixel in the mask picture corresponding to the last position of the mouse when it was over the control. This method would normally be called only during one of the mouse events for the mask control.

This method is relevant only when the **style** property of the **JadeMask** control is set to **Style_Mask_Color** (3). If the style is not **Style_Mask_Color**, the returned value is undefined.

An example of the use of this is a control that displays a map of a road network. The picture mask is built with the roads drawn in different colors and the rest of the map drawn as some other color (for example, white).

When the user moves the mouse over the control, the **mouseMove** event calls the **currentMaskColor** method. If the returned value is not white, the color is used to index the name of the road and this is displayed to the user in bubble help.

When using this style, the roll over (**pictureRollOver**) and roll under (**pictureRollUnder**) features do not apply.

delete

Signature `delete();`

The **delete** method of the **Form** class or **Control** class has additional meaning when the object has an associated Windows form or control. Deleting such an object causes the object of the window to be destroyed as well.

The **delete** method, which is the destructor method for the **Form** or **Control** class and its subclasses, is automatically invoked when an instance of the **Form** or **Control** class or subclass is deleted. You cannot call this method directly.

Deleting a running form causes the form to be unloaded and destroyed, and the transient form and control objects to be deleted. Deleting a control causes that control to be removed from its owner form. No further events are issued for the window that is being deleted, as the object is not available to receive these events.

Deleting a form queues the window for deletion, but if another window is deleted before the next idle point, previously queued deleted windows are re-evaluated for deletion. If the queued window or its children have no outstanding Windows message, there are no incomplete event methods for that window or its children, and the method that created the window has exited or the window was deleted, the physical window is deleted.

Notes If you want the **queryUnload** and **unload** events to occur when your form is deleted, use the **unloadForm** method instead of the **delete** method.

To ensure that a form has no orphaned children when deleting a window, the lowest child in the hierarchy is deleted first, followed by the next lowest child, and so forth up the hierarchy until the parent is deleted. You therefore do not need to handle the deletion the children of a form in your code, as the child or children no longer exist when the parent is deleted.

Avoid referring to user-defined properties or methods on the **Application** or **Global** classes in the **delete** method. When the form is displayed in the JADE Painter, the application used is not the user application, so these properties or methods are not available at run time. If the property or method must be referenced, the method must perform a runtime check to ensure that the application is the expected one; otherwise, an exception will be raised.

deleteColumn

Signature `deleteColumn(col: Integer);`

The **deleteColumn** method deletes the column specified in the **col** parameter from the current sheet of a **Table** control.

This deletion may affect other **Table** control properties; for example, the **column** or **leftColumn** property.

The method in the following example shows the use of the **deleteColumn** method.

```
buttonDelColumn_click(btn: Button input);
begin
    if selectedColumn <> null then
        table1.deleteColumn(selectedColumn);
        table1.clearAllSelected;
        selectedColumn := null;
    else
        app.msgBox("You must select a column", "No column selected",
                   MsgBox_OK_Only);
        return;
    endif;
end;
```

deleteRow

Signature `deleteRow(row: Integer);`

The **deleteRow** method deletes the row specified in the **row** parameter from the current sheet of a **Table** control.

This deletion may affect other table control properties; for example, the **row** property.

The code fragment in the following example shows the use of the **deleteRow** property.

```
tbl.column  := 1;
row         := 1;
while row <= tbl.rows do
    tbl.row := row;
    if tbl.text = "" then
        tbl.deleteRow(row);      // Remove empty rows.
        row := row - 1;          // Move back one row so we don't work
    endif;                       // on the next row.
    row     := row + 1;
endwhile;
```

deleteSheet

Signature `deleteSheet(sheet: Integer);`

The **deleteSheet** method deletes the sheet specified in the **sheet** parameter from the **Table** control. This deletion may affect the **topSheet** property.

Note You cannot delete the last sheet of a table control.

discard

Signature `discard();`

The **discard** method of the **OleControl** class closes the OLE object presentation and deletes any stored data.

displayCollection

Signature

```
displayCollection(c:          Collection;     (Table)
                  update:     Boolean;
                  showHow:    Integer;
                  startObj:   Object);

displayCollection(c:          Collection;     (ComboBox, ListBox)
                  update:     Boolean;
                  showHow:    Integer;
                  startObj:   Object;
                  extraEntry: String);
```

The **displayCollection** method enables a collection to be attached to the current sheet of a **Table** control, to the list portion of a **ComboBox** control, or to a **ListBox** control.

The **displayCollection** method parameters are listed in the following table.

Parameter	Description
c	Specifies the attached collection and can be any type of object collection. The scrolling and positioning of non-object collections cannot be handled, and calling **displayCollection** with such collections is rejected. (The **topRow** property is set to the first non-fixed row when using attached collections.)
update	If **true**, changes to the collection are reflected in the control. (This parameter does not apply to transient objects.)
showHow	Specifies an effective bit mask, which defines whether the collection is accessed forward or reversed or whether the user can scroll prior to the start object. (If the start object is **null**, this option has no effect.)
startObj	Specifies the object in the collection from which to start. If this parameter is not specified (that is, it is null), it starts at the beginning (or end) of the collection.
extraEntry	For **ComboBox** and **ListBox** controls only, determines whether the displayed list includes an extra entry.

When the **displayCollection** method is called, the number of rows in that sheet or list is set to the number of fixed rows defined for that sheet or list. Each collection entry adds a non-fixed row to the sheet or list, unless the **displayRow** event method indicates that an entry is to be ignored. The definition of the fixed rows of the sheet or list remains untouched.

When this method is called, the sheet or list contains the number of rows required to fully fill the visible portion of the table or list. (The actual number could be slightly more, to cater for the scroll bar being present.) As the entries are scrolled, the entries no longer visible are removed from the table or list and additional entries are obtained from the collection, as required. The initial entries in the table or list are displayed automatically when the **displayCollection** method is called, causing the table or list to access any collection entries required to fill the display size.

Note When the selected item is scrolled out of view, it is deselected. This deselection behavior can be avoided for list boxes and combo boxes by using the **listCollection** method instead of **displayCollection**.

You can drag the scroll bar thumb of **ListBox** and **Table** controls when a collection is attached to the list or table using the **displayCollection** method and the collection is an **Array**, **Dictionary**, or **Set**. The **Collection** class **indexNear** method determines the approximate position of a displayed object and the **Iterator** class **startNearIndex** method positions the displayed entries.

Implementing this feature has the following side effects.

- If the **bcontinue** parameter returns **false**, the list size is adjusted and remembered. However, users can drag the list past that entry without the **displayRow** event being called for that entry, and as a result, JADE is unaware that the display size was to be limited. The **bcontinue** parameter therefore cannot be relied on to limit the display up to the required point.
- The thumb track reflects the relative position in the collection from the start point (forwards or backwards, as required), regardless of whether entries have been left out of the display.
- The use of the **startNearIndex** method is a shortcut method of approximating the required position through the collection based on the estimated size of segments of the collections (for example, Btree segments). However, if the collection population is distorted, the estimation may result in strange behavior, where dragging the thumb could result in the same position being shown or even going the wrong way through the collection.

Note For both **ListBox** and **Table** controls, Ctrl+Home positions the display to the first entry in the collection list and Ctrl+End displays the last entry in the collection list.

Use the **getCollection** method to return the collection attached to the control or the **clear** method to detach a collection from a control and clear the contents of the control.

The entries in the table or list can be accessed from logic, but the content of a table or list is treated as though it is the complete set of data. Access to rows that are not displayed is therefore not available. Attempting to call the **displayCollection** method in a **ListBox** or **Table** control with a virtual collection (for example, **myClass.instances**) is rejected and an exception is raised. Virtual collections do not implement the methods required by the **displayCollection** method. The **ListBox** class **listCollection** method handles a virtual collection, but only in the forward direction.

Notes As the table holds only the displayed entries, no sorting can be performed. Any defined sort columns are ignored.

The **displayCollection** method and the related **displayRow** method should not be used in a Web-enabled JADE forms application because they provide a small *virtual window* over the underlying collection of potentially thousands of objects. The HTML generated contains only the entries required to fill the virtual window and (unlike a GUI application) does *not* provide a scroll bar to access additional entries.

For a Web-enabled JADE forms application, use the **addItem** method to populate the **ComboBox**, **ListBox**, or **Table** control with collection entries to be displayed.

When the table is at the bottom of the collection and there are no more rows to display, pressing the Page Down key selects the last row in the table. Similarly, the Page Up key selects the first row in the table when at the top of the collection. When paging down, the last entry is displayed on the next page only if the row is not fully displayed in the table.

Use the **ComboBox**, **ListBox**, or **Table** class constants listed in the following table in the **showHow** parameter to control access to a table or list collection.

Class Constant	Integer Value	Description
DisplayCollection_Forward	0	Display the collection forwards
DisplayCollection_Reversed	1	Display the collection reversed
DisplayCollection_NoPrior	0	No access to collection entries prior to start object
DisplayCollection_Prior	2	Allow access to entries prior to start object

To specify multiple collection access options, combine the options by adding them, as shown in the following code fragment.

```
table1.displayCollection(myColl, true, DisplayCollection_Forward +
                         DisplayCollection_Prior, startObj);
```

As a combo box is scrolled, entries that are not visible are removed from the list. If the currently selected item is removed, the **listIndex** property is set to **-1** and the combo box text is cleared. The **findObject** method searches only the entries that are currently loaded when the **displayCollection** method is called.

If the value of the **extraEntry** parameter for a **ComboBox** or **ListBox** control is null (**""**), the parameter is ignored and the displayed list consists only of collection entries. If the **extraEntry** parameter contains a non-null string, the displayed list includes an extra entry, using the specified string.

The entry is treated as though it is the first entry in the collection (or the last entry, when the collection is displayed in reverse order). The exception to this is when a start object (specified in the **startObj** parameter) is requested and the **showHow** parameter value does not include **DisplayCollection_Prior**. In this case, the extra entry is treated as though it is the starting object, followed by the real starting object in the list. The extra object has a null row object associated with it in the list box and can be recognized by using that null row object (for example, **if list.itemObject[list.listIndex] = null then** ...).

The code fragment in the following example shows the use of the **extraEntry** parameter.

```
listbox1.displayCollection(custlist, true, 0, null, "<Default>");
```

If the object is already displayed in a **ComboBox** or **ListBox** list, the **listIndex** property value is set to the existing list entry. If the object is not in the displayed list, the current list is discarded. The iterator position within the collection is then adjusted so that the rebuilt display list includes the requested object. The **listIndex** property is then set to the list entry of the requested object. If the object cannot be found within the collection, an exception is raised.

Note If a collection is attached to a combo box or list box by using the **listCollection** method and the object is not in the displayed list, additional entries from the collection are added until that entry is included.

In addition, when the **displayCollection** method of the **ComboBox** or **ListBox** class is called with the **extraEntry** parameter set to a string value and the **listObject** property is set to null, the value of the **listIndex** property is set to the **extraEntry** list entry value (which is always **1**). If that entry is not in the list that is currently displayed, the current list is discarded, restarted from the beginning of the collection entries, and includes the extra entry.

Note Setting the **listIndex** property to **-1** remains unchanged; that is, no entry is selected.

When a collection is associated with a sheet of a control, the following restrictions apply.

- The **addItem**, **addItemAt**, **removeItem**, **Table**::**resort**, **Table**::**moveRow**, and **Table**::**deleteRow** methods are not available.

  As the scrolling of the control relies on the object associated with each row, the scrolling principle would break down with the use of these methods. In addition, the number of rows cannot be changed (although the number of fixed rows can be).

- The **clear**, **displayCollection**, and **Table**::**deleteSheet** methods cannot be called for a sheet or list from the **displayRow** event method for that sheet or list.

- The **itemObject** property for a row is set to the collection object for that row. This value cannot be changed.

Note If prior access to the start object is allowed and there are insufficient entries in the collection after the start object to fill the control, prior entries are inserted at the top of the control (that is, the requested start object may not be the first entry in the control).

The processing of the **displayCollection** method functions as follows.

1. Logic attaches the collection to the control by using the **displayCollection** method, as shown in the method in the following example that associates a collection with a table called **ListProducts**.

   ```
   load() updating;
   vars
       company : Company;
   begin
       company := Company.firstInstance;
       tableProducts.displayCollection(company.allProducts, true, 0, null);
   end;
   ```

2. Any existing **displayCollection** for that control is discarded.
3. The number of rows for that control is set to the number of fixed rows for a **Table** control or to zero (**0**) for a **ComboBox** or **ListBox** control.
4. If a starting object is specified in the **startObj** parameter, the collection entries are extracted, starting with the specified entry.
5. The **displayRow** event method is called for each entry in the collection, as required, and only the number of entries that are need to fill the table or list are accessed.
6. When the control is scrolled, non-visible entries are discarded and additional entries are obtained by using the **displayRow** event method.
7. This process continued until the end of the collection is reached or the **displayRow** event method indicates that the entries in the collection beyond this point are to be ignored.

dockMdi

Signature
```
dockMdi();
```

The **dockMdi** method of the **Form** class docks a floating MDI child form back into its MDI frame. This method does nothing if the form is not floating or if the form is not an MDI child.

When the MDI child form is docked, the position and size of the form is restored to its values when it was floated if the current top-most MDI child form in the MDI frame is not maximized. If the current top-most MDI child form in the MDI frame is maximized, the docked form is also maximized.

Applies to Version: 2020.0.01 and higher

doLinemarker

Signature
```
doLinemarker(action: Integer;
             param1: Integer;
             param2: Integer): Integer;
```

The **doLinemarker** method of the **JadeTextEdit** class performs linemark-related actions.

The values of the **action**, **param1**, and **param2** parameters and the return values are listed in the following table. (The **action** parameter values are represented by **JadeTextEdit** class constants.)

action	param1	param2	Returns…	Description
LMACT_ADD (1)	Line #	Marker #	Handle #	Adds the linemark number specified in **param2** to the line specified in **param1**. If **param1** is zero (**0**), the **currentLine** property value is used. It returns the handle of the new linemark if successful, else **-1** if the add action failed.

action	param1	param2	Returns…	Description
LMACT_DELETE (2)	Line #	Marker #	N/A	Removes the linemark number specified in **param2** from the line specified in **param1** (if present). If **param1** is zero (**0**), the **currentLine** property value is used. Removes all markers on the specified line deleted if **-1** specified in **param2**.
LMACT_DELETEALL (3)	Line #	N/A	N/A	Removes the linemark number specified in **param1** (if present) from all lines. If **param1** is **-1**, all linemarks are deleted from all lines.
LMACT_DELETEBYHANDLE (4)	Handle #	N/A	N/A	Removes the linemark associated with the specified linemark handle, if it exists.
LMACT_GETBITMASK (5)	Line #	N/A	Bit mask	Returns a bit mask that represents the linemarks currently set on the line specified in **param1**.
LMACT_GETLINEFROMHANDLE (6)	Handle #	N/A	Line #	Returns the line number of the line that has the linemark associated with the linemark handle specified in **param1**. Returns zero (**0**) if handle is not found.
LMACT_GONEXTBYNUMBER (7)	Line #	Marker #	New line #	Moves the caret to the start of the next line following the one specified in **param1** that has a linemark with the number specified in **param2**. If **param1** is zero (**0**), the search starts after the value of the **currentLine** property. It returns the new line number if found, else zero (**0**).

action	param1	param2	Returns...	Description
LMACT_GOPRIORBYNUMBER (8)	Line #	Marker #	New line #	Moves the caret to the start of the first line preceding the one specified in **param1** that has a linemark with the number specified in **param2**. If **param1** is zero (**0**), then the search starts before the value of the **currentLine** property. If **param1** is greater than the value of the **lineCount** property, the search starts from the last line. It returns the new line number if found, else zero (**0**).
LMACT_GONEXTINBITMASK (9)	Line #	Bit mask	New line #	Moves the caret to the start of the next line following the one specified in **param1** that has a linemark with its number matching any bit in the bit mask specified in **param2**. If **param1** is zero (**0**), the search starts after the value of the **currentLine** property. It returns the new line number if found, else zero (**0**).
LMACT_GOPRIORINBITMASK (10)	Line #	Bit mask	New line #	Moves the caret to the start of the first line preceding the one specified in **param1** that has a linemark with its number matching any bit in the bit mask specified in **param2**. If **param1** is zero (**0**), the search starts before the value of the **currentLine** property. If **param1** is greater than the value of the **lineCount** property, the search starts from the last line. It returns the new line number if found, else zero (**0**).

action	param1	param2	Returns…	Description
LMACT_GOHANDLE (11)	Handle #	N/A	New line #	Moves the caret to the start of the line that has the linemark associated with the handle specified in **param1**. It returns the new line number if found, else zero (**0**).
LMACT_MOVESINGLETOLINE (12)	Line #	Marker #	Handle #	Deletes all occurrences of the linemark number specified in **param2** then adds a linemark with the number specified in **param2** to the line specified in **param1**, making the line visible. It returns the handle of the new linemark if successful, else **-1**.
LMACT_MOVESINGLETOPOSITION (13)	Offset #	Marker #	Handle #	Deletes all occurrences of the linemark number specified in **param2** then adds a linemark with the number specified in **param2** to the line that contains the character at the offset specified in **param1**, making that line visible. It returns the handle of the new linemark if successful, else **-1**.
LMACT_ADDATPOSITION (14)	Offset #	Marker #	Handle #	Adds linemark number specified in **param2** to the line that contains the character at offset specified in **param1**. It returns the handle of the new linemark if successful, else **-1**.

In this table, the first three columns represent the method parameters (that is, **action**, **param1**, and **param2**, respectively). The add, delete, and get actions do not move the caret position.

The values in the second, third, and fourth columns of the preceding table (that is, values in the middle three columns) are described in the following table.

Value	Description
Line #	Line number in the range **1** through the number of the last line in the text editor. A line number value of zero (**0**) indicates the current line.
Marker #	Marker number in the range zero (**0**) through **31**. However, as folding uses marker numbers in the range **25** through **31**, you should not use these values.
Bit mask	Integer representing marker numbers.
Handle #	Unique integer value assigned when each marker is added.
New line #	Line number to which the caret is moved, and zero (**0**) if there is no match.

The code fragment in the following example shows the use of the **doLinemarker** method. It deletes all linemarks with number **1**, adds a linemark number **1** to the current line, and saves its handle. It adds a linemark number **2** to every line that contains the text "**theWord**", moving the caret (and view) to the first linemark number **2** following the current line. It moves the caret (and view) back to the starting line.

```
handl:=jteSource.doLinemarker(JadeTextEdit.LMACT_MOVESINGLETOPOSITION,0,1);
jteSource.findMarkAll( "theWord", 0, 1, false, false, 0, 2, true);
jteSource.doLinemarker(JadeTextEdit.LMACT_GONEXTBYNUMBER, 0, 2);
jteSource.doLinemarker(JadeTextEdit.LMACT_GOHANDLE, handle, 0);
```

doWindowEvents

Signature `doWindowEvents(waitTime: Integer);`

The **doWindowEvents** method of the **Window** class processes all pending Windows events for this window and all of its children.

The **doWindowEvents** method for a control processes events generated by the control and its children. In addition, it processes copies of the **gotFocus**, **lostFocus**, **keyDown**, **keyUp**, and **keyPress** events on the control, which are also sent to the form.

When the **doWindowEvents** method has processed all pending Windows events it then waits, and processes any further Windows events that arrive for this window until the specified time from when the method was initiated has expired.

Use the **waitTime** parameter to specify in milliseconds the required interval this method is to wait.

Note If a parent of the window requires repainting, the **doWindowEvents** method will not cause the window to be repainted.

The most common use of this method is to enable a **Cancel** button to be monitored during lengthy processing. (See also the **Application** class **doWindowEvents** method.)

The methods in the following examples show the use of the **doWindowEvents** method.

```
btnCancel_click(btn: Button);
begin
    cancelWasClicked := true;
end;
```

```
    longProcess() updating;
    begin
        cancelWasClicked := false;
        while true do
            ...                             // some kind of processing
            btnCancel.doWindowEvents(0);    // allow button press
            if cancelWasClicked then
                ...                         // cancel processing
                break;
            endif;
        endwhile;
    end;
```

Do *not* use the **doWindowEvents** method in the following situations.

- When causing a repaint of a window. Use the **refreshNow** method to repaint a window.
- When involved in the processing of ActiveX controls and OLE objects. As the OLE control processes requests synchronously using Window events, the **Window** class **doWindowEvents** method can cause asynchronous processing to be attempted, with resulting failure.

Call the **Application** class **doWindowEvents** method from a server method to process server notifications and timers.

Caution Indiscriminate use of this method can cause unwanted side effects. For example, it can change the order of Windows event processing and can allow users to click on other controls, menus, or forms that could have an impact on the current process.

It can also cause recursive loops. For example, if a **keyDown** event calls a **doWindowEvents** method and the user is holding down the key, that method will invoke another **keyDown** event, and so on. JADE handles this situation, by discarding messages for a specific window if there are already five such messages in the call stack.

dragColumn

Signature `dragColumn(): Integer;`

The **dragColumn** method of the **Table** class provides table-specific location information of the drag and drop processes that correspond to the *x* (horizontal) and *y* (vertical) positions received from **dragOver** events during a drag operation. Similarly, when the dragged window is dropped, the window that it is dropped onto receives a **dragDrop** event.

The **dragOver** and **dragDrop** events specify the *x* and *y* location of the drag operation.

Note If the drag and drop process occurs over an empty part of the table, one of the **row** or **column** property values may still be a non-zero value, indicating that it is in a specific row or column.

The method in the following example shows the use of the **dragColumn** method.

```
    table1_dragDrop(table: Table input;
                    win:   Window input;
                    x, y:  Real) updating;
    begin
        if selectedColumn <> null then
            table.moveColumn(selectedColumn, table1.dragColumn);
        endif;
    end;
```

dragListIndex

Signature `dragListIndex(): Integer;`

The **dragListIndex** method of the **ListBox** class provides the **listIndex** property of a list box entry that corresponds to the *x* (horizontal) and *y* (vertical) positions received from **dragOver** events during the drag operation. Similarly, when the dragged window is dropped, the window that it is dropped onto receives a **dragDrop** event.

The **dragOver** and **dragDrop** events specify the *x* and *y* location of the drag operation.

The **dragListIndex** method value is **-1** if there is no list box entry at that position.

The code fragment in the following example shows the use of the **dragListIndex** method.

```
targetIndex.caption := listBox1.dragListIndex.String;
```

dragRow

Signature `dragRow(): Integer;`

The **dragRow** method of the **Table** class provides table-specific location information of the drag and drop processes that corresponds to that *x* (horizontal) and *y* (vertical) positions received from **dragOver** events during a drag operation. Similarly, when the dragged window is dropped, the window that it is dropped onto receives a **dragDrop** event.

The **dragOver** and **dragDrop** events specify the *x* and *y* location of the drag operation.

Note If the drag and drop process occurs over an empty part of the table, one of the **row** or **column** property values may still be a non-zero value, indicating that it is in a specific row or column.

The method in the following example shows the use of the **dragRow** method.

```
dragDrop(table: Table input; win: Window input; x: Real; y: Real) updating;
begin
    inheritMethod(table, win, x, y);
    if dragRow > 1 and dragColumn > 0 then
        row := dragRow;
        column := dragColumn;
        if text <> "" then
            calendar.changeType := calendar.ChangeType_Day;
            calendar.date.setDate(text.Integer, calendar.date.month,
                                  calendar.date.year);
        endif;
    endif;
end;
```

dragSheet

Signature `dragSheet(): Integer;` (Table)

`dragSheet(): Sheet;` (Folder)

The **dragSheet** method of the **Table** class or **Folder** class provides respective table-specific or sheet-specific location information of the drag and drop processes that corresponds to the *x* (horizontal) and *y* (vertical) positions received from **dragOver** events during a drag operation.

Similarly, when the dragged window is dropped, the window that it is dropped onto receives a **dragDrop** event.

The **dragOver** and **dragDrop** events specify the *x* and *y* location of the drag operation.

Use the **dragSheet** method in situations where the drag and drop process occurs over the tab area of a multiple sheet table or folder. In this situation for a table, the **dragColumn** and **dragRow** method values are set to zero (**0**).

Note It is possible for the returned value of the sheet for the **Folder** class tab to be null, which means that the **dragOver** or **dragDrop** event is over the folder but the position is not over a tab.

The value returned by the **dragSheet** method has meaning only when called from within the **dragOver** or **dragDrop** event of a folder. Calling the method under any other circumstance returns the value that was last set during a prior drag operation or it returns null if it has never been set.

eject

Signature `eject();`

The **eject** method of the **MultiMedia** class causes the device to eject its medium.

The code fragment in the following example shows the use of the **eject** method.

```
if cd.canEject then
    cd.stop;
    cd.eject;
endif
```

If the **useDotNetVersion** property is set to **true**, the **eject** method is not available and it generates exception 1068 (*Feature not available in this release*).

See also the **canEject** method.

embedFromClass

Signature `embedFromClass(class: String);`

The **embedFromClass** method of the **OleControl** class initiates the specified server application to create or attach a new OLE object.

The class names are listed in the registry of your operating system, and are displayed in the second column of the **Embed from New** list of the Insert dialog. For more details, see the **showInsertForm** method.

embedFromFile

Signature `embedFromFile(filename: String);`

The **embedFromFile** method of the **OleControl** class embeds an OLE object from the specified file into a control. The file specified in the **filename** parameter must be registered as belonging to a registered OLE server."

emptyUndoBuffer

Signature `emptyUndoBuffer(): Integer;`

The **emptyUndoBuffer** method of the **JadeTextEdit** class clears the editor and discards any undo or redo history. This method returns zero (**0**).

The value of the **modified** property is also set to **false**. The next text editor action causes the **firstChange** event method.

enableEvent

Signature
```
enableEvent(name:   String;
            enable: Boolean): Boolean;
```

The **enableEvent** method of the **Window** class enables you to control at run time whether JADE logic associated with an event for a specific form or control is executed. You could use this method in thin client mode, for example, to speed up the data entry process for a **TextBox** control by disabling the **keyDown** event. The **enableEvent** method returns the previous state of the event (that is, it returns **true** if the event was enabled or **false** if it was disabled).

Use the **name** parameter to specify the name of the event that is to be disabled. Set the **enable** parameter to **false** if you want to disable the event specified in the name parameter. All events are enabled by default; that is, the **enable** parameter is set to **true**.

An exception is raised if the event name specified in the **name** parameter is not valid. Although a check of the event name is performed, no check is made to ensure that the event belongs to the window of the receiver.

Notes Enabling or disabling an event has no impact if there is no logic associated with that event.

Event methods can be enabled or disabled in both standard (fat) client mode and in thin client mode.

For a **Control** class, calling an event method results in a call on that method of the control (for example, the **keyDown** method of the **TextBox** class) and then a call on the specific instance of the form of that control method (for example, **textbox1_KeyDown**).

Disabling an event method results in neither method being called.

The method in the following example shows the use of the **enableEvent** method.

```
load();
begin
    tbDesc.enableEvent("keyDown", false);
    self.enableEvent("mouseMove", false);
end;
```

The **Window** class **isEventEnabled** method returns whether a specified event is currently enabled.

endNotifyAutomationEvent

Signature
```
endNotifyAutomationEvent(receiver:          Object;
                         eventClassRefName: String);
```

The **endNotifyAutomationEvent** method of the **ActiveXControl** class terminates a previous **beginNotifyAutomationEvent** method.

The parameters for this method, listed in the following table, must be the same as the parameters specified in the **beginNotifyAutomationEvent** method.

Parameter	Description
receiver	The object that is to receive the event notification

Parameter	Description
eventClassRefName	The name of the reference (an instance of the **IDispatch** subclass) that implements the notification events

ensureCaptionIsVisible

Signature `ensureCaptionIsVisible(): Boolean;`

The **ensureCaptionIsVisible** method of the **Form** class returns **true** if it is necessary to move the form so that the caption (or the top portion of a form with no caption) is visible on at least one monitor. If you have multiple monitors in various configurations, saved or predetermined positions can leave a form inaccessible.

The **ensureCaptionIsVisible** method does nothing if the **isCaptionVisible** method would return **true**. The **ensureCaptionIsVisible** method is not affected by the visibility of the form or the **zOrder** method. It does not make the form visible, but repositions it if required.

For a non-MDI form, the form is repositioned at the top left of the work area of the monitor displaying the form, or the primary monitor if the form is not over any monitor. For an MDI form, the form is repositioned at the top left of the client area of the MDI frame.

eventItemName

Signature `eventItemName(): String;`

The **eventItemName** method of the **JadeXamlControl** class returns the name of the element of the XAML control for which a standard JADE event has occurred. This method can be used in logic to determine the element of the XAML content that issued the event.

In thin client mode, it does not cause a message to be sent to the presentation client.

In the following example, the XAML content for a control defines two buttons.

```
jadeXamlCtl.xaml :=
  '<Canvas Name="SimpleExample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Button Canvas.Top="20" Canvas.Left="50" Content="Hello World" />
    <Button Name="Hi" Canvas.Top="20" Canvas.Left="50" Content="Hello" />
    <Button Name="Bye" Canvas.Top="60" Canvas.Left="50" Content="Bye" />
  </Canvas>';
```

The **eventItemName** method returns the name of the button that was clicked, as shown in the following example.

```
jadeXamlCtl_click(jadeXamlControl: JadeXamlControl input) updating;
begin
  write jadeXamlCtl.eventItemName; // Outputs "Hi", "Bye", or "jadeXamlCtl"
end;
```

find

Signature

```
find(text:    String;          (JadeRichText)
     start:   Integer;
     finish:  Integer;
     options: Integer): Integer;

find(match:          String;    (JadeTextEdit)
     range:          Integer;
     direction:      Integer;
     caseSensitive:  Boolean;
     wholeWord:      Boolean;
     wordStart:      Boolean;
     interpretation: Integer): Integer;
```

The **find** method returns the character position at which the text was found. If no match was found, this method returns **-1**.

The **find** method of the **JadeRichText** class searches for text within the contents of the control. The **find** method parameters for rich text controls are listed in the following table.

Parameter	Description
text	Text to be located.
start	Start of the search range, as a character index into the text or specified as **Find_BeginningOfText**.
finish	End of the search range, as a character index into the text or specified as **Find_EndOfText**.
options	One or more of the following values, separated by the plus symbol (**+**). ■ **Find_MatchCase**, which finds only text with the matching case; otherwise search is case-insensitive. ■ **Find_WholeWord**, which finds only whole words; otherwise parts of words satisfy the search. ■ **Find_SearchBack**, which searches backwards through the text; otherwise the search direction is forward to the end of the text. ■ **Find_Default**, when no other option is specified. You cannot use this value in any combination with any other option value.

The **find** method of the **JadeTextEdit** class searches for text within the contents of the text editor. The **find** method parameters for text edit controls are listed in the following table.

Parameter	Description
match	Mandatory value, which specifies the text to be located.
range	Range of search. One of **FIND_RANGE_ALL** (0), **FIND_RANGE_CARET** (1), or **FIND_RANGE_SELECTION** (2).
direction	Direction in which to search. Specify **-1** to search backwards towards the top of the text, zero (**0**) or **+1** to search forwards towards the bottom of the text.
caseSensitive	If **true**, finds only text with the matching case. If **false**, the search is case-insensitive.

Parameter	Description
wholeWord	If **true**, finds only whole words. If **false**, finds parts of words that satisfy the search.
wordStart	If **true**, the match must occur at the start of a word (that is, the matched text must be preceded by a non-word character).
interpretation	One of the following values, represented by **JadeTextEdit** class constants. ■ **FIND_INTERP_NONE** (0), to search for text as is (the default value) ■ **FIND_INTERP_POSIXREGEXPR** (3), to search for a POSIX regular expression Same as **FIND_INTERP_REGEXPR** except that the \ and \ sequences become more POSIX-compatible unslashed **(** and **)** sequences. ■ **FIND_INTERP_REGEXPR** (2), to search for a regular expression The search is performed using the match text a "regular expression". For a list of characters that have special meaning within a regular expression, see the following table. ■ **FIND_INTERP_UNSLASH** (1), to search for backslash control characters Before the search begins, the match text has the following backslash escape sequences replaced by the appropriate single character: **\a** (bell), **\b** (backspace), **\f** (form feed), **\n** (line feed), **\r** (carriage return), **\t** (tab), **\v** (vertical feed), \\ (backslash), **\OOO** (OOO is three octal digits), **\xHH** (HH is two hexadecimal digits). A backslash preceding an unrecognized escape character is discarded.

For **JadeTextEdit** controls, the matching text is selected in the text editor and the caret is positioned at the end of the matched text (that is, the selection).

Characters that have special meaning within a regular expression for interpretation represented by the **JadeTextEdit** class **FIND_INTERP_REGEXPR** constant are listed in the following table.

Character	Description
.	Matches any character.
\\(	Marks the start of a region for tagging a match.
\\)	Marks the end of a tagged region.
\n	The **n** value in the range **1** through **9** refers to the first through ninth tagged region when replacing. For example, if the search string is **Fred\\([1-9]\\)XXX** and the replace string is **Sam\1YYY**, when applied to **Fred2XXX**, this generates **Sam2YYY**.
\\<	Matches the start of a word using Scintilla's definitions of words.
\\>	Matches the end of a word using Scintilla's definition of words.
\x	Enables you to use a character **x** that would otherwise have a special meaning. For example, **\\[** is interpreted as **[** and not as the start of a character set, and \\ means a single backslash character.
[...]	Indicates a set of characters; for example, **[abc]** means any of the characters **a**, **b**, or **c**. You can also use ranges; for example, **[a-z]** for any lowercase character.
[^...]	Complement of the characters in the set. For example, **[^A-Za-z]** means any character except an alpha character.
^	Matches the start of a line (unless used inside a set, as stated for **[^...]** in the previous row).

Character	Description
$	Matches the end of a line.
*	Matches zero or more times. For example, **Sa*m** matches **Sm**, **Sam**, **Saam**, **Saaam**, and so on.
+	Matches one or more times. For example, **Sa+m** matches **Sam**, **Saam**, **Saaam**, and so on.

A regular expression can also contain the escape sequences listed for **FIND_INTERP_UNSLASH,** except for **\xHH**. The **find** method returns **-1** if the match text is an invalid regular expression (for example, an unbalanced **[**).

findAgain

Signature `findAgain(direction: Integer): Integer;`

The **findAgain** method of the **JadeTextEdit** class searches for the match text and parameter values reused from the most recent **find**, **findMarkAll**, or **replaceAll** method.

The **findAgain** method continues searching in the specified direction, skipping the selected text. In the **direction** parameter, specify **-1** to search backwards towards the top of the text editor or specify zero (**0**) or **+1** to search forwards towards the bottom of the text editor.

The matching text is selected in the text editor, the caret is positioned at the end of the matched text (that is, the selection), and the character position at which the text was found is returned. If no match was found, this method returns **-1**.

The method in the following example shows the use of the **findAgain** method.

```
mnuEditFindNext_click(menuItem: MenuItem input) updating;
vars
    findpos : Integer;
begin
    findpos := jteSource.findAgain(0);
    if findpos < 0 then
        app.msgBox(" No matching text", "Find Next",0);
    endif;
end;
```

findMarkAll

Signature

```
findMarkAll(match:          String;
            range:          Integer;
            direction:      Integer;
            caseSensitive:  Boolean;
            wholeWord:      Boolean;
            wordStart:      Boolean;
            interpretation: Integer;
            markNumber:     Integer;
            clearMarks:     Boolean): Integer;
```

The **findMarkAll** method of the **JadeTextEdit** class searches for the match text within the text editor, using the parameters specified in the following table.

The specified linemark is set on each line where the match text is found.

Parameter	Description
match	Mandatory value, which specifies the text to be located.
range	Range of search. One of **FIND_RANGE_ALL** (0), **FIND_RANGE_CARET** (1), or **FIND_RANGE_SELECTION** (2).
direction	Direction in which to search. Specify **-1** to search backwards towards the top of the text editor, zero (**0**) or **+1** to search forwards towards the bottom of the text editor.
caseSensitive	If **true**, finds only text with the matching case. If **false** (the default), the search is case-insensitive.
wholeWord	If **true**, finds only whole words. If **false** (the default), finds parts of words that satisfy the search.
wordStart	If **true**, the match must occur at the start of a word (that is, the matched text must be preceded by a non-word character).
interpretation	One of the following values, represented by **JadeTextEdit** class constants. ■ **FIND_INTERP_NONE** (0), to search for match text as is ■ **FIND_INTERP_POSIXREGEXPR** (3), to search with match as a POSIX regular expression ■ **FIND_INTERP_REGEXPR** (2), to search with match as a regular expression ■ **FIND_INTERP_UNSLASH** (1), to search with match containing backslash control characters For more details about interpretation, see the **find** method.
markNumber	Mark number set on each line in which the match text is found.
clearMarks	If **true**, all instances of the linemark **markNumber** parameter value are removed from the text editor before the find action is performed. If **false**, linemarks are not cleared.

You can use the **doLinemarker** method to navigate between linemarks.

The **findMarkAll** method returns the number of matches. If no match was found, this method returns zero (**0**).

The code fragment in the following example shows the use of the **findMarkAll** method to place linemark 1 on all lines that contain the characters **"fred"**.

```
count := self.theJadeTextEdit.findMarkAll("fred", 0 /*range*/,
                                          1 /*direction*/,
                                          false /*case-insensitive*/,
                                          false /*wholeWord*/,
                                          false /*wordStart*/,
                                          0 /*interpretation*/, 1, true);
```

findObject

Signature `findObject(object: Object): Integer;`

The **findObject** method of the **ListBox** and **ComboBox** classes searches the **itemObject** property values of the list entries of a list box or combo box control for the object specified in the **object** parameter.

If the specified object is found, its **listIndex** value is returned. If the specified object is not found, **-1** is returned.

Note The **findObject** method searches only the entries that are currently loaded (which are the visible entries only) when the **displayCollection** method is used to populate the control.

The code fragment in the following example shows the use of the **findObject** method.

```
cmb_Category.listIndex := cmb_Category.findObject(myCustomer.myCategory);
```

The **findObject** method does the same as the logic in the following code fragment.

```
foreach indx in 1 to list1.listCount do
    if list1.itemObject[indx] = obj then
        return indx;
    endif;
endforeach;
return -1;
```

findReplaceDialog

Signature

```
findReplaceDialog(findOnly: Boolean;
                  left:     Real;,
                  top:      Real):Form;
```

The **findReplaceDialog** method of the **JadeRichText** control class opens a dialog so that a search can be carried out over the text in the control.

If the value specified in the **findOnly** parameter is **false**, the dialog also displays a text box enabling replacement text to be entered. The values specified for the **left** and **top** parameters determine the position of the dialog.

Note If the control already has a Find and Replace dialog associated with it, the dialog is shown at the current position.

findString

Signature

```
findString(startIndx: Integer;
           str:       String): Integer;
```

The **findString** method of the **ListBox** class or **ComboBox** class searches the entries in a list box or combo box control for an entry with the string specified in the **str** parameter.

The search is not case-dependent. The **findString** method matches any entry with the specified string prefix.

The search starts with the value specified in the **startIndx** parameter and returns the next found entry in the list. If no matching entry is located, it returns **-1**.

The method in the following example shows the use of the **findString** method.

```
textBoxLeftStart_change(textbox: TextBox input) updating;
vars
    count : Integer;
begin
    count := listBoxLeft.findString(1, textBoxLeftStart.text);
    if count <> -1 then
        listBoxLeft.topIndex := count;
    endif;
end;
```

findStringCaseSensitive

Signature `findStringCaseSensitive(startIndx: Integer;`
```
                        str:       String): Integer;
```

The **findStringCaseSensitive** method of the **ListBox** class or **ComboBox** class searches the entries in a list box or combo box control for an entry with the case-sensitive string specified in the **str** parameter.

The **findStringCaseSensitive** method matches any entry with the specified case-sensitive string prefix.

The search starts with the value specified in the **startIndx** parameter and returns the next found case-sensitive entry in the list. If no matching entry is located, it returns **-1**.

The method in the following example shows the use of the **findStringCaseSensitive** method.

```
textBoxLeftStart_change(textbox: TextBox input) updating;
vars
    count : Integer;
begin
    count := listBoxLeft.findStringCaseSensitive(1, textBoxLeftStart.Text);
    if count <> -1 then
        listBoxLeft.topIndex := count;
    endif;
end;
```

Applies to Version: 2016.0.02 (Service Pack 1) and higher

findStringExact

Signature `findStringExact(startIndx: Integer;`
```
                str:       String): Integer;
```

The **findStringExact** method of the **ListBox** class or **ComboBox** class searches the entries in a list box or combo box control for an entry with the string specified in the **str** parameter.

The search is not case-dependent. The **findStringExact** method matches only entries where the strings are an exact match.

The search starts with the value specified in the **startIndx** parameter and returns the next found entry in the list. If no matching entry is located, it returns **-1**.

The method in the following example shows the use of the **findStringExact** method.

```
vars
    custs : CustomersByContactNameDict;
    cust  : Customers;
begin
    app.mousePointer := MousePointer_HourGlass;
    comboBoxCity.addItem("[All Cities]");
    create custs;
    foreach cust in custs do
        listBoxCustomers.addItem (cust.contactName & ", " & cust.address);
        if comboBoxCity.findStringExact (1, cust.city) = -1 then
            comboBoxCity.addItem (cust.city);
        endif;
    endforeach;
epilog
```

```
        app.mousePointer := MousePointer_Default;
    end;
```

findStringExactCaseSensitive

Signature
```
findStringExactCaseSensitive(startIndx: Integer;
                             str:       String): Integer;
```

The **findStringExactCaseSensitive** method of the **ListBox** class or **ComboBox** class searches the entries in a list box or combo box control for an entry with the case-sensitive string specified in the **str** parameter.

The **findStringExactCaseSensitive** method matches only entries where the strings are an exact case-sensitive match.

The search starts with the value specified in the **startIndx** parameter and returns the next found entry in the list. If no matching entry is located, it returns **-1**.

The method in the following example shows the use of the **findStringExactCaseSensitive** method.

```
    vars
        custs : CustomersByContactNameDict;
        cust  : Customers;
    begin
        app.mousePointer := MousePointer_HourGlass;
        comboBoxCity.addItem("[All Cities]");
        create custs;
        foreach cust in custs do
            listBoxCustomers.addItem (cust.contactName & ", " & cust.address);
            if comboBoxCity.findStringExactCaseSensitive (1, cust.City) = -1 then
                comboBoxCity.addItem (cust.City);
            endif;
        endforeach;
    epilog
        app.mousePointer := MousePointer_Default;
    end;
```

Applies to Version: 2016.0.02 (Service Pack 1) and higher

firstVisibleLine

Signature `firstVisibleLine(): Integer;`

The **firstVisibleLine** method of the **JadeRichText** and **TextBox** control classes returns the first visible line in the rich text control or the text box.

For a **JadeRichText** or **TextBox** control that scrolls vertically, this method returns the line number of the first visible text line displayed in the control (**1**-relative). If the control does not scroll vertically, the return value is always **1**.

flagControlForSave

Signature `flagControlForSave(ctl: Control);`

When the **flagControlForSave** method of the **Form** class is invoked and passed a control that was not painted on the form using Painter functionality, it flags the control specified in the **ctl** parameter as one that is saved by Painter when the form is saved.

Any controls that were not painted on the form by using Painter and are not flagged for saving by specifying them in the **ctl** parameter of this method are not saved when the form is saved.

Notes This method is valid only in Painter. If this method is invoked when the form is not open in Painter, an exception is raised. To check that the form is open, use the **isInPainter** method of the **Control** class, which returns **true** if the control is open in Painter.

If the length of the name of the control is greater than 16 characters, an exception 1035 *(String too long)* is raised.

This process becomes necessary, for example, when a user-defined control adds additional controls to a form using the **addControl** method of the **Form** class as part of its creation process.

If the **flagControlForSave** method is not invoked for these additional controls when a form on which the user-defined control has been painted is saved, only the user-defined control is saved with the form, and not the additional controls.

float

Signature
```
float(lft: Integer;
      tp:  Integer;
      wth: Integer;
      hgt: Integer);
```

The **float** method of the **JadeDockBase** class causes the **JadeDockBar** or **JadeDockContainer** docking control to be floated by creating a floating form at the left, top, width, and height screen positions specified in the **lft**, **tp**, **wth**, and **hgt** parameters, respectively.

If the control is already floating, this method repositions and resizes the floating form. The resulting size of the floating form may vary if the automatic sizing of the dock control or controls requires a different size to the specified width and height. The control is still floated even if the value of the **floatingStyle** property is set to **FloatingStyle_None** (0).

Note The control remains logically attached to the original form and all events generated for the control while it is floating are still passed to the original form. The **Control** class **form** property for the floated control remains set to the original form and the **Control** class **parent** property of the control is set to a null value, as the floating form does not have a JADE object associated with it.

Calling the **float** method does not generate a **JadeDockBar** or **JadeDockContainer** class **floated** event. (For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document. See also the docking examples under the **JadeDockContainer** class **allowDocking** property, earlier in this document.)

floatMdi

Signature `floatMdi();`

The **floatMdi** method of the **Form** class floats an MDI child form; that is, it takes an MDI child form out of the MDI frame and allows it to be moved independently from the MDI frame (for example, on to another monitor on the PC). This method does nothing if the form is not docked or if the form is not an MDI child.

Note When floated, an MDI child form is positioned on the screen in the same position, except it is not a child restricted to the MDI frame.

The MDI child form is always on top of the MDI frame. To reposition the form programmatically, set the **left** and **top** properties, or use the **Window** class **move** method.

Applies to Version: 2020.0.01 and higher

generateHTML

Signature

```
generateHTML(includeDefaultSessionInfo: Boolean;
             useControlNames:           Boolean;
             generateFormDataOnly:      Boolean;
             internetExplorerOnly:      Boolean): String;
```

When the **generateHTML** method of the **Form** class is called on a form object, it generates the HTML string for that form from the string returned from the receiver. You can then perform one of the following actions.

- Manipulate this string before sending it back to the Web browser.
- Save the string as a file, which enables you to obtain a snapshot of the file and then periodically update it, if required (by using notifications, for example).

Tips Use this method to reduce generation and Windows resource overhead in situations where the data seldom changes or it is not necessary to have the most current data available.

When the **generateHTML** method is called to generate an HTML string, the HTML is generated without word wrapping when the **wordWrap** property is set to **false**. Set this property to **true** if you want an HTML string in a table cell generated with word wrapping.

The parameters for the **generateHTML** method are listed in the following table.

Parameter	Set this parameter to true if you want …
includeDefaultSessionInfo	JADE to automatically generate the hidden fields that are sent with the form for session control.
useControlNames	JADE to use control names instead of oids in the HTML generation. Note that control names may be duplicated when there are cloned controls that have not been given another name. If the control was an input field (for example, a text box), the Web browser will return multiple *name-value* pairs with the same names. It is your responsibility to handle this situation.
generateFormDataOnly	JADE to generate only the HTML between the <FORM> and </FORM> tags, inclusive. You can then insert this string in a user template that sets up the header and footer parts of the Web page, for example.
internetExplorerOnly	To generate Internet Explorer 4.0 (and above) specific code.

See also the **Form** class **generateHTMLStatic** method.

generateHTMLStatic

Signature

```
generateHTMLStatic(includeDefaultSessionInfo: Boolean;
                   useControlNames:           Boolean;
                   generateFormDataOnly:      Boolean;
                   internetExplorerOnly:      Boolean;
                   machineName:               String;
                   virtualDirectory:          String;
                   protocol:                  String): String;
```

When the **generateHTMLStatic** method of the **Form** class is called on a form object and values are specified in the **machineName**, **virtualDirectory**, and **protocol** parameters, JADE generates the static HTML string for that form from the string returned from the receiver and builds the full Uniform Resource Locator (URL) action line.

You can then perform one of the following actions.

- Manipulate this static string before sending it back to the Web browser.
- Save the string as a file, which enables you to obtain a snapshot of the file and then periodically update it, if required (by using notifications, for example).

Tip Use this method to reduce generation and Windows resource overhead in situations where the data seldom changes or it is not necessary to have the most current data available.

The parameters for the **generateHTMLStatic** method are listed in the following table.

Parameter	Set this parameter to true if you want …
includeDefaultSessionInfo	JADE to automatically generate the hidden fields that are sent with the form for session control.
useControlNames	JADE to use control names instead of oids in the HTML generation. Note that control names may be duplicated when there are cloned controls that have not been given another name. If the control was an input field (for example, a text box), the Web browser will return multiple *name-value* pairs with the same names. It is your responsibility to handle this situation.
generateFormDataOnly	JADE to generate only the HTML between the **<FORM>** and **</FORM>** tags, inclusive. You can then insert this string in a user template that sets up the header and footer parts of the Web page, for example.
internetExplorerOnly	To generate Internet Explorer 4.0 (and above) specific code.
machineName	To specify the name of the Web server machine.
virtualDirectory	To specify the name of the virtual directory for HTML-enabled applications on the Web server (or IIS).
protocol	To specify the protocol for transmitting data; for example, Hypertext Transfer Protocol (HTTP) or HyperText Transfer Protocol Secure (HTTPS).

If you do not specify all three **machineName**, **virtualDirectory**, and **protocol** parameters, the application code must provide this information.

See also the **Form** class **generateHTML** method.

getCellFromPosition

Signature
```
getCellFromPosition(x:      Real;
                    y:      Real;
                    row:    Integer output;
                    column: Integer output): Boolean;
```

The **getCellFromPosition** method of the **Table** class assigns to the **row** and **column** parameter values the position of the cell of the current sheet of a table corresponding to the horizontal and vertical **x** and **y** parameters (for example, the coordinates of the **Table** class **mouseDown** event), in units specified by the value of the **scaleMode** property.

This method returns **true** if the **row** and **column** parameter values of the cell have also been returned, or it returns **false** if the **x** and **y** parameters do not correspond to a cell position.

getCellSelected

Signature
```
getCellSelected(r: Integer;
                c: Integer): Boolean;
```

The **getCellSelected** method of the **Table** class returns the selected status of the cell specified in the **r** and **c** parameters (row and column) of the current sheet of a table.

This method achieves the same as the **selected** property, except that the **row** and **column** properties do not need to be set. The **getCellSelected** method has no impact on the values of the current **row** and **column** properties. See also the **setCellSelected**, **selectedCount**, and **selectedNext** methods.

getCellText

Signature
```
getCellText(r: Integer;
            c: Integer): String;
```

The **getCellText** method of the **Table** class returns the text of the cell specified in the **r** and **c** parameters (row and column) of the current sheet of a table.

This method achieves the same as accessing the **text** property, except that the **row** and **column** properties do not need to be set. The **getCellText** method has no impact on the current values of the **row** and **column** properties. (See also the **setCellText** method.)

getCharacterFormat

Signature
```
getCharacterFormat(selection:  Boolean;
                   faceName:   String output;
                   size:       Real output;
                   color:      Integer output;
                   bold:       Integer output;
                   italic:     Integer output;
                   strikethru: Integer output;
                   underline:  Integer output);
```

The **getCharacterFormat** method of the **JadeRichText** class retrieves common character formatting attributes of the receiver.

The method parameters are listed in the following table.

Parameter	Description
selection	Specify **true** to get the character format of any selected text, or if no text is selected, of the insertion point and specify **false** to get the default character format of the control
faceName	Obtains the value of the **selFontName** property
size	Obtains the value of the **selFontSize** property
color	Obtains the value of the **selTextColor** property
bold	Obtains the value of the **selFontBold** property
italic	Obtains the value of the **selFontItalic** property
strikethru	Obtains the value of the **selFontStrikethru** property
underline	Obtains the value of the **selFontUnderline** property

If no text is selected, this method applies to the insertion point. The character formatting of the insertion point is applied to newly inserted text if the current selection is empty.

When the selection changes, the default formatting changes to match the first character in the new selection.

Although individual properties enable you to get character formatting attributes, you should consider the number of requests made to the control, particularly when running the JADE application in thin client mode. For example, calling the **getCharacterFormat** method involves one request from the application server but making individual calls from the application server to the presentation client for specific formatting information within the control requires seven requests.

getClipBuffer

Signature `getClipBuffer(buffnum: Integer): String;`

The **getClipBuffer** method of the **JadeTextEdit** class returns the contents of the editor text (clip) buffer specified in the **buffnum** parameter. The value of the **buffnum** parameter can be in the range zero (**0**) through **JadeTextEdit.CLIPBUFFER_MAX**.

For details about setting the editor text buffer, see the **setClipBuffer** method.

getCollection

Signature `getCollection(): Collection;`

The **getCollection** method returns the collection attached to the current sheet of a **Table** control, to the list portion of a **ComboBox** control, or to a **ListBox** control by using the **listCollection** or **displayCollection** method.

If no collection is attached to the control, null is returned. For a table control, the collection attached to the current sheet is returned.

getControl

Signature `getControl(id: Integer): SelfType;`

The **getControl** method of the **Control** class accesses references to dynamically created controls of the same type as the calling control for the method. (The **controlCount** or **controls** method of the **Form** class can also be used to get a form control.)

If no control copy matching the unique identifier specified in the **id** parameter exists, **null** is returned.

The code fragment in the following example shows the use of the **getControl** method.

```
// A transaction has been created, resulting in a notification being sent.
// The notification is registered in the addTickerTape method of MarketForm
ttFrame  := sourceLabel.getControl(nextLabel).TTLabel;
while tt <> null and tt.caption <> null do
    count := count + 1;
    tt := ttF.sourceLabel.getControl(ttFrame.nextLabel + count).TTLabel;
endwhile;
```

getControlByName

Signature `getControlByName(controlName: String): Control;`

The **getControlByName** method of the **Form** class returns the control on the form with the name specified by the value of the **controlName** parameter. If the control is not on the form, a **null** value is returned.

The following example shows the use of the **getControlByName** method.

```
vars
    frm: BingSearch;
    control: Control;
begin
    create frm;
    control := frm.getControlByName("btnSearch");
    write control;
epilog
    delete frm;
end;
```

getControlWindowId

Signature `getControlWindowId(): Integer;`

The **getControlWindowId** method of the **Control** class returns the Windows identifier that JADE creates and assigns to each control window.

This Windows identifier of the control is used by some testing tools to locate the control elements involved in the script processing via a Windows API call and it can be accessed from your JADE code.

For details about the control identification number that is passed to Windows, see "Testing Tools and Control Identification".

Applies to Version: 2016.0.01 and higher

getCoordinates

Signature

```
getCoordinates(which: Integer;
               xx:    Integer output;
               yy:    Integer output): Boolean;
```

The **getCoordinates** method of the **JadeTextEdit** class returns the coordinates of the requested location (in pixels) relative to the client area of the text editor. The location can be outside the client area.

Use the **which** parameter to specify **LOCAT_MOUSEPOINTER** (0) for the cursor position or **LOCAT_CARET** (1) for the current caret position. If the coordinates are inside the client area, this method returns **true**.

The code fragment in the following example shows the use of the **getCoordinates** method.

```
vis := jteSource.getCoordinates(JadeTextEdit.LOCAT_MOUSEPOINTER, xx, yy);
```

getDeskTopWorkArea

Signature

```
getDeskTopWorkArea(lft:  Integer output;
                   tp:   Integer output;
                   wdth: Integer output;
                   hgt:  Integer output);
```

The **getDeskTopWorkArea** method of the **Window** class retrieves the size of the desktop work area (that is, the area outside the system taskbar and application desktop toolbars), based on the values specified in the parameters listed in the following table.

Parameter	Obtains the…
lft	Left point of the desktop work area
tp	Top point of the desktop work area
wdth	Width of the desktop work area
hgt	Height of the desktop work area

When the workstation is running multiple desktops, the **getDeskTopWorkArea** method returns the available desktop area of the primary monitor. See also the **Window** class **getMonitorArea** and **getMonitorWorkArea** methods.

getEndPosition

Signature

```
getEndPosition(): Integer;
```

The **getEndPosition** method of the **MultiMedia** class returns the end position of the content of the medium in the current device.

The end position is expressed in units of the **timeFormat** property.

getFloatingPosition

Signature
```
getFloatingPosition(lft: Integer output;
                    tp:  Integer output;
                    wth: Integer output;
                    hgt: Integer output);
```

The **getFloatingPosition** method of the **JadeDockBase** class returns the most-recent left, top, width, and height screen positions and size of the floating form on which the control resides or last resided in the **lft**, **tp**, **wth**, and **hgt** parameters, respectively. The control does not have to be the first child of the floating form.

For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document. See also the docking examples under the **JadeDockContainer** class **allowDocking** property, earlier in this document.

getFormLeft

Signature
```
getFormLeft(): Real;
```

The **getFormLeft** method of the **Control** class returns the absolute left position of the control on the form in pixels.

If the parent of the control is the form, the **getFormLeft** method returns the same value as the **Window** class **left** property (unless the value of the **Window** class **scaleMode** property is not in pixels, in which case it uses the unit of measurement defined by the scale mode; for example, twips or points).

getFormTop

Signature
```
getFormTop(): Real;
```

The **getFormTop** method of the **Control** class returns the absolute top position of the control on the form in pixels.

If the parent of the control is the form, the **getFormLeft** method returns the same value as the **Window** class **top** property (unless the value of the **Window** class **scaleMode** property is not in pixels, in which case it uses the unit of measurement defined by the scale mode; for example, twips or points).

getControlByName

Signature
```
getControlByName(controlName: String): Control;
```

The **getControlByName** method of the **Form** class returns the control on the form with the name specified by the value of the **controlName** parameter. If the control is not on the form, a **null** value is returned.

The following example shows the use of the **getControlByName** method.

```
vars
    frm: BingSearch;
    control: Control;
begin
    create frm;
    control := frm.getControlByName("btnSearch");
    write control;
epilog
    delete frm;
end;
```

getFormParent

Signature `getFormParent(): Form;`

The **getFormParent** method of the **Form** class returns the parent of the form that was set by using the **setFormParent** method or if the parent form was set directly by using a Windows API call.

Note As the **getFormParent** method does not return the MDI frame for an MDI child, use the **getMdiFrame** method if you want to return the MDI frame for an MDI child.

getGlobalSettings

Signature `getGlobalSettings(): String;`

The **getGlobalSettings** method of the **JadeTextEdit** class returns a copy of the global settings table.

getHwnd

Signature `getHwnd(): Integer64;`

The **getHwnd** method of the **Window** class returns the Microsoft Windows handle for a form or control.

Use this method instead of the **Window** class **hwnd** method. Although these methods return the same value if the value is a 32-bit integer, if you call the **hwnd** method and the Window handle is larger than a 32-bit integer, exception 1406 *(Result of expression overflows Integer precision)* is raised.

If a node can execute as a 32-bit or a 64-bit node, you should use the **getWindowHandle** method, which returns a **MemoryAddress** to ensure that the correct Windows handle is used.

getInterface

Signature `getInterface(interface: Class): IDispatch;`

The **getInterface** method of the **ActiveXControl** class returns the ActiveX interface specified in the **interface** parameter, if it exists. (As ActiveX interfaces are created as subclasses of the **IDispatch** class when an ActiveX type library is imported, use the Class List of the Class Browser to obtain the names of ActiveX interfaces, if required.)

If the specified interface does not exist, a **null** value is returned.

getLanguageName

Signature `getLanguageName(langnum: Integer;): String;`

The **getLanguageName** method of the **JadeTextEdit** class returns the name of the programming language associated with the number specified in the **langnum** parameter. For details about the languages that are understood by Scintilla, see http://scintilla.sourceforge.net/ScintillaDoc.html.

Some of the possible values for the **langnum** parameter are the **JadeTextEdit** class constants listed in the following table.

Class Constant	Value	Class Constant	Value
SCLEX_BASH	62	SCLEX_BATCH	12
SCLEX_CONF	17	SCLEX_CPP	3
SCLEX_CSS	38	SCLEX_DIFF	16
SCLEX_HTML	4	SCLEX_JADE	65
SCLEX_JAVA	65539	SCLEX_JAVASCRIPT	131075
SCLEX_MAKEFILE	11	SCLEX_PERL	6
SCLEX_PROPERTIES	9	SCLEX_PS	42
SCLEX_PYTHON	2	SCLEX_TEXT	1
SCLEX_VB	8	SCLEX_VBSCRIPT	28
SCLEX_XML	5		

See also the **JadeTextEdit** class **language** property.

The code fragment in the following example shows the use of the **getLanguageName** method.

```
languageList := "";
foreach int in 1 to 80 do
    langName := jteRead.getLanguageName(int);
    if langName <> "" then
        languageList := languageList & int.String & "=" & langName & Tab;
    endif;
endforeach;
```

getLength

Signature `getLength(): Integer;`

The **getLength** method of the **MultiMedia** class returns the length of the content of the medium in the current device. The length is expressed in units of the **timeFormat** property.

The code fragment in the following example shows the use of the **getLength** method.

```
if mode = MultiMedia.Mode_Stopped and pos >= cd.getLength then
    cd_notifyMode(cntrl, MultiMedia.Mode_Stopped);
endif;
```

getLine

Signature

```
getLine(lineNumber:     Integer;
        format:         Integer;
        firstCharIndex: Integer output;
        lineLength:     Integer output;
        lineHeight:     Integer output): String;
```

The **getLine** method of the **JadeRichText** class returns a string containing the line specified in the **lineNumber** parameter in the format specified in the **format** parameter.

The **getLine** method parameters are listed in the following table.

Parameter	Description
lineNumber	The line being requested, where the first line of the control is line **1**.
format	Format in which the requested line is returned (**JadeRichText.GetLine_PlainText** to return plain text or **JadeRichText.GetLine_RTF** to return text including RTF codes).
firstCharIndex	An output value specifying the index of the first character in the line.
lineLength	An output value specifying the number of characters in the line. (Each embedded object counts as one character).
lineHeight	An output value specifying the height (in pixels) of the line.

If the **lineNumber** parameter is less than **1** or greater than the number of lines in the control, an exception is raised.

getLineFromCharacterIndex

Signature `getLineFromCharacterIndex(index: Integer): Integer;`

The **getLineFromCharacterIndex** method of the **JadeRichText** class returns the number of the line that contains the character at the position specified in the **index** parameter.

getLineHeight

Signature

```
getLineHeight(): Integer;                    (ListBox)

getLineHeight(line: Integer): Integer;       (JadeTextEdit)
```

The **getLineHeight** method of the **ListBox** class returns the height in pixels of each list box entry.

If the value of the **ListBox** class **defaultLineHeight** property is greater than zero (**0**), the **getLineHeight** method returns that value.

The **getLineHeight** method of the **JadeTextEdit** class returns the height in pixels of the text editor line specified in the **line** parameter, which is 1-relative and must therefore be greater than zero (**0**).

getLineStartPosition

Signature `getLineStartPosition(line: Integer): Integer;`

The **getLineStartPosition** method of the **JadeTextEdit** class returns the zero-based character offset of the first character in the line specified in the **line** parameter.

If the specified line number is less than **1**, this method returns the character offset of the start of the line that contains the start of the current selection. If the specified line number is greater than the number of lines in the text editor, it returns **-1**.

getLineText

Signature
```
getLineText(linenum: Integer): String;
```

The **getLineText** method of the **JadeTextEdit** class returns a string containing the text in line specified in the **linenum** parameter, which is 1-relative and must therefore be greater than zero (**0**).

No end-of-line sequence conversion is performed.

getLinemarkLines

Signature
```
getLinemarkLines(markNumber: Integer;
                 lineList:   Boolean;
                 list:       IntegerArray input);
```

The **getLinemarkLines** method of the **JadeTextEdit** class populates the array specified in the **list** parameter with each occurrence (line or position) of the linemark specified in the **markNumber** parameter.

Note The list is limited by the **DskParam** size to 128 entries.

Set the **lineList** parameter to **true** if you want the list array to contain line numbers (**1**-relative) or to **false** if you want it to contain character offsets to the start of each marked line.

getListIndex

Signature
```
getListIndex(x: Real;
             y: Real): Integer;
```

The **getListIndex** method of the **ListBox** class returns the index of the displayed list entry in the list box corresponding to the position specified by the **x** and **y** parameters (in units specified by the value of the **scaleMode** property), provided that the specified position corresponds to any point on the associated entry line of the item within the list box.

If the position specified by the horizontal and vertical **x** and **y** parameters does not correspond to a displayed list entry, a value of **-1** is returned. For example, you can use this method during the **mouseMove** method to determine the entry that the mouse is passing over.

The method in the following example shows the use of the **getListIndex** method.

```
listInstances_mouseMove(listbox: ListBox input;
                        button:  Integer;
                        shift:   Integer;
                        x, y:    Real) updating;
vars
    cust : Customer;
begin
    if listbox.getListIndex(x, y) <> -1 and
        listbox.getListIndex (x, y) <> lastIndex then
        cust := listbox.itemObject[listbox.getListIndex(x, y)].Customer;
        listbox.bubbleHelp := cust.name.toUpper & Cr
                    & "________________________________________" & Cr
                    & cust.address & Cr & cust.contact;
        lastIndex := listbox.getListIndex(x, y);
    endif;
    if listbox.getListIndex (x, y) = -1 then
```

```
            listbox.bubbleHelp := null;
        endif;
    end;
```

getListIndexText

Signature `getListIndexText(x: Real;`
`                  y: Real): Integer;`

The **getListIndexText** method of the **ListBox** class returns the index of the displayed list entry in the list box corresponding to the position specified by the **x** and **y** parameters (in units specified by the value of the **scaleMode** property), provided that the specified position is within the **itemText** text portion of the list entry.

If the position specified by the horizontal and vertical **x** and **y** parameters does not correspond to a displayed list entry, a value of **-1** is returned.

The method in the following example shows the use of the **getListIndexText** method.

```
    vars
        indx : Integer;
    begin
        indx := listbox.getListIndexText(x, y);
        if indx > 0 and indx <> lastListIndex2 then
            lastListIndex2     := indx;
            listbox.bubbleHelp := "Entry # " & indx.String;
        endif;
    end;
```

getMdiFrame

Signature `getMdiFrame(): Form;`

The **getMdiFrame** method of the **Form** class accesses a reference to the current MDI frame for a form. (The **mdiFrame** property of the **Application** class contains a reference to the class of the next MDI frame, which may not necessarily be the current MDI frame for a form.)

This method returns a **null** value if the form is not an MDI form or the MDI frame is the supplied default (that has no JADE object).

Use the **getFormParent** method to return the parent of the form specified by using the **setFormParent** method or if the parent form was set directly by using a Windows API call.

getMode

Signature `getMode(): Integer;`

The **getMode** method of the **MultiMedia** class returns the current operating mode of the current device.

The mode that is returned can be one of the values listed in the following table.

MultiMedia Class Constant	Returned Mode	Description
Mode_Not_Ready	1	Not ready
Mode_Open	7	Door open

MultiMedia Class Constant	Returned Mode	Description
Mode_Paused	6	Paused
Mode_Playing	3	Playing
Mode_Recording	4	Recording
Mode_Seeking	5	Seeking
Mode_Stopped	2	Stopped

The method in the following example shows the use of the **getMode** method.

```
stop_click(btn: Button input) updating;
begin
    if cd.getMode = MultiMedia.Mode_Playing then
        cd.stop;
        cd.position := cd.getStartPosition;
    endif;
end;
```

getMonitorArea

Signature
```
getMonitorArea(left:   Integer output;
               top:    Integer output;
               width:  Integer output;
               height: Integer output);
```

The **getMonitorArea** method of the **Window** class retrieves the full area of the current monitor on which the window resides.

The **getMonitorArea** method parameters are listed in the following table.

Parameter	Description
left	Obtains the left point of the monitor area
top	Obtains the top point of the monitor area
width	Obtains the width of the monitor area
height	Obtains the height of the monitor area

Use the **Window** class **getDeskTopWorkArea** method to retrieve the available desktop area of the primary monitor. (For details about running an application on a workstation with multiple monitors, see "Window Class", earlier in this document.)

getMonitorWorkArea

Signature
```
getMonitorWorkArea(left:   Integer output;
                   top:    Integer output;
                   width:  Integer output;
                   height: Integer output);
```

The **getMonitorWorkArea** method of the **Window** class retrieves the full area of the position of the available desktop area of the monitor on which the window resides.

The **getMonitorArea** method parameters are listed in the following table.

Parameter	Description
left	Obtains the left point of the monitor area
top	Obtains the top point of the monitor area
width	Obtains the width of the monitor area
height	Obtains the height of the monitor area

If the window spans multiple monitors, the monitor containing the greater area of the control that has focus is used.

Use the **Window** class **getDeskTopWorkArea** method to retrieve the available desktop area of the primary monitor. (For details about running an application on a workstation with multiple monitors, see "Window Class", earlier in this document.)

getNamedAttribute

Signature `getNamedAttribute(attName: String): Any;`

The **getNamedAttribute** method of the **JadeTextEdit** class returns the value of the named attribute specified in the **attName** parameter. (For details about setting named attributes and those implemented by the **JadeTextEdit** control, see the **setNamedAttribute** method.)

The method in the following example shows the use of the **getNamedAttribute** method.

```
mnuEditSetIndentwrds_click(menuItem: MenuItem input) updating;
vars
    val : String;
begin
    val := jteSource.getNamedAttribute("smart.indent.words").String;
    if self.askForStringSetting(self.caption, "Smart Indent Words", val) then
        jteSource.setNamedAttribute("smart.indent.words", val);
    endif;
end;
```

getParagraphFormat

Signature
```
getParagraphFormat(leftIndent:      Integer output;
                   rightIndent:     Integer output;
                   firstLineIndent: Integer output;
                   alignment:       Integer output);
```

The **getParagraphFormat** method of the **JadeRichText** class retrieves common paragraph formatting attributes of the receiver (that is, the paragraph that contains the insertion point).

The **getParagraphFormat** method parameters are listed in the following table.

Parameter	Description
leftIndent	Obtains the value of the **leftIndent** property
rightIndent	Obtains the value of the **rightIndent** property
firstLineIndent	Obtains the value of the **firstLineIndent** property
alignment	Obtains the value of the **alignment** property

Although individual properties enable you to get paragraph formatting attributes, you should consider the number of requests made to the control, particularly when running the JADE application in thin client mode. For example, calling the **getParagraphFormat** method involves one request from the application server but making individual calls from the application server to the presentation client for specific formatting information within the paragraph requires four requests.

getPersistentObject

Signature `getPersistentObject(): Window;`

The **getPersistentObject** method of the **Window** class returns a reference to the persistent object of the receiver.

getPropertyDisplay

Signature
```
getPropertyDisplay(property: String;
                   str:      String io): Boolean;
```

The **getPropertyDisplay** method of the **Window** class is primarily for use with ActiveX controls and user-defined controls, where the ActiveX or user-defined control is requested to return its textual representation of a property value.

The parameters of the **getPropertyDisplay** method are listed in the following table.

Parameter	Description
property	Name of the property being requested
str	Returned string representation

The return values of the **getPropertyDisplay** method are listed in the following table.

Value	Description
true	The **str** parameter contains the property string text
false	Not implemented for this property

The **getPropertyDisplay** method is called by the Painter for a property to give it the opportunity of formatting the property value for display. If a string is returned, the Painter uses it. If a string is not returned, the Painter default formatting is used.

Note If you want your subclassed controls to return their textual representation of a property value, you must reimplement this method so that it can be called by Painter.

The method in the following example shows the use of the **getPropertyDisplay** method.

```
getPropertyDisplay(property: String; str: String io): Boolean;
begin
    inheritMethod(property, str);
    if property = "clockType" then
        str := "clockType";
        return true;
    endif;
end;
```

getRedoAndUndoState

Signature
```
getRedoAndUndoState(canRedo:    Boolean output;
                    canUndo:    Boolean output;
                    canPaste:   Boolean output;
                    redoAction: Integer output;
                    undoAction: Integer output);
```

The **getRedoAndUndoState** method of the **JadeRichText** class obtains the redo, undo, and paste operations that can be performed and the type of redo and undo actions that can be performed.

The **canRedo**, **canUndo**, and **canPaste** parameters retrieve **true** if a redo, undo, and paste operation, respectively, can be performed in the receiver or **false** if they cannot.

Note You can undo or redo up to 100 edit actions.

The values that can be retrieved in the **redoAction** parameter are the following **JadeRichText** class constants.

- **Redo_Cut**
- **Redo_Delete**
- **Redo_DragDrop**
- **Redo_Paste**
- **Redo_Typing**
- **Redo_Unknown**

The values that can be retrieved in the **undoAction** parameter are the following **JadeRichText** class constants.

- **Undo_Cut**
- **Undo_Delete**
- **Undo_DragDrop**
- **Undo_Paste**
- **Undo_Typing**
- **Undo_Unknown**

getRegisteredFormKeys

Signature `getRegisteredFormKeys(array: IntegerArray);`

The **getRegisteredFormKeys** method of the **Form** class populates an array of the form keys that are in effect for the form of the receiver. This array contains entries only if the **registerFormKeys** method of the **Form** class has been called.

The method in the following example returns whether the Tab key is registered for any control on the form.

```
isTabRegistered():Boolean
vars
    aray : IntegerArray;
    ky   : Integer;
begin
```

```
        create aray transient;
        getRegisteredFormKeys(aray);
        foreach ky in aray do
            if ky = 9 then
                return true;
            endif;
        endforeach;
        return false;
    epilog
        delete aray;
    end;
```

getRegisteredKeys

Signature `getRegisteredKeys(array: IntegerArray);`

The **getRegisteredKeys** method of the **Control** class populates an array of the keys that are in effect for the control of the receiver (that is, the keys that will generate **keyUp**, **keyDown**, and **keyPress** events when they are pressed).

This method applies only to controls that can get the focus and therefore receive key events.

This array contains entries only if the **registerKeys** method of the **Control** class or **TextBox** control class has been called.

getScrollRange

Signature

```
getScrollRange(scrollBar: Integer;
               min:       Integer output;
               max:       Integer output;
               smallChg:  Integer output;
               largeChg:  Integer output);
```

The **getScrollRange** method determines the scroll range information for the window. The **getScrollRange** method gets the scroll ranges for **Form**, **ListBox**, **TextBox**, **BaseControl**, **Picture**, and **JadeRichText** controls.

Scroll range data has no impact unless the window also has a corresponding scroll bar.

TextBox controls can obtain the current scroll range but cannot set the current scroll range. This range is determined automatically by the amount of text data in the control.

ListBox controls also only offer the **getScrollRange** method, as the ranges are set automatically by the control.

The scroll data that is available using the method parameters is listed in the following table.

Parameter	Description
scrollBar	**1** for horizontal, **2** for vertical
min	Minimum of scroll range
max	Maximum scroll range
smallChg	Size of scroll change when user clicks a scroll arrow
largeChg	Size of scroll change when user clicks elevator or uses Page keys

All data units are in pixels.

For **Form** and **Picture** controls, the default scroll range data is listed in the following table.

Value	**Default**
min	0
max	1000
smallChg	20
largeChg	100

For a **TextBox** control, the default scroll range data is determined by the amount of text in the control and cannot be set. For text controls, **smallChg** and **largeChg** data returned for the **getScrollRange** method is zero (**0**), as these are controlled by Windows and are unobtainable.

Control of the scroll bar position can be obtained by using the **scrollHorzPos** and **scrollVertPos** properties.

For the **JadeRichText** control, scroll range data has no impact unless the control also has a corresponding scroll bar. The scroll data that is available using the method parameters is listed in the following table.

Parameter	**Description**
scrollBar	**ScrollBars_Horizontal** or **ScrollBars_Vertical**
min	Minimum of scroll range
max	Maximum scroll range
smallChg	Always retrieves zero (**0**) as it does not apply to the **JadeRichText** control
largeChg	Size of scroll change when user clicks elevator or uses Page keys

All data units are in pixels.

getStartPosition

Signature `getStartPosition(): Integer;`

The **getStartPosition** method of the **MultiMedia** class returns the start position of the content of the medium in the current device.

The start position is expressed in units of the **timeFormat** property.

The method in the following example shows the use of the **getStartPosition** method.

```
stop_click(btn: Button input) updating;
begin
    if cd.getMode = MultiMedia.Mode_Playing then
        cd.stop;
        cd.position := cd.getStartPosition;
    endif;
end;
```

getSystemColor

Signature `getSystemColor(displayElement: Integer): Integer;`

The **getSystemColor** method of the **Window** class returns the system color specified in the **displayElement** parameter for the window.

In JADE thin client mode, this method always returns the colors defined for the presentation client.

The method in the following example shows the use of the **getSystemColor** method.

```
create() updating;
begin
    borderStyle      := BorderStyle_None;
    width            := 30;
    height           := 30;
    backColor        := getSystemColor(Color_3DFace);
    transparent      := true;
    transparentColor := 192 + 192*256 + 192*256*256;
end;
```

For a list of the system color constants, see "Window Class Constants".

getSystemMetrics

Signature `getSystemMetrics(index: Integer): Integer;`

The **getSystemMetrics** method of the **Window** class returns the system metric value (width and height) specified in the **index** parameter for the window.

The system metric values are the sizes of the standard window objects. In JADE thin client mode, this method always returns information as defined for the presentation client.

For a list of the system metrics constants, see "Window Class Constants".

getTabStops

Signature `getTabStops(): IntegerArray;`

The **getTabStops** method of the **JadeRichText** class returns an array containing the positions of the tab stops in the receiver control.

getTextAsDate

Signature `getTextAsDate(): Date;`

The **getTextAsDate** method returns the text from the **textUser** property of a **JadeEditMask** control converted to a **Date** value.

If the **mask** property does not specify that the data is a full date field (for example, **ddMMMyyyy** or **dd/MM/yyyy**) or if the date data is incomplete but not empty, an exception is raised.

Use the **isEmpty** and **isValid** methods to prevent incomplete date data from generating an exception.

Notes By default, a **JadeEditMask** control expects dates to be formatted according to the locale that the control is using (for details, see the **languageId** property). This applies when accessing the **text** or **textUser** property and when the user enters data.

By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default short date format of **dd/MM/yyyy**, and it has been overridden with a short date format of **yyyy-MM-dd**, this is returned in the default **dd/MM/yyyy** format.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server. When the parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

Setting date text by using the **setTextFromDate** method converts the date into the appropriate string for that locale. Retrieving the **Date** value by using the **getTextAsDate** method does the reverse. You can therefore use the **getTextAsDate** and **setTextFromDate** methods to access date text so that JADE handles the locale format for you.

getTextAsCurrencyDecimal

Signature `getTextAsCurrencyDecimal(): Decimal;`

The **getTextAsCurrencyDecimal** method returns the text from the **text** property of a **TextBox** control in currency format converted to a **Decimal** value.

If the text box is empty, zero (**0**) is returned. If the text of a **TextBox** control is not a valid numeric string in the locale of the user application, an exception is raised. You should therefore use this method only for a numeric text box (that is, the **dataType** property value of **DataType_Numeric** (1), **DataType_SignedNumeric** (2), or **DataType_Currency** (3)) that guarantees the text is a valid numeric. The text can be valid but incomplete (for details, see the **isValid** method).

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The value of **decimals** property of the text box determines the number of decimal places in the returned decimal.

Notes A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** property and when the user enters data.

Setting decimal text by using the **setTextFromCurrencyDecimal** method converts the numeric into the appropriate string for that locale. Retrieving the **Decimal** value by using the **getTextAsCurrencyDecimal** does the reverse. You can therefore use the **getTextAsCurrencyDecimal** and **setTextFromCurrencyDecimal** methods to access decimal text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123**-).

getTextAsCurrencyReal

Signature `getTextAsCurrencyReal(): Real;`

The **getTextAsCurrencyReal** method returns the text from the **text** property of a **TextBox** control in currency format converted to a **Real** value.

If the text box is empty, zero (**0**) is returned. If the text of a **TextBox** control is not a valid numeric string in the locale of the user application, an exception is raised. You should therefore use this method only for a numeric text box (that is, the **dataType** property value of **DataType_Numeric** (1), **DataType_SignedNumeric** (2), or **DataType_Currency** (3)) that guarantees that the text is a valid numeric. The text can be valid but incomplete (for details, see the **isValid** method).

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** property and when the user enters data.

Setting text by using the **setTextFromCurrencyReal** method converts the numeric into the appropriate string for that locale. Retrieving the currency **Real** value by using the **getTextAsCurrencyReal** does the reverse. You can therefore use the **getTextAsCurrencyReal** and **setTextFromCurrencyReal** methods to access numeric text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space; for example, (**- 123** will be **-123** and **123 -** will be **123-**).

getTextAsDecimal

Signature `getTextAsDecimal(): Decimal;`

The **getTextAsDecimal** method returns the text from the **textUser** property of a **JadeEditMask** control or the **text** property of a **TextBox** control converted to a **Decimal** value.

If the text of a **JadeEditMask** control is not a valid numeric string in the locale of the control (for details, see the **languageId** property), an exception is raised. You should therefore use this method only when the **mask** property of the control indicates a numeric field (for example, **###,##9.#**) that guarantees that the text is a valid numeric.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The text can be valid but incomplete, according to the mask (for details, see the **isEmpty** and **isValid** methods).

If the text of a **TextBox** control is not a valid numeric string in the locale of the user application, an exception is raised. You should therefore use this method only for a numeric text box (that is, the **dataType** property value of **DataType_Numeric** (1), **DataType_SignedNumeric** (2), or **DataType_Currency** (3)) that guarantees the text is a valid numeric.

The value of **decimals** property of the text box determines the number of decimal places in the returned decimal.

Notes For the **JadeEditMask** control, the number of decimal places included in the edit mask determines the number of decimal places in the returned decimal. A numeric **JadeEditMask** control expects the negative sign, decimal place, and separator characters to be in the form defined for the locale that the control is using.

A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data. Setting decimal text by using the **setTextFromDecimal** method converts the numeric into the appropriate string for that locale. Retrieving the decimal value using the **getTextAsDecimal** does the reverse. You can therefore use the **getTextAsDecimal** and **setTextFromDecimal** methods to access decimal text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **JadeEditMask** or **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123**-).

getTextAsInteger

Signature `getTextAsInteger(): Integer;`

The **getTextAsInteger** method returns the text from the **textUser** property of a **JadeEditMask** control or the **text** property of a **TextBox** control converted to an **Integer** value.

If the text of a **JadeEditMask** control is a decimal, the values are rounded to the nearest integer.

If the text is not a valid numeric string in the locale of the control (for details, see the **languageId** property), an exception is raised. You should therefore use this method only when the **mask** property of the control indicates a numeric field (for example, **###,##9.#**) that guarantees that the text is a valid numeric.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The text can be valid but incomplete, according to the mask (for details, see the **isEmpty** and **isValid** methods).

If the text of a **TextBox** control is a decimal, the values are rounded to the nearest integer. If the text is not a valid numeric string in the locale of the user application, an exception is raised. You should therefore use this method only for a numeric text box (that is, the **dataType** property value of **DataType_Numeric** (1), **DataType_SignedNumeric** (2), or **DataType_Currency** (3)) that guarantees that the text is a valid numeric.

A numeric **JadeEditMask** control expects the negative sign, decimal place, and separator characters to be in the form defined for the locale that the control is using. A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data.

Setting text by using the **setTextFromInteger** method converts the integer into the appropriate string for that locale. Retrieving the **Integer** value using the **getTextAsInteger** does the reverse. You can therefore use the **getTextAsInteger** and **setTextFromInteger** methods to access numeric text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **JadeEditMask** or **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123**-).

getTextAsInteger64

Signature `getTextAsInteger64(): Integer64;`

The **getTextAsInteger64** method returns the text from the **textUser** property of a **JadeEditMask** control or the **text** property of a **TextBox** control converted to an **Integer64** value.

If the text of a **JadeEditMask** control is a **Decimal**, the values are rounded to the nearest **Integer64** value.

If the text is not a valid numeric string in the locale of the control (for details, see the **languageId** property), an exception is raised. You should therefore use this method only when the **mask** property of the control indicates a numeric field (for example, **###,##9.#**) that guarantees that the text is a valid numeric.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The text can be valid but incomplete, according to the mask (for details, see the **isEmpty** and **isValid** methods).

If the text of a **TextBox** control is a **Decimal**, the values are rounded to the nearest **Integer64** value. If the text is not a valid numeric string in the locale of the user application, an exception is raised. You should therefore use this method only for a numeric text box (that is, the **dataType** property value of **DataType_Numeric** (1), **DataType_SignedNumeric** (2), or **DataType_Currency** (3)) that guarantees that the text is a valid numeric.

A numeric **JadeEditMask** control expects the negative sign, decimal place, and separator characters to be in the form defined for the locale that the control is using. A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data.

Setting text by using the **setTextFromInteger64** method converts the **Integer64** into the appropriate string for that locale. Retrieving the **Integer64** value using the **getTextAsInteger64** does the reverse. You can therefore use the **getTextAsInteger64** and **setTextFromInteger64** methods to access numeric text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **JadeEditMask** or **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123**-).

getTextAsLongDate

Signature `getTextAsLongDate(): Date;`

The **getTextAsLongDate** method returns the text in long date format from the **text** property of a **TextBox** control converted to a **Date** value.

If the text box is empty, null is returned. If the text is not a valid long date, an exception is raised. Use the **isValid** method to prevent incomplete long date data from generating an exception.

Notes By default, a **TextBox** control expects dates to be formatted according to the current locale. This applies when accessing the **text** property and when the user enters data.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default long date format of **dddd/MMMM/yyyy**, and it has been overridden with a long date format of **yyyy-MMMM-dddd**, this is returned in the default **dddd/MMMM/yyyy** format.

Setting date text by using the **setTextFromLongDate** method converts the long date into the appropriate string for that locale. Retrieving the long date value by using the **getTextAsLongDate** method does the reverse. You can therefore use the **getTextAsLongDate** and **setTextFromLongDate** methods to access long date text so that JADE handles the locale format for you.

getTextAsReal

Signature `getTextAsReal(): Real;`

The **getTextAsReal** method returns the text from the **textUser** property of a **JadeEditMask** control or the **text** property of a **TextBox** control converted to a **Real** value.

If the text of a **JadeEditMask** control is not a valid numeric string in the locale of the control (for details, see the **languageId** property), an exception is raised. You should therefore use this method only when the **mask** property of the control indicates a numeric field (for example, **###,##9.#**) that guarantees that the text is a valid numeric. The text can be valid but incomplete, according to the mask (for details, see the **isEmpty** and **isValid** methods).

If the text of a **TextBox** control is not a valid numeric string in the locale of the user application or it is empty, an exception is raised. You should therefore use this method only for a numeric text box (that is, the **dataType** property value of **DataType_Numeric** (1), **DataType_SignedNumeric** (2), or **DataType_Currency** (3)) that guarantees that the text is a valid numeric.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

A numeric **JadeEditMask** control expects the negative sign, decimal place, and separator characters to be in the form defined for the locale that the control is using. A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data.

Setting text by using the **setTextFromReal** method converts the integer into the appropriate string for that locale. Retrieving the **Real** value using the **getTextAsReal** does the reverse. You can therefore use the **getTextAsReal** and **setTextFromReal** methods to access numeric text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **JadeEditMask** or **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123-**).

getTextAsShortDate

Signature `getTextAsShortDate(): Date;`

The **getTextAsShortDate** method returns the text in short date format from the **text** property of a **TextBox** control converted to a **Date** value.

If the text box is empty, null is returned. If the text is not a valid short date, an exception is raised. Use the **isValid** method to prevent incomplete date data from generating an exception.

Notes By default, a **TextBox** control expects dates to be formatted according to the locale that the control is using. This applies when accessing the **text** property and when the user enters data.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default short date format of **dd/MM/yyyy**, and it has been overridden with a short date format of **yyyy-MM-dd**, this is returned in the default **dd/MM/yyyy** format.

Setting date text by using the **setTextFromShortDate** method converts the short date into the appropriate string for that locale. Retrieving the short date value by using the **getTextAsShortDate** method does the reverse. You can therefore use the **getTextAsShortDate** and **setTextFromShortDate** methods to access short date text so that JADE handles the locale format for you.

getTextAsTime

Signature `getTextAsTime(): Time;`

The **getTextAsTime** method returns the text from the **textUser** property of a **JadeEditMask** control or the **text** property of a **TextBox** control converted to a **Time** value.

If the **mask** property does not specify that the data is a time field with at least an hour and minutes mask, (for example, **hh:mm**) or if the time data is incomplete but not empty, an exception is raised. If the mask does not have a seconds part, the seconds part of the time is set to zero (**0**).

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default short time format of **HH:mm:ss** (24-hour clock), and it has been overridden with a short time format of **hh:mm:ss** (12-hour clock), this is returned in the default **HH:mm:ss** format.

Use the **isEmpty** and **isValid** methods to prevent incomplete time data from generating an exception.

If the text box is empty, a zero (**0**) time is returned. If the text is not a valid time, an exception is raised.

Note A **JadeEditMask** control or **TextBox** control expects times to be formatted according to the locale that the control is using (for details, see the **languageId** property). This applies when accessing the **text** or **textUser** property and when the user enters data.

Setting time text by using the **setTextFromTime** method converts the time into the appropriate string for that locale. Retrieving the **Time** value by using the **getTextAsTime** method does the reverse. You can therefore use the **getTextAsTime** and **setTextFromTime** methods to access time text so that JADE handles the locale format for you.

getTextExtent

Signature `getTextExtent(str: String): Integer;`

The **getTextExtent** method of the **Window** class returns the width in pixels required to display the string specified in the **str** parameter in this window using its current font.

The string is treated as a single line, unless it contains carriage return characters, in which case this method returns the width of the longest line of the string.

Note The appearance of text in a table cell when setting the text width to the result of the **getTextExtent** method may not produce the result that you require, as pixels are required for grid lines and spacing.

The method in the following example shows the use of the **getTextExtent** method.

```
listBoxSource_mouseDown(listbox:        ListBox input;
                        button, shift:  Integer;
                        x, y:           Real) updating;
vars
    count : Integer;
begin
    wantToCopy := false;
    if listBoxSource.listCount >= 1 then
        listBoxSource.itemSelected[listBoxSource.listIndex] := true;
        mousedown := true;
        foreach count in 1 to listbox.listCount do
            if listbox.itemSelected[count] <> true then
                listbox.itemEnabled[count] := false;
            endif;
        endforeach;
        frameDisplay.width := listBoxSource.text.length *
                    getTextExtent("*");
        drawInfo(x, y, listBoxSource.text);
        frameDisplay.visible := true;
        transfer := listBoxSource.itemObject[listBoxSource.listIndex].Fault;
        if shift.bitAnd(2) = 2 then
            wantToCopy := true;
        endif;
    endif;
end;
```

getTextHeight

Signature `getTextHeight(str: String): Integer;`

The **getTextHeight** method of the **Window** class returns the height in pixels required to display the string specified in the **str** parameter in this window using its current font.

The string is treated as a single line, unless it contains carriage return characters, in which case this method returns the total height of all the lines.

See also the **getTextHeightForWidth** method.

getTextHeightForWidth

Signature
```
getTextHeightForWidth(str:   String;
                      width: Integer): Integer;
```

The **getTextHeightForWidth** method of the **Window** class returns the height in pixels required to display the string specified in the **str** parameter as it would be displayed when word wrapped within a rectangle of the width specified in the **width** parameter using the font of the window. The string is treated as a single line, unless it contains carriage return characters, in which case this method returns the total height of all of the lines.

This method is similar to the **Window** class **getTextHeight** method except that the **getTextHeight** method returns the height without word wrap (a single line unless the text contains carriage return characters).

The code fragment in the following example returns the height of the **textBox1** text using word wrapping for a width of **80** using the font of the **label1** control.

```
hgt := label1.getTextHeightForWidth(textBox1.text, 80);
```

getTextLength

Signature `getTextLength(): Integer;`

The **getTextLength** method of the **JadeTextEdit** class returns the number of character of text in the text editor control.

getTextProtection

Signature
```
getTextProtection(start:  Integer;
                  length: Integer): Integer;
```

The **getTextProtection** method of the **JadeRichText** class returns the protection state of text from the position specified in the **start** parameter through to the length specified in the **length** parameter. The method returns an **Integer** value indicating whether the text is protected, unprotected, or contains a mixture of protected and unprotected text.

The return values correspond to the **JadeRichText** class constants listed in the following table.

Constant	Integer Value	Description
TextProtection_Set	1	All text in the specified range is protected
TextProtection_NotSet	0	No text in the specified range is protected
TextProtection_Mixed	#80000000	Mixture of protected and unprotected text

getTextRange

Signature
```
getTextRange(startPos: Integer;
             stopPos:  Integer): String;
```

The **getTextRange** method of the **JadeTextEdit** class returns a string containing the text from the text editor in the range specified by the **startPos** and **stopPos** parameter values. The **startPos** and **stopPos** parameters are the zero-based offsets of the first and last characters to return, respectively. A **stopPos** parameter value of **-1** specifies the last character in the text editor.

No end-of-line sequence conversion is performed.

The value of the **startPos** parameter must be less than that of the **stopPos** parameter and less than the length of the text.

getToggleKeyStates

Signature
```
getToggleKeyStates(insert:     Integer io;
                   capLocks:   Integer io;
                   numLock:    Integer io;
                   scrollLock: Integer io);
```

The **getToggleKeyStates** method of the **JadeTextEdit** class returns the on (**1**) or off (**0**) state of the Insert, CAPS LOCK, NUM LOCK, and SCROLL LOCK keys.

getUserName

Signature
```
getUserName(type: Integer): String;
```

The **getUserName** method of the **OleControl** class returns the OLE description of the OLE objects in an OLE control.

The values that can be specified for the **type** parameter are listed in the following table.

OleControl Class Constant	Value	Description
GetUsername_Full	1	Full OLE object name
GetUsername_Short	2	Short OLE object name
GetUsername_App	3	Application name

getValue

Signature

```
getValue(controlName: String;
         memberName:  String;
         paramList:   ParamListType): Any;
```

The **getValue** method of the **JadeXamlControl** class returns the value of a Windows Presentation Foundation (WPF) method or property for an entity in the XAML control.

The parameters are combined to form a sequence of accesses to the WPF entities involved. The JADE method parameters are a mixture of property names, method names, and WPF method parameters, as described in the following table.

Parameter	Description
controlName	Name of the WPF FrameworkElement involved. If the name is null or equal to the control name, the search for the **memberName** starts with the parent control; otherwise the search starts with the first child element with the specified name. The search succeeds when the entity or one of its children is found to have the specified **memberName** value. An exception is raised if the **controlName** or **memberName** is not found.
memberName	Name of the first method or property being accessed.
paramList	Remaining property, methods, and parameters being used in sequence.

The code fragments in the following examples show how these parameters are used.

```
jadeXamlCtl.getValue("list", "SelectedIndex");  /* returns the integer value of the
currently selected item of the list box entity named "list" */

jadeXamlCtl.getValue ("list", "SelectedItem", "Content");  /* returns the text of
the currently selected item of the list box (assuming the content is a string) by
executing the WPF sequence list.SelectedItem.Content. */

JadeXamlCtl.getValue ("list", "Items", "GetItemAt", 3, "Content")  /* returns the
text of the list box entry with an index of 3, (assuming the content is a string)by
executing the WPF sequence list.Items.GetItemAt(3).Content. */
```

Note the following restrictions.

- Only JADE primitives types are supported as parameters to WPF method calls.
- All names are case-sensitive.
- Access to static WPF properties and methods is not supported.
- The final property or method accessed in the sequence must return a JADE primitive type value.

- Parameter types must match the same basic type; that is, an **Integer** parameter must be passed as an **Integer**, a floating point or real number as a **Real**, a byte as a **Byte**, and so on.
- For a presentation client, calls to this method are not buffered, causing the application server to send a message to the presentation client and wait for the reply.

getWebEventMappings

Signature `getWebEventMappings(): HugeStringArray;`

The **getWebEventMappings** method of the **Window** class returns all Web event mappings for the receiver.

The method in the following example returns a list of all Web event mappings for the **textBox1** control.

```
vars
    strArray : HugeStringArray;
begin
    strArray := textBox1.getWebEventMappings;
    ...    // do some processing here
epilog
    delete strArray;
end;
```

Each entry in the returned array has the following format.

```
<event-name>=<function-name>
```

The following is an example of an entry in a returned array of Web event mappings.

```
lostFocus=processLostFocus(this, 'lostFocus')
```

getWindowHandle

Signature `getWindowHandle(): MemoryAddress;`

The **getWindowHandle** method of the **Window** class returns the Microsoft Windows handle as a **MemoryAddress** for a form or control.

If a node can execute as a 32- or 64-bit node, you should use this method to ensure that the correct Windows handle is used, rather than the **Window** class **hwnd** and **getHwnd** methods, which return an integer value.

Applies to Version: 7.1.07 (Service Pack 6) and higher

getWordAt

Signature `getWordAt(action: Integer): String;`

The **getWordAt** method of the **JadeTextEdit** class returns the word at the specified location in the text editor.

In the **action** parameter, use the **JadeTextEdit** class constant of **LOCAT_MOUSEPOINTER** (0) if you want to return the word at the current mouse position or **LOCAT_CARET** (1) if you want to return the word at the current caret position.

The word consists of all the characters that precede and follow the specified location that are currently defined as *word* characters. See the **setWordCharactersets** method.

hasAudio

Signature `hasAudio(): Boolean;`

The **hasAudio** method of the **MultiMedia** class returns **true** if the current device type that is loaded in the control supports audio. (See also the **hasVideo** property.)

hasPicture

Signature `hasPicture(): Boolean;`

The **hasPicture** method of the **Table** class returns **true** if the current cell indicated by the **sheet**, **column**, and **row** properties has a picture displayed.

hasPropertyPage

Signature `hasPropertyPage(nam: String): Boolean;`

The **hasPropertyPage** method of the **Window** class returns whether the window has its own Properties dialog for the property specified in the **nam** parameter. If the **nam** parameter contains an empty string, the method returns whether there is a global Property page for the whole window. Although this method is primarily for ActiveX controls, you can also use it for subclassed controls.

By default, the **hasPropertyPage** method returns **false**. For ActiveX controls, the ActiveX is asked if such a property exists. The Painter asks the window if it has a custom-built dialog (property page) for editing the current property selected in the Properties form.

If such a dialog exists, it is used instead of the standard Painter editing. This dialog is initiated by using the **showPropertyPage** method for the window.

Using the **Control** class **saveProperties** method saves properties edited during this process.

The method in the following example shows the use of the **hasPropertyPage** method.

```
hasPropertyPage(nam : String): Boolean updating;
begin
    if nam = 'layout' then
        return true;
    endif;
    return inheritMethod(nam);
end;
```

Notes Reimplement this method in user-defined subclassed controls in your application to return **false** if you want to stop the JADE Painter from using its own Properties dialog for the specified property.

Any Properties dialog is run under the JADE application when used by the Painter.

hasSystemTrayEntry

Signature `hasSystemTrayEntry(): Boolean;`

The **hasSystemTrayEntry** method of the **Form** class returns **true** if the form currently has a system tray icon entry, or it returns **false** if no entry is currently defined for the form (by using the **Form** class **setSystemTrayEntry** method). See also the **removeSystemTrayEntry** method.

hasVideo

Signature `hasVideo(): Boolean;`

The **hasVideo** method of the **MultiMedia** class returns **true** if the current device type that is loaded in the control supports video. (See also the **hasAudio** property.)

hwnd

Signature `hwnd(): Integer;`

The **hwnd** method of the **Window** class returns the Windows handle to a form or control.

The Windows environment identifies each form and control in an application by assigning it a handle (or hwnd). The Windows handle is useful with many Windows API calls that require **hwnd** as a parameter.

If a node can execute as a 32-bit or a 64-bit node, you should use the **getWindowHandle** method, which returns a **MemoryAddress** to ensure that the correct Windows handle is used.

Notes The value of **hwnd** can change while a program is running, if property values that determine the style of a control are altered. For example, changing the **scrollHorizontal** property of a **TextBox** control requires that the text box be recreated in the new style, resulting in a new Windows handle being assigned.

To reduce the amount of memory being used by Web sessions, JADE creates a physical window only for **Ocx**, **OleControl**, **ActiveXControl**, and **MultiMedia** controls and a form only if an **Ocx**, **OleControl**, **ActiveXControl**, or **MultiMedia** control is created. In most cases when running an HTML thin client, the **hwnd** method returns a null value unless a physical window or a form is created for an **Ocx**, **OleControl**, **ActiveXControl**, or **MultiMedia** control.

initializeAppSettings

Signature `initializeAppSettings();`

The **initializeAppSettings** method of the **JadeTextEdit** class removes all entries from the application settings table.

The code fragment in the following example shows the use of the **initializeAppSettings** method.

```
jteSource.initializeAppSettings();
jteSource.language := JadeTextEdit.SCLEX_CPP;
jteSource.updateAppSettings("font.base=font:Verdanna,size:101" & CrLf
                            & "font.comment=size:9,italic" & CrLf);
jteSource.applySettings();
```

initializeJadeEditor

Signature `initializeJadeEditor(useProfile: Boolean);`

The **initializeJadeEditor** method of the **JadeEditor** class initializes the control for editing JADE method text.

If the value of the **useProfile** parameter is **true**, the editor options for the current user are used if the control is run in the development system; otherwise, the default editor options are used. If the value of the **useProfile** parameter is **false**, the values used are those set on the control at the time this method is called.

Note Call the **initializeJadeEditor** method before you call the **setCurrentSchema** method, to ensure that the form displays entities in the correct color.

insertColumn

Signature `insertColumn(at: Integer);`

The **insertColumn** method enables a single column to be inserted into the current sheet of a **Table** control. The column is empty and assumes the default column width. The existing columns are shifted to the right of the column specified in the **at** parameter and remain untouched.

The method in the following example shows the use of the **insertColumn** method.

```
buttonAddColumn_click(btn: Button input) updating;
begin
    if selectedColumn <> null then
        table1.inputType := InputType_TextBox;
        table1.insertColumn(selectedColumn);
        table1.clearAllSelected;
        selectedColumn := null;
        comboBoxColMoveTo.addItem("Move to Column " &
                                  (comboBoxColMoveTo.listCount + 1).String);
        comboBoxColumns.addItem((comboBoxColumns.listCount + 1).String);
        refresh;
    else
        app.msgBox("You must select a column", "No column selected",
                   MsgBox_OK_Only);
        return;
    endif;
end;
```

insertObject

Signature
```
insertObject(fileName: String;
             link:     Boolean;
             icon:     Boolean);
```

The **insertObject** method of the **JadeRichText** class inserts a COM object (for example, an Excel chart, bitmap, or Word document) at the current position of the receiver (that is, at the insertion point in the control). The **insertObject** method parameters are listed in the following table.

Parameter	Description
fileName	Existing file containing the object to be inserted. If you do not specify a path, JADE looks in the current directory and raises an exception if the specified file is not in the directory or if it does not exist.
link	Specify **true** if the object should be linked or **false** if it should be embedded.
icon	Specify **true** if the object should be displayed as an icon or **false** if the object contents should be displayed.

For an example of the use of this method, see "JadeRichText Control Method Example", earlier in this document.

insertObjectDialog

Signature
```
insertObjectDialog();
```

The **insertObjectDialog** method of the **JadeRichText** class invokes the OLE Insert Object dialog that enables the user to insert a COM object at the current position of the receiver (that is, at the insertion point in the control).

insertTable

Signature
```
insertTable(rows:      Integer;
            cols:      Integer;
            left:      Integer;
            colWidths: IntegerArray);
```

The **insertTable** method of the **JadeRichText** class inserts a table at the current position of the receiver (that is, at the insertion point in the control).

The **insertTable** method parameters are listed in the following table.

Parameter	Description
rows	Specifies the number of rows in the table
cols	Specifies the number of columns in the table
left	Specifies the **leftIndent** value in the table (in pixels)
colWidths	Specifies the width of table columns (in pixels)

is3D

Signature
```
is3D(): Boolean;
```

The **is3D** method of the **Control** class returns whether the control was drawn three-dimensionally (that is, it returns **true** if the **show3D** property was set to **Show3D_Use3D** or to **Show3D_UseAppDefault** and that control was selected in the **3D Controls** list box of the **Form** sheet of the Define Application dialog).

This method returns **false** if the **show3D** property was set to **Show3D_Not3D** or the control was not selected with the three-dimensional default value for the application.

You can call this method only on a control that has an associated window.

isCaptionVisible

Signature
```
isCaptionVisible(): Boolean;
```

The **isCaptionVisible** method of the **Form** class returns **true** if the caption of a form (or the top portion of a form with no caption) is visible on at least one monitor.

If you have multiple monitors in various configurations, saved or predetermined positions can leave a form inaccessible.

For a non-MDI form, this method returns **true** if a significant part of the caption is visible in the work area of at least one monitor. For an MDI form, this method returns **true** if a significant part of the caption is visible within the client area of the MDI frame.

isDroppedDown

Signature `isDroppedDown(): Boolean;`

The **isDroppedDown** method of the **ComboBox** class returns **true** if the drop-down list of the combo box control is open.

Use the **closeDropDown** or **showDropDown** methods to show or hide the drop-down list, respectively.

isEmpty

Signature `isEmpty(): Boolean;`

The **isEmpty** method of the **JadeEditMask** class returns **true** if there is any text in character positions in which text can be entered or it returns **false** if there is no text in those positions. For example, the **isEmpty** method returns:

- False, if **21/__/____**
- True, if __/__/____

isEventEnabled

Signature `isEventEnabled(name: String): Boolean;`

The **isEventEnabled** method of the **Window** class returns **true** if the event specified in the **name** parameter is currently enabled for the window of the receiver. By default, events are enabled.

To disable an event, use the **Window** class **enableEvent** method.

isFloating

Signature `isFloating(): Boolean;`

The **isFloating** method of the **JadeDockBase** class returns whether the **JadeDockBar** or **JadeDockContainer** dock control is the first child of a floating form. For more details, see the **float** method.

A floating dock container returns **true** and any child dock controls return **false**.

For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document. See also the docking examples under the **JadeDockContainer** class **allowDocking** property, earlier in this document.

isHyperlinkSet

Signature
```
isHyperlinkSet(hyperlinkRow:    Integer;
               hyperlinkColumn: Integer): Boolean;
```

The **isHyperlinkSet** method of the **Table** class returns **true** if there is an associated HyperText link for the row specified in the **hyperlinkRow** parameter and the column specified in the **hyperlinkColumn** parameter.

If there is no HyperText link in the specified row and column, this method returns **false**.

isInPainter

Signature `isInPainter(): Boolean;`

The **isInPainter** method of the **Control** class returns **true** when the control is in the JADE Painter process (that is, the **Process** class **type** property has a value of **4**) or **false** if the control is on a runtime form.

The method in the following example shows the use of the **isInPainter** method.

```
paint(cntrl: Calendar input) updating;
vars
    today : Date;
begin
    inheritMethod(cntrl);
    if isInPainter then
        buildSelf;
    elseif not isBuilt then
        buildSelf;
        isBuilt := true;
        date    := today;
    endif;
end;
```

isMdiFloating

Signature `isMdiFloating(): Boolean;`

The **isMdiFloating** method of the **Form** class returns **true** if the MDI child form is floating; otherwise it returns **false** if the form is docked or it is not an MDI child form.

Applies to Version: 2020.0.01 and higher

isModal

Signature `isModal(): Boolean;`

The **isModal** method of the **Form** class returns **true** if the form was displayed using the **Form** class **showModal** method, or it returns **false** if the form has not been displayed or it was displayed using the **Form** class **show** method.

Applies to Version: 2016.0.01 and higher

isMoveable

Signature `isMoveable(): Boolean;`

The **isMoveable** method of the **Control** class returns **true** if the control can be dragged to move it around the form in the JADE Painter. By default, controls can be moved on a form in Painter.

isMySheetVisible

Signature `isMySheetVisible(): Boolean;`

The **isMySheetVisible** method of the **Control** class provides a control with the ability to determine if the sheet of a **Folder** control on which it is placed is the current visible sheet.

The control does not need to be a direct child of the folder.

Folders provide the ability to share the same screen space for a number of control images. The sheets of the folders are special group boxes on which controls are placed. The caption of each sheet is displayed in tabs that can be selected above the images. One sheet only is displayed at any time. If the control is not on a sheet, the **isMySheetVisible** method returns **true**.

Note Setting the focus to a control on a sheet that is not the current **topSheet** causes that sheet to become the top sheet.

isObjectOpen

Signature `isObjectOpen(): Boolean;`

The **isObjectOpen** method of the **OleControl** class specifies whether the OLE server for an OLE object in an OLE control has the object open for editing.

As this method does not apply to an object that is activated in-place and therefore has no window separate from that of its container, it returns **true** if the **allowInPlace** property is set to **false**.

isPrinterForm

Signature `isPrinterForm(): Boolean;`

The **isPrinterForm** method of the **Form** class returns whether the form was declared as a printer form on the New Form dialog in the JADE Painter; that is, the **Printer** option button was selected in the Form Style group box.

This method then allows the **setDefaultPainterControlProperties** method re-implementation to set properties only on a printer-style form, for example.

The following is an example of the **isPrinterForm** method.

```
setDefaultPainterControlProperties();
begin
    if self.form.isPrinterForm() then
        self.fontName := "Arial";
    endif;
end;
```

Applies to Version: 2018.0.02 (Service Pack 1) and higher

isSelectable

Signature `isSelectable(): Boolean;`

The **isSelectable** method of the **Control** class returns **true** if instances of the control can be selected in the JADE Painter. By default, controls can be selected in the Painter. Reimplement this method in your application to return **false** if you want to stop the JADE Painter from selecting controls of a specific type.

When this method returns **false**, the Properties dialog and the Find Control dialog exclude that control in their control list boxes.

Tip This is useful if you have created a custom (user-defined) control that has other controls embedded in it. In most cases, you would not want the embedded controls to be selected during painting.

isSizeable

Signature `isSizeable(): Boolean;`

The **isSizeable** method of the **Control** class returns **true** if the control can be resized in the JADE Painter. Reimplement this method in user-defined controls of your application to return **false** if you want to stop the JADE Painter from resizing your subclassed controls.

By default, controls can be resized in Painter.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **JadeEditMask** class or **TextBox** class returns **true** if the text is valid and complete; that is, all required fields of the text have characters in them, as defined by the **JadeEditMask** class **mask** property or the **TextBox** class **dataType** property (for example, **10/12/2010** versus **10/12**).

This method returns **false** if it is not or if the control is empty.

The following are examples of the **isValid** method return value.

- False, if **21/__/____**
- False, if **__/__/____**
- True, if **2_/1_/2001** (**2** is a valid day and **1** is a valid month)
- False, if **0_/10/2001** (**0** is not a valid day)
- False, if **12/0_/2001** (**0** is not a valid month)
- True, if **12/10/2001**

itemFullName

Signature `itemFullName(index: Integer): String;`

The **itemFullName** method of the **ListBox** class returns the fully qualified name of an item from a list box control. The fully qualified name is the concatenation of the item with its parent item, the parent item of the parent item, and so on, until the parent item at indentation level 1 is reached.

Each entry is separated from the next by the **nameSeparator** property character.

The value of the **nameSeparator** property is used to delimit the level names. The default value is **"\"**. For example, **itemFullName**(**3**) would return the string **"one\two\three"** for the following hierarchy.

```
one
    two
        three
```

itemHasSubItems

Signature `itemHasSubItems(index: Integer): Boolean;`

The **itemHasSubItems** method of the **ListBox** class or the **ComboBox** class returns a value that indicates whether an item has subitems.

The **itemHasSubItems** method is an array of **Boolean** values with the same number of items as that returned by the **listCount** method.

itemVisible

Signature `itemVisible(index: Integer): Boolean;`

The **itemVisible** method of the **ListBox** class or the **ComboBox** class returns a value that indicates whether an item in a list box control is visible; that is, part of the expanded tree.

The **itemVisible** method is an array of **Boolean** values with the same number of items as returned by the **listCount** method.

lineCount

Signature `lineCount(): Integer;`

The **lineCount** method of the **JadeRichText**, **JadeTextEdit**, and **TextBox** control classes returns the number of lines of text in a rich text control, text editor, or text box, respectively.

Note The number of lines may be greater than the number of lines that are displayed. (See also the **lines** method.)

This method returns the actual number of lines of text. If there is no text, the number of lines that are returned is still **1**.

lines

Signature `lines(): Integer;`

The **lines** method of the **JadeTextEdit**, **ListBox**, and **TextBox** control classes returns the number of lines available for display in the control. Effectively, this method returns the client height (**clientHeight** property value) divided by the height of each line.

If the **integralHeight** property is set to **false**, the last line may be a partial line, which is included in the count.

The code fragment in the following example shows the use of the **lines** method.

```
foreach count in 1 to listBoxRight.lines do
    iter.next(myProduct);
    listBoxRight.addItem(myProduct.name);
endforeach;
```

linkFromFile

Signature `linkFromFile(filename: String);`

The **linkFromFile** method of the **OleControl** class creates an OLE object, linking to the file specified in the **filename** parameter.

The file is not copied into the JADE database; only a link to that file is retained.

listCollection

Signature
```
listCollection(c:       Collection;
               update:  Boolean;
               showHow: Integer);
```

The **listCollection** method of the **ListBox** and **ComboBox** class enables list box or combo box controls to have a collection attached to them. If you use this method to attach a collection to a **ListBox** or **ComboBox** control, little is required to load the entries into the list. Use the **displayCollection** method to automatically attach only as many entries as required to fill the list of the control. The differences between the **listCollection** method and the **displayCollection** method are as follows.

- The **listCollection** method retains all entries added to the list or combo box when the user scrolls the view.
- For the **listCollection** method, the number of entries in the list (returned by **listCount**) is logically the size of the attached collection minus discarded entries. For the **displayCollection** method, the **listCount** method returns only the number of entries that are displayed.
- The **displayCollection** method enables you to specify a starting object.

Entries in the collection are retrieved only when the entry is to be displayed or it is accessed by logic; for example, only 15 entries from the collection may initially be accessed instead of the entire contents of the collection (which may be hundreds, or even thousands). You must therefore ensure that this collection remains valid for the lifetime of the control for which you called the **listCollection** method.

The **ListBox** class **listCollection** method handles a virtual collection, but only in the forward direction. (Attempting to call the **displayCollection** method in a **ListBox** or **Table** control with a virtual collection is rejected and an exception is raised.)

The **listCollection** method is not available in tables on forms in Web-enabled applications, as the HTML framework cannot predict the final size of the table and adjust the HTML page accordingly. If you want to ensure that all possible entries in the table are displayed on a Web page, use some other "virtual window" for the Web generation, or populate the table with all entries from the underlying collection if you know the number of rows will not cause excessive page size.

The parameters passed in the **listCollection** method are listed in the following table.

Parameter	Description
c	Specifies the attached collection and can be any type of collection
update	If **true**, changes to the collection are to be reflected in the list box (not available for transient objects)
showHow	If **0**, displays the collection in collection order, and if **1**, displays the collection in reverse collection order

The **ListBox** class **listCollection** method handles a virtual collection, but only in the forward direction. (Attempting to call the **displayCollection** method in a **ListBox** or **Table** control with a virtual collection (for example, **myClass.instances**) is rejected and an exception is raised. Virtual collections do not implement the methods required by the **displayCollection** method).

The attachment of a collection to a list box or combo box functions as follows.

1. Logic attaches the collection to the list box or combo box, by using the **listCollection** method, as shown in the following example that associates a collection with a list box called **listProducts**.

```
load() updating;
vars
    company : Company;
begin
    company := Company.firstInstance;
    listProducts.listCollection(company.allProducts, true, 0);
end;
```

2. The collection is displayed in order (or reverse order) of the entries in the collection, unless the value of the **sorted** property for the list box is set to **true**.

3. To specify the text to be displayed for each entry in the collection, the **displayEntry** event for the list box or combo box is called, as shown in the method in the following example.

```
listProducts_displayEntry(listbox:  ListBox input; obj: Any;
                          lstIndex: Integer): String updating;
vars
    prod : Product;
begin
    prod := obj.Product;
    return " " & prod.code & " " & prod.name & " ";
end;
```

The **displayEntry** event uses the parameters listed in the following table.

Parameter	Description
control-type	Passes the type of control; that is, **ListBox** or **ComboBox**.
obj	Passes the object to be displayed (it must be cast to the required type for actual access).
lstIndex	Passes the position at which the entry is placed in the list box or combo box, so that colors, levels, and so on can also be set. The text to be displayed is returned as a string. (It is the responsibility of the JADE developer to pass the text back.)

If the **displayEntry** event returns an empty string, that entry is ignored and is not included in the list box and no further items in the collection are displayed. When the **listCollection** method is used to associate a collection with a list box or combo box, the **itemObject** property contains a reference to the object displayed in each entry of the list box or combo box.

Note When an exception is raised by the **displayEntry** or **displayRow** event method, the size of the collection is treated as being one less than the number of entries already processed.

No further attempts are made to access the additional entries from the collection until a new **displayCollection** or **listCollection** method is executed against the control.

4. If the value of the **sorted** property is **true**, the entire contents of the collection are accessed during the **listCollection** method call, with calls to the **displayEntry** event for each collection object.

5. If the **update** parameter is set to **true**:

   - Deleting the collection results in the list box or combo box being cleared, and the collection is no longer associated with the list box or combo box.

   - Any changes to the collection cause the contents of the list box or combo box to be discarded, and the collection is rebuilt to the current display point (the current entry is reselected if it still exists).

If the **update** parameter is set to **false**, the list box or combo box is not updated and may contain out-dated information.

Note The value of the **update** parameter must be **false** for transient collections, as JADE does not issue system notifications for the addition, change, or deletion of a transient collection.

The following conditions apply to a list box or combo box with an attached collection.

- The **addItem** and **removeItem** methods are not available.
- The **itemObject** property returns the collection member for each entry. In this case, the **itemObject** property is read-only.
- The **getCollection** method returns the collection attached to the combo box, list box, or current sheet of the table.
- The **clear** method clears the list box or combo box and detaches the collection from the list box or combo box.
- The list box or combo box has a **beginNotification** condition established (all events). These events are also passed to the **sysNotify** event of the list box or combo box.
- When the **listCollection** method is used with the **listCount** method, the value returned by the **listCount** method is the logical number of entries in the list box or combo box (that is, it returns a total of the number of entries for which the **displayEntry** event method has already returned a string and the number of entries in the collection that are yet to be accessed).
- Setting the **topIndex** property causes the list box to be filled with entries up to that point when using the **listCollection** method and the **topIndex** property value is greater than the entries that are obtained so far.

listCount

Signature `listCount(): Integer;`

The **listCount** method of the **ComboBox** or **ListBox** class returns the current number of actual entries and entries that are yet to be read in the list portion of a list box or combo box control.

Use the **listCount** method to iterate through the entries in the list box or combo box, or to test whether there are any entries in the list box or combo box.

The code fragments in the following examples show the use of the **listCount** method.

```
app.msgBox("Done " & fromList.listCount.String, "" ,0);
if listBox1.listCount > 20 then
        listBox1.listObject := listBox1.itemObject[19];
    else
        listBox1.listObject := null;
endif;
```

The return value is adjusted when the **displayEntry** event method returns a null string (**""**), indicating that the entry does not appear in the list.

Tip It is much more efficient to copy a GUI value into a local variable for reuse rather than request the value again. For example, **listBox.listCount** requires the calling of a list box method to retrieve the value. Storing the value in a local property for reuse avoids a significant overhead for the second and subsequent requests for that value when it will not change.

The first of the following examples is much more expensive than the second of these examples.

```
while count <= listBox.listCount do   // inefficient use of the method

vars
    listCount : Integer;
begin
    listCount := listBox.listCount;   // recommended use of the method
    while count <= listCount do
        ...
    endwhile;
end;
```

When the **listCount** method is used with the **listCollection** method, the value returned by the **listCount** method is the logical number of entries in the list box or combo box (that is, it returns a total of the number of entries for which the **displayEntry** event method has already returned a string and the number of entries in the collection that are yet to be accessed).

When the **listCount** method is used with the **displayEntry** event method, the returned value is only the number of entries loaded in the list box or the list portion of the combo box. It has no relationship to the size of the attached collection.

load

Signature
```
load(text:    String;
     replace: Integer);
```

The **load** method of the **JadeRichText** class loads text starting at the current position of the receiver (that is, at the insertion point in the control). For an example of the use of this method, see "JadeRichText Control Method Example", earlier in this document.

The **load** method parameters are listed in the following table.

Parameter	Description
text	Text to be loaded. If the text starts with a valid RTF sequence (for example, **"{\rtf "**), the text is loaded in rich text format.
replace	A **JadeRichText** class constant value that indicates where the text is loaded, as follows. ■ **Load_ReplaceAll**, which replaces the current contents of the receiver. ■ **Load_ReplaceSelection**, which replaces the selected text of the receiver. ■ **Load_Append**, which appends the text to the end of the current contents of the receiver.

loadCollectionEntries

Signature `loadCollectionEntries();`

The **loadCollectionEntries** method of the **Table** class causes the table to access any collection entries required to fill the display size. (As the **displayCollection** method does this automatically, this method is now redundant.)

The **displayRow** event method is called for each entry in the collection, as required. Only the number of entries that are needed to fill the table are accessed. The entries in the table can be accessed from logic as normal, but the content of the table is treated as though it is the complete set of data.

Access to rows that are not displayed is not available. See also the **Table** class **displayCollection** method and the **displayRow** event method.

loadControl

Signature `loadControl(index: Integer): Control;`

The **loadControl** method of the **Control** class enables an existing control (with an attached window) defined in the JADE development environment to be "cloned" at run time; that is, one or more copies of that control can be created at run time on that form. Using the development copy of the control creates these new controls.

Each control calls the same methods defined for the original control, passing its own control object as the first parameter. (Note, however, that control event methods are not called if you change the name of a control.)

In addition, you must assign a unique identifier to each control, to be passed to the **loadControl** method. You can use the **index** parameter to access this unique identifier through the control.

Note Although all primitive properties values are copied, the **loadControl** method clones only the persistent instance of the control; not the transient runtime instance.

Controls created in Painter in the JADE development environment or by using the **addControl** method have an **index** parameter value of zero (**0**). Most commonly, the value of the **index** parameter would be just that: an index. The values do not need to be sequential, but they cannot be duplicated.

The code fragment in the following example shows the creation of a new control.

```
cntrl := c.loadControl(id);
```

In the above example, the **c** value is the control that is being cloned, the **cntrl** value is the new copy of the created control, and the **id** value is the unique copy index supplied by the caller.

Any control that is added is automatically deleted when the form is destroyed. These controls can also be deleted dynamically, by using the **delete** instruction.

Access to these cloned controls can be achieved by using the **controls** method or by using the **getControl** method.

loadFile

Signature `loadFile(fileName: String);`

The **loadFile** method of the **WebInsert** class dynamically loads the specified file as part of the HTML generate process.

Use the **fileName** parameter to specify the fully qualified name of the text file that you want inserted into the Web page during the HTML generate. This file name includes the full path of the JADE application if the file is not located in the default directory.

If the file cannot be located or it cannot be read, an error is displayed in an exception dialog.

loadFromDB

Signature `loadFromDB(): Integer;`

The **loadFromDB** method of the **OleControl** class loads the control from its **oleObject** property. (This is done by default when the form is created.)

The method in the following example shows the use of the **loadFromDB** method.

```
load() updating;
vars
    bin   : Binary;
    obj   : Object;
    count : Integer;
begin
    foreach obj in ReviewOLEObj.instances do
        count := 1 + count;
        if count = 1 then
            oleReview1.oleObject.copy(obj.ReviewOLEObj);
            oleReview1.loadFromDB;
        elseif count = 2 then
            oleReview2.oleObject.copy(obj.ReviewOLEObj);
            oleReview2.loadFromDB;
        elseif count = 3 then
            oleReview3.oleObject.copy(obj.ReviewOLEObj);
            oleReview3.loadFromDB;
        endif;
    endforeach;
end;
```

loadFromFile

Signature
```
loadFromFile(fileName: String;
             replace:  Integer;
             format:   Integer): Integer;
```

The **loadFromFile** method of the **JadeRichText** class loads the contents of a file at the current position of the receiver (that is, at the insertion point in the control).

The **loadFromFile** method parameters are listed in the following table.

Parameter	Description
fileName	Existing file containing the file whose contents are to be read. If you do not specify a path, JADE looks in the current directory and raises an exception if the specified file is not in the directory or if it does not exist.
replace	A **JadeRichText** class constant value that indicates where the text is loaded, as follows. ■ **LoadFromFile_ReplaceAll**, which replaces the entire contents of the receiver. ■ **LoadFromFile_ReplaceSelection**, which replaces only the selected text of the receiver.
format	A **JadeRichText** class constant value that indicates the format of the loaded text, as follows. ■ **LoadFromFile_PlainText**, which is read as plain text. ■ **LoadFromFile_UnicodeText**, which is read as plain Unicode text. ■ **LoadFromFile_RTF**, which is read as rich text format text if the text starts with a valid RTF sequence (for example, **"{\rtf "**).

If the read operation of the file raised an exception, the **loadFromFile** method returns the appropriate JADE file handling error number (for example, *5030 - File is in use by another process*). If the file read operation was successful, this method returns zero (**0**).

loadPicture

Signature `loadPicture(fileName: String): IJadeAutoPicture;`

The **loadPicture** method of the **ActiveXControl** class creates a picture object from the external file specified in the **fileName** parameter, which can be a valid file name or it can be the fully qualified path and name of a valid picture file. If the specified file does not exist, a **null** value is returned.

loadTextFromFile

Signature `loadTextFromFile(fileName: String): Integer;`

The **loadTextFromFile** method of the **JadeTextEdit** class dynamically loads the specified file into the text editor (for example, into a JADE workspace).

Use the **fileName** parameter to specify the fully qualified name of the text file that you want loaded into the text editor. This file name includes the full path of the JADE application if the file is not located in the default directory.

If the file cannot be located or it cannot be read, the **loadTextFromFile** method returns the appropriate exception code rather than raising exceptions to report errors (for example, *5003 - Requested file not found*). If the file read operation was successful, this method returns zero (**0**).

The maximum size of a file that you can load into the control by using this method is the smaller of 50M bytes or a tenth of the physical memory.

An ANSI JADE system is limited to loading a **File** object of **kind** type **Kind_ANSI**.

A Unicode JADE system loads most kinds of text files. Non-ANSI files are filtered during UTF8 conversion and if an invalid Unicode sequence is detected, the load returns error **15645**.

End-of-line conversion is performed, to force all end-of-line sequences to match the current value of the **endOfLineMode** property.

If the text files contains null characters, only the text preceding the first null character are loaded.

The code fragment in the following example shows the use of the **loadTextFromFile** method.

```
str := "c:\Temp\utf8.txt";
int := jteSource.loadTextFromFile(str);
if int > 0 then
    app.msgBox("Load failed", "Load Editor text", 0);
endif;
```

makeAutomationObject

Signature `makeAutomationObject();`

The **makeAutomationObject** method of the **ActiveXControl** class creates an **ActiveXAutomation** object instead of a control, so that an ActiveX control can be treated as a standard automation object when you define a control that does not require a form in which to reside. (An ActiveX control is a standard automation object with interfaces to handle GUI requirements.)

Although the ActiveX object was imported as a control, it can be used as an automation object on the application server when you set the **usePresentationClient** property to **false**.

The following example assumes that you have imported the Microsoft **SysInfo** control type library (that is, **sysInfo.ocx**) into JADE as an ActiveX control. This example creates a JADE control instance and calls the **makeAutomationObject** method instead of adding the control to a form.

```
createSysInfoAsAutoObject();
vars
    actx : SysInfo;
begin
    create actx transient;
    actx.makeAutomationObject;
    write actx.oSVersion;
    write actx.oSBuild;
    delete actx;
end;
```

makePicture

Signature `makePicture(binary: Binary): IJadeAutoPicture;`

The **makePicture** method of the **ActiveXControl** class creates a picture object from the JADE binary specified in the **binary** parameter. A **null** value is returned if an invalid binary is specified.

menuItemCount

Signature `menuItemCount(): Integer;`

The **menuItemCount** method of the **Form** class returns the number of menu items on the form.

menuItems

Signature `menuItems(menuNumber: Integer): MenuItem;`

The **menuItems** method of the **Form** class enables logic to access a reference to the menu items on an active form at run time.

This method returns the active menu item object specified in the **menuNumber** parameter or **null** if there is no specified menu item. The method returns an object of type **MenuItem**, which enables the properties of a menu item to be accessed.

The method in the following example examines all menu items on a form.

```
vars
    mnu  : MenuItem;
    indx : Integer;
begin
    foreach indx in 1 to menuItemCount do
        mnu := menuItems(indx);
        if mnu.name = "Item1" then
            ...                   // do some processing here
        endif;
    endforeach;
end;
```

move

Signature

```
move(x:      Real;
     y:      Real;
     width:  Real;
     height: Real);
```

The **move** method of the **Window** class enables logic to move and size a form or control from a single method call.

The position values for a control are in units, determined by the **scaleMode** property of the parent control. Form units are in pixels. The **resize** or **formMove** event for a form is not generated when the **move** method is called.

It is far more efficient to call the **move** method than to set more than one of the **left**, **top**, **width**, **height**, **clientWidth**, or **clientHeight** properties, as each change to these properties results in a call to the **move** method, changing only a specific property.

Each call potentially results in a move and resize taking place. If the window being affected has other controls aligned to it, these are also resized or moved each time.

The following examples show the use of the **move** method.

```
if global.appCount = 1 then
    self.move(0, 0, self.width, self.height);
    label3.backColor := Yellow;
    bOpen.value      := true;
else
    self.move(400, 0, self.width, self.height);
    label3.backColor := Red;
    bListen.value    := true;
endif;

pictureEnlarged_mouseMove(pict: Picture input; button: Integer;
                          shift: Integer; x: Real; y: Real) updating;
begin
    if movePic = true then
        pictureEnlarged.move(pictureEnlarged.left + (x - moveX),
                             pictureEnlarged.top + (y - moveY),
                             pictureEnlarged.width,
                             pictureEnlarged.height);
        pictureEnlarged.refreshNow;
    endif;
end;
```

moveCaret

Signature

```
moveCaret(action: Integer;
          param1: Integer;
          param2: Integer): Integer;
```

The **moveCaret** method of the **JadeTextEdit** class moves the caret or visible text in the text editor. In the **action** parameter, use the **JadeTextEdit** class constants:

- **MVCRT_VIEWCARET** (1) to force the caret into view and return zero (**0**). The values of the**param1** and **param2** parameters must be zero (**0**)

- **MVCRT_VIEWSELECTION** (2) to force the start of the selection into view and return zero (**0**)

- **MVCRT_WORDEND** (3) to force the caret to the end of the current word and return the new caret position
- **MVCRT_WORDSTART** (4) to force the caret to the start of the current word and return the new caret position

The **param1** and **param2** parameters are reserved for future use and should be set to zero (**0**).

The code fragment in the following example shows the use of the **moveCaret** method.

```
jteSource.selStart := 5600;
jteSource.selLength := 100;
jteSource.moveCaret(JadeTextEdit.MVCRT_VIEWSELECTION, 0, 0);
jteSource.setFocus();
return;
```

moveColumn

Signature
```
moveColumn(src: Integer;
           dst: Integer);
```

The **moveColumn** method of the **Table** class can be used to move a column of the current sheet of a table. The following example of the **moveColumn** method moves column **4** to column **2**. Column **2** becomes column **3**, and column **3** becomes column **4**.

```
table1.moveColumn(4, 2);
```

The following example of the **moveColumn** method moves column **2** to column **4**. Column **3** becomes column **2**, and column **4** becomes column **3**.

```
table1.moveColumn(2, 4);
```

The current column is adjusted if that column is affected.

The methods in the following examples show the use of the **moveColumn** method.

```
comboBoxColMoveTo_click(combobox: ComboBox input) updating;

theTable_dragDrop(table: Table input;
                  win:   Window input;
                  x, y:  Real) updating;
// Dropping a column on the table causes the dragged column to be moved
// before or after the column it is dropped on, based on the move direction
vars
    count : Integer;
begin
    if win = table then
        count := convertPositionToColumn(x);
        if count <> 0 then
            win.Table.moveColumn(win.Table.column, count);
        endif;
        win.dragMode := DragMode_None;
    endif;
end;

begin
    if selectedColumn <> null then
        table1.moveColumn(selectedColumn, comboBoxColMoveTo.listIndex);
        selectedColumn := null;
        table1.clearAllSelected;
```

```
            buttonSort.setFocus;
        else
            app.msgBox("You must select a column", "No column selected!",
                       MsgBox_OK_Only);
            return;
        endif;
    end;
```

moveMdiClient

Signature
```
moveMdiClient(left:   Real;
              top:    Real;
              width:  Real;
              height: Real);
```

The **moveMdiClient** method of the **Form** class can be used to position the client window, as required.

When a form is built as an MDI frame, it also automatically creates a child client window that covers the non-border area of the frame. Child MDI forms are placed inside this client window.

If the MDI frame is defined with controls, this client window may hide these controls. By default, an attempt is made to position the client window into the largest rectangular area that does not overlap any of the controls. This process works for most situations when the controls are positioned at the edges of the form client area; for example, a status line or toolbar control.

Use the **moveMdiClient** method if this positioning is not satisfactory. This method would normally be called from the **resize** event of the form. When this method has been called for an MDI frame, no automatic positioning of the client is attempted again, as this remains your responsibility.

The default positioning is performed only during the **resize** event of the MDI frame. Changing the visibility or position of a control belonging to the MDI frame has no impact on the position of the MDI client window.

The method in the following example positions the MDI client window below a toolbar control positioned at the top of the client area.

```
form_resize();
begin
    toolBar.width           := clientWidth;
    toolBar.bevelOuterWidth := 1;
    toolBar.bevelInnerWidth := 1;
    // Set size of the mdi client window
    moveMdiClient(0, toolBar.height, clientWidth, clientHeight
                  - toolBar.height);
end;
```

moveRow

Signature
```
moveRow(src: Integer;
        dst: Integer);
```

The **moveRow** method of the **Table** class can be used to move a row of the current sheet of a table. The current row is adjusted if that row is affected. The **src** parameter specifies the source row (that is, the row to be moved) and the **dst** parameter specifies the destination of the moved row. The code fragment in the following example of the **moveRow** method moves row **4** to row **2**. Row **2** becomes row **3**, and row **3** becomes row **4**.

```
table1.moveRow(4, 2);
```

The code fragment in the following example of the **moveRow** method moves row **2** to row **4**. Row **3** becomes row **2**, and row **4** becomes row **3**.

```
table1.moveRow(2, 4);
```

The method in the following example shows the use of the **moveRow** method.

```
buttonFind_click(btn: Button input) updating;
vars
    row : Integer;
begin
    table1.clearAllSelected;
    if textBoxFind.text <> null then
        table1.column  := comboBoxColumns.listIndex;
        foreach row in 1 to table1.rows do
            table1.row := row;
            if table1.text = textBoxFind.text then
                table1.moveRow (row, 2);
                table1.selected := true;
                return;
            endif;
        endforeach;
        app.msgBox("No match found", " ", MsgBox_OK_Only);
        return;
    else
        app.msgBox("You must enter something to search for", "No search
                   name entered", MsgBox_OK_Only);
        return;
    endif;
end;
```

newFile

Signature `newFile(str: String);`

The **newFile** method of the **MultiMedia** class creates a new file for the device specified in the **str** parameter.

If the **useDotNetVersion** property is set to **true**, the **newFile** method is not available and it generates exception 1068 (*Feature not available in this release*).

This method is normally used only before commencing a recording process. See also the **record** and **save** methods.

newIndex

Signature `newIndex(): Integer;`

The **newIndex** method of the **ComboBox** or **ListBox** class returns the index of the item most recently added to a combo box or list box control, or the index of the item that most recently had its text changed by using either the **itemText** or **text** property if the value of the **sorted** property is **true**. Use the **newIndex** method with sorted lists when you need a list of values that correspond to each item in the **itemObject** or **itemData** property array. As you add an item in a sorted list, that item is inserted alphabetically in the list.

The **newIndex** method tells you where the item was inserted, so that you can set a corresponding value in the **itemData** or **itemObject** property array for the item at the same index. This value is also returned by the **addItem** method. The **newIndex** method returns **-1** if there are no items in the list or if an item has been deleted since the last item was added.

When the text of an item is altered, the item may be repositioned in the list because of the **sorted** property. The new position of the item can be obtained by using the **newIndex** method, regardless of whether or not the entry shifted position.

The method in the following example shows the use of the **newIndex** method.

```
loadListBox() updating;
vars
    obj : Object;
begin
    app.mousePointer := self.MousePointer_HourGlass;
    if currentDict <> null then
        foreach obj in currentDict do
            listInstances.addItem(obj.display);
            listInstances.itemObject[listInstances.newIndex] := obj;
            if listInstances.newIndex.isEven then
                listInstances.itemBackColor[listInstances.newIndex] :=
                             LightYellow;
            endif;
        endforeach;
    endif;
    app.mousePointer := self.MousePointer_Arrow;
end;
```

objectPropertiesDialog

Signature `objectPropertiesDialog();`

The **objectPropertiesDialog** method of the **JadeRichText** class invokes the OLE <*COM-object-name*> Properties dialog that displays the properties of the object of the receiver. This dialog enables users to change the view of the object (that is, as an icon view or the full contents view).

objectType

Signature `objectType(): Integer;`

The **objectType** method of the **OleControl** class returns the loaded object type.

The returned object type values are listed in the following table.

OleControl Class Constant	Value	Description
ObjectType_Embedded	1	Embedded object is present
ObjectType_Linked	2	Linked object present
ObjectType_None	0	No object is present

ocxClassName

Signature
```
ocxClassName(): String;
```

The **ocxClassName** method of the **Ocx** control class returns the Windows Registry name of the ActiveX control.

openDialog

Signature
```
openDialog();
```

The **openDialog** method of the **MultiMedia** class displays a modified version of the common File Open dialog that allows a user to select the file to be loaded into the control.

The dialog automatically lists the types of files that are supported by the devices installed on that workstation. It can also sample play the selected file.

If the **useDotNetVersion** property is set to **true**, MP4 file types are included in the list.

Before the dialog is called, any existing device or file associated with the control is closed. On return from the method call, the **mediaName** property contains the name of the selected file or an empty string if the dialog was cancelled.

pageMargins

Signature
```
pageMargins(lft:  Integer;
            rght: Integer;
            tp:   Integer;
            btm:  Integer);
```

The **pageMargins** method of the **JadeRichText** class specifies the left, right, top, and bottom margins, respectively, around a rich text control on a printed page. Specify the parameter values as *twentieths of a point* (twip), which is 1/1440th of an inch (that is, there are 1440 twips in an inch).

All margins are half an inch (720 twips) by default. Although you can set a margin as small as zero (**0**) twips, the actual margin can be no smaller than that used by the printer to which the control is output.

paintIfRequired

Signature
```
paintIfRequired() clientExecution;
```

The **paintIfRequired** method of the **Form** class causes the form to be repainted if a repaint is required; for example, while performing a long processing loop, to ensure that the user presentation is updated after the user brings another application to the front and then returns to JADE.

The JADE executable calls the **DisableProcessWindowsGhosting()** Microsoft API on initiation, which disables Windows' ghosting so that a non-responsive form does not show *Not Responding*, nor does it have the ghosting effect applied by Windows. However, the form will still not automatically paint itself when the presentation thread is busy processing JADE logic. Windows automatically redraws that part of the form or forms that need refreshing from a saved copy of the previously painted image or images.

A **refreshNow** event is performed on that part of the form that needed refreshing. If **paint** events are not required, no action is performed.

The **paintIfRequired** method performs any repainting required without having to perform an **app.doWindowEvents** method call, and therefore does not allow the user interface to be active.

Other than any **paint** events, no other events, notifications, or timer events will be processed as a result of this **paintIfRequired** method call.

Note After a repaint, any clicked button that initiated the processing loop will be drawn in the up position, so it will be important that the user is given a visual indication that the processing is still progressing by some other means; for example, by using the **app.mousePointer := 11** (busy) property value.

You will need to add a call to your logic loop that is regularly performed; for example, call it when the **Cancel** button is checked for a **click** event, when a progress bar update ticks over a percentage, or at a specified number of seconds, as shown in the following code fragment.

```
cancelled := false;
    while not cancelled do
        // ... logic
    // the click event sets the cancelled property
    btnCancel.doWindowEvents(0);
    form1.paintIfRequired();
endwhile;
```

pasteFromClipboard

Signature `pasteFromClipboard(): Integer;`

The **pasteFromClipboard** method of the **JadeTextEdit** class pastes the contents of the clipboard into the control at the current caret position and returns **1**.

The value of the **endOfLineMode** property controls the end-of-line conversion that is performed on the inserted text.

The caret is positioned at the end of the inserted text.

paste_

Signature `paste_();`

The **paste_** method of the **JadeRichText** class programmatically pastes content (for example, text or an image) from the Windows clipboard into the **JadeRichText** control at the current cursor position.

Before you call the **paste_** method, call the **canPaste_** method to confirm there is suitable content available.

If the clipboard does not contain suitable content, the method does not result in any change. (This method is equivalent to selecting the **Paste** command in the context menu of the **JadeRichText** control at run time.)

Applies to Version: 2020.0.01 and higher

pause

Signature `pause();`

The **pause** method of the **MultiMedia** class pauses the playing or recording of the device or file associated with the control.

The method in the following example shows the use of the **pause** method.

```
pause_click(btn: Button input) updating;
begin
```

```
        cd.pause;
    end;
```

To restart the playing or recording, use the **play**, **playFromTo**, or **resume** method.

pictureHeight

Signature `pictureHeight(): Integer;`

The **pictureHeight** method of the **Picture** class returns the height (in pixels) of the current image in the control. The height that is returned is the height of the unstretched image.

pictureType

Signature `pictureType(): Integer;`

The **pictureType** method of the **Picture** class returns the type of picture loaded into a picture control. This method functions on both a runtime picture control and on a static (database) definition of a picture in a picture control. (The **pictureType** method of the **Binary** primitive type also performs the same function, allowing an existing binary object to be examined for picture suitability.)

The values returned by the **pictureType** method are listed in the following table.

Window Class Constant	Value	Description
PictureType_None	0	Not a valid picture
PictureType_Bitmap	1	Bitmap
	2	Not used
PictureType_Icon	3	Icon
PictureType_MetaFile	4	Metafile
PictureType_Cursor	5	Cursor
PictureType_Tiff	6	Tag Image File Format (Tiff)
PictureType_Jpeg	7	Joint Photographic Experts Group (JPEG)
PictureType_Jpeg2000	7	Joint Photographic Experts Group (JPEG)
PictureType_Png	8	Portable Network Graphics (png)
PictureType_Gif	9	Graphics Interchange Format (GIF)

For more details about picture types, see "Supported Picture Image Formats", in the following subsection.

Supported Picture Image Formats

The picture image formats that are handled by JADE using the **pictureType** method of the **Picture** class are listed in the following table.

Picture Format	File Type	Description
Windows bitmap	.bmp	Standard Microsoft bitmap image.
Windows compressed bitmap	.rle	Both RLE4 and RLE8.

Picture Format	File Type	Description
Windows icons	.ico	File can contain single or multiple icons. If multiple icons, JADE selects the icon with the closest color characteristics to the video display of the user. Note that icons can be used as cursors, except that there is no hot spot.
Windows metafile	.wmf	Standard Microsoft Windows metafile.
Windows enhanced metafile	.emf	Standard Microsoft enhanced metafile.
Windows cursor	.cur	Standard Microsoft cursor. (Note that cursors are converted to an icon when used as a picture rather than a cursor.)
Tag Image File Format	.tif	TIFF files (both the Intel and Motorola versions). JADE supports the following TIFF compression styles (*CCITT* is the International Telegraph and Telephone Consultative Committee). ■ Uncompressed ■ CCITT Group 3 1D ■ CCITT Group 3 2D ■ CCITT Group 4 ■ PackBits ■ Modified Huffman encoding
Graphics Interchange Format	.gif	Only **.gif** files containing images are handled. Text records within a **.gif** file are ignored. JADE does not support the ability to convert images to **.gif** files. Animated GIF handling is supported only if the **picture** property of a **Picture** or **JadeMask** control is set to a GIF binary that contains more than one image.
Joint Photographic Experts Group	.jpg	All JPEG files in compliance with the JFIF standard are supported, including compressed JPEG images. Also the JPEG 2000 style images.
Portable Network Graphics	.png	Lossless compression images for greater clarity of graphics.

pictureWidth

Signature `pictureWidth(): Integer;`

The **pictureWidth** method of the **Picture** class returns the width (in pixels) of the current image in the control.

The width that is returned is the width of the unstretched image.

play

Signature
```
play();                  (MultiMedia)

play(delay: Integer);    (Picture)
```

The **play** method of the **MultiMedia** class starts the device playing from the current position in the content.

The following examples show the use of the **MultiMedia** class play method to play a video file.

```
mmcontrol.mediaName := "c:\image.avi";
mmcontrol.play;

play_click(btn: Button input) updating;
begin
    cd.play;
    stop.setFocus;
end;
```

See also the **MultiMedia** class **playFromTo** and **playReverse** methods.

The **play** method of the **Picture** class causes a separate thread to be initiated, which cycles through the pictures in an array defined by the **setPicture** method with a pause between the display of each picture in the array. Use the **delay** parameter to specify in milliseconds the length of the pause between each picture that is displayed. (See also the **stop** method.)

When using the **play** method to display pictures:

- If the value of the **pictureCount** property of the **Picture** control is zero (**0**), this method does nothing.
- If the value of the **pictureCount** property of the **Picture** control is **1**, this method displays the only picture in the picture array.
- If the value of the **pictureCount** property or the **pictureIndex** property of the **Picture** control is changed, the animation stops.
- The picture is not resized for each picture. The current size of the picture is retained.
- The value of the **pictureIndex** property of the **Picture** control accessed by logic does not reflect the index in use by this animation.
- If the value of the **pictureIndex** property of the **Picture** control is zero (**0**), the animation starts with the first picture in the array. If the value of the **pictureIndex** property is non-zero, the animation starts with the next picture in that array.
- The **paint** event is not invoked during the animation process.

When running the JADE application in thin client mode, this method executes on the presentation client by default.

playFromTo

Signature

```
playFromTo(src: Integer;
           dst: Integer);
```

The **playFromTo** method of the **MultiMedia** class plays the content of the current device or file, starting at the position specified in the **src** parameter and stopping when the position specified in the **dst** parameter is reached. Specify the **src** and **dst** parameter positions in units of the **timeFormat** property.

If the **useDotNetVersion** property is set to **true**, the **playFromTo** method is not available and it generates exception 1068 (*Feature not available in this release*).

The **playFromTo** method does not invoke continuous loop play when the value of the **repeat** property is set to **true**.

The method in the following example shows the use of the **playFromTo** method.

```
buttonPlay_click(btn: Button input) updating;
vars
    from, two : Integer;
begin
    from := textBoxFrom.text.Integer;
    two  := textBoxTo.text.Integer;
    if from <> null and two <> null then  // "to" is a JADE reserved word
        multimedia.playFromTo(from, two);
    else
        multimedia.play;
    endif;
end;
```

See also the **play** method.

playReverse

Signature `playReverse();`

The **playReverse** method of the **MultiMedia** class starts the device playing from the current position in the content in the reverse direction.

If the **useDotNetVersion** property is set to **true**, the **playReverse** method is not available and it generates exception 1068 (*Feature not available in this release*).

The **playReverse** method does not invoke continuous loop play when the value of the **repeat** property is set to **true**.

Note Not all devices have the ability to play in reverse.

The method in the following example shows the use of the **playReverse** method.

```
buttonPlayReverse_click(btn: Button input) updating;
vars
    from, two : Integer;
begin
    from := textBoxFrom.text.Integer;
    two  := textBoxTo.text.Integer;
    if from <> null and two <> null then  // "to" is a JADE reserved word
        multimedia.playFromTo(two, from);
    else
        multimedia.playReverse;
    endif;
end;
```

See also the **play** and **playFromTo** methods.

popupMenu

Signature
```
popupMenu(menu: MenuItem;
          x:    Integer;
          y:    Integer);
```

The **popupMenu** method of the **Form** class invokes a popup menu for the form.

The parameters of the **popupMenu** method are listed in the following table.

Parameter	Description
menu	An existing popup menu item of the menu of the form
x	The left position to display the popup menu
y	The top position to display the popup menu

The menu item must be part of the menu of the form and must have subitems. The submenu does not need to be visible in the current form menu. The menu **click** message is not received until the current window activity is finished (that is, until after the calling methods have exited).

A **MenuItem**::**select** event that has the **closed** parameter set to **false** is called for the menu referenced by the **popupMenu** method before the menu is displayed. A **select** event that has the **closed** parameter set to **true** (that is, deselected) for the popup menu is called only when the menu is closed.

Note The **popupMenu** method cannot be invoked from a server method.

The **popupMenu** method can be invoked when the user clicks the right mouse button, as shown in the methods in the following examples.

```
theTable_mouseDown(table:  Table input;
                   button: Integer;
                   shift:  Integer;
                   x, y:   Real) updating;
// left button  - go into drag mode
// right button - popup menu of table columns
begin
    xColSave := convertPositionToColumn(x);
    yRowSave := convertPositionToRow(y);
    if xColSave <> 0 and yRowSave <> 0 then
        if theTable.row = 1 then
            if button = Window.MouseButton_Left then
                table.dragMode := DragMode_Drag;
            else
                popupMenu(mPopupColumnList, (x + tableGroup.left +
                          table.left).Integer, (y + tableGroup.top +
                          table.top).Integer);
            endif;
        endif;
    endif;
end;

table1_mouseDown(table:         Table input;
                 button, shift: Integer;
                 x, y:          Real) updating;
begin
    if button = Window.MouseButton_Left then
        dragMode := DragMode_Drag;
    else
        popupMenu(menuItemAction, x.Integer, y.Integer);
    endif;
    selectedColumn := table.column;
end;
```

For more details, see "Windows Events and JADE Events", later in this document.

positionCollection

Signature
```
positionCollection(obj: Object;
                   row: Integer);
```

The **positionCollection** method of the **ListBox** class positions the collection attached to the list box control to an object in that collection and to a position within the list box.

Use the **obj** parameter to specify the object to be positioned and the **row** parameter to specify the visible row in which to position that object.

You can use this method to scroll through an existing collection display by specifying the new position of an object within the current display. For example, the following code fragment scrolls the collection view so that the second item is positioned in the top row.

```
listBox1.positionCollection(listBox1.itemObject[2], 1);
```

When using the **positionCollection** method:

- The specified row may not be the resulting displayed row if the required control cannot display sufficient entries to fill the list box.
- The **listIndex** property is set to the row of the object.
- If the specified object is not a visible member of the collection in the list box, the display starts from the first visible collection entry.
- If the specified row is:
  - Less than **1**, **1** is assumed.
  - Greater than the number of rows in the list box, the number of visible rows is assumed.

positionLeft

Signature
```
positionLeft(): Integer;   (Table)

positionLeft(index: Integer): Integer;   (ListBox)
```

The **positionLeft** method of the **Table** control returns the displayed left position of the current cell in pixels, relative to the left of the client area of the table (the area inside borders).

The **sheet**, **column**, and **row** properties define the current cell. If the current cell is not visible, one or both of the **positionLeft** or **positionTop** methods returns **-1**. The **positionLeft** method of the **ListBox** control returns the displayed left position in pixels of the start of the text in the list box entry specified in the **index** parameter, relative to the client area of the **ListBox** control (that is, the area inside borders).

If the requested list entry is not valid, the **positionLeft** method returns **-1**.

Note The indicated position is for the text of the list entry and not for any pictures displayed before the text. (Use the **getListIndex** method to return the index of the displayed list entry corresponding to the specified **x** and **y** positions.)

The method in the following example shows the use of the **positionLeft** method to return the table column whose left and width position covers the **x** coordinate that is passed.

```
convertPositionToColumn(xPos: Real): Integer updating;
vars
    originalColumn : Integer;
    ix             : Integer;
begin
    originalColumn := theTable.column;
    ix             := theTable.leftColumn;
    while ix <= theTable.columns do
        theTable.column := ix;
        if theTable.positionLeft <= xPos then
            if xPos <= (theTable.positionLeft + theTable.columnWidth[ix])
                        then
                theTable.column := originalColumn;
                return ix;
            endif;
        endif;
        ix := ix + 1;
    endwhile;
    theTable.column := originalColumn;
    return 0;
end;
```

positionTop

Signature `positionTop(): Integer; (Table)`

`positionTop(index: Integer): Integer; (ListBox)`

The **positionTop** method of the **Table** control returns the displayed top position of the current cell in pixels, relative to the top of the client area of the table (the area inside borders). The **sheet**, **column**, and **row** properties define the current cell.

If the current cell is not visible, one or both of the **positionLeft** or **positionTop** methods returns **-1**.

The **positionTop** method of the **ListBox** control returns the displayed top position in pixels of the list box entry specified in the **index** parameter, relative to the client area of the **ListBox** control (that is, the area inside borders). If the requested list entry is not visible or the specified index is not valid, the **positionTop** method returns **-1**. (Use the **getListIndex** method to return the index of the displayed list entry corresponding to the specified **x** and **y** positions.)

print

Signature
```
print(docName:          String;
      selection:        Boolean);
```

The **print** method of the **JadeRichText** class outputs the contents of the rich text control to the printer of the application (that is, **app.printer**).

The **print** method parameters are listed in the following table.

Parameter	Specifies …
docName	The output name used in the print queue.
selection	Whether the whole control or only the selected portion of the control is printed. As the whole control is printed by default, set this parameter to **true** if you want to print the selected portion only.

processInputFromWeb

Signature `processInputFromWeb(reply: String);`

The **processInputFromWeb** method of the **Ocx** control or the **ActiveXControl** class is the template method that processes ActiveX controls used on Web pages.

The **reply** parameter may return a value from the submit operation.

If you want to process ActiveX controls used on Web pages, you must reimplement this method in your own control class.

record

Signature `record();`

The **record** method of the **MultiMedia** class begins recording content at the current position of the content of the device, and overwrites existing data for the duration of the recording. The function that the device performs during recording depends on the characteristics of the device.

A device that uses files (for example, a waveform audio device) sends data to the file during recording. A device that does not use files (for example, a video-cassette recorder) receives and externally records data on another medium.

Notes Not all devices support recording.

If the **useDotNetVersion** property is set to **true**, the **record** method is not available and it generates exception 1068 (*Feature not available in this release*).

The **mediaData** property does not contain the recorded data.

See also the **newFile**, **canRecord**, and **save** methods.

recordReplay

Signature
```
recordReplay(action : Integer;
             options: Integer;
             buffer : String io): Integer;
```

The **recordReplay** method of the **JadeTextEdit** class records or replays keystrokes.

The value of the action parameter must be one of the following new **JadeTextEdit** class **Integer** constants.

Class Constant	Description
KMACRO_GETCOMMANDS	Returns the list of available macro commands in the buffer.
KMACRO_GETTEMPORARY	Returns the current temporary macro in the buffer.
KMACRO_PLAYTEMPORARY	Plays the current temporary macro.
KMACRO_PLAYTEXT	Plays the macro text in the buffer without overwriting the current temporary macro.
KMACRO_RECORDSTART	Clears the temporary macro and begins recording keystrokes for the current temporary macro.
KMACRO_RECORDSTOP	Stops recording keystrokes.
KMACRO_SETTEMPORARY	Replaces the current temporary macro with the contents of the buffer. If the value of the **options** parameter is **1**, a syntax check only is done.

The **recordReplay** method returns zero (**0**) if no syntax or replay errors occurred; otherwise it returns the positive character offset of the start of the macro line in error. It can also return a negative result value, indicating JADE runtime errors; for example:

- 15650 Macro halted - caret moved outside document
- 15651 Macro halted - no match for find

Embedded macros are not allowed. You cannot record or play another macro when recording is in progress, nor can you record or play another macro while a macro is playing.

Macro replay halts when the cursor is moved outside the document (method source editor pane) limits and when a find command has no match.

Note Each **JadeTextEdit** control instance has its own private temporary macro.

redo

Signature `redo(): Boolean;` (JadeRichText)

`redo(): Integer;` (JadeTextEdit)

The **redo** method of the **JadeRichText** class reapplies the last operation that was undone in the receiver control and returns **true** if the redo operation was successful or it returns **false** if the redo operation failed. You can redo up to 100 edit or format actions in a rich text control.

The **redo** method of the **JadeTextEdit** class reapplies the last operation that was undone in the receiver control and returns zero (**0**). There is no limit to the number of redo actions in a text edit control.

refresh

Signature `refresh();`

The **refresh** method of the **Window** class forces a repaint of a form or control. Windows issues this paint when there are no other types of Windows messages waiting to be processed.

Use this method to force a complete repaint of a form or control when you want a form to display completely while another form is loading, or when you want to update the contents of a control.

Generally, painting a form or control is handled automatically. The paint occurs at a point when other events have completed. However, there may be situations where you want the form or control to be updated.

If the **refresh** method is called in the **paint** event of the window, it causes an exception to be raised, as this method calls the **paint** event again, and so on.

The method in the following example shows the use of the **refresh** method.

```
listBoxScrollBar_scrolled(scroll:    ScrollBar input;
                          scrollBar: Integer) updating;
vars
    count : Integer;
begin
    listBoxRight.clear;
    textBoxRightStart.text := theArray[listBoxScrollBar.value].Product.name;
    app.myCompany.allProducts.startKeyGeq(textBoxRightStart.text, iter);
    while count < listBoxRight.lines and iter.next(myProduct) do
        count := count + 1;
        listBoxRight.addItem(myProduct.name);
    endwhile;
epilog
    textBoxRightStart.refresh;
end;
```

refreshEntries

Signature `refreshEntries(obj: Object);`

The **refreshEntries** method of the **ComboBox**, **ListBox**, and **Table** class refreshes the list of entries when a collection is attached to the control. For the **Table** class, this method applies only to the current sheet of the table.

This method is equivalent to what happens when an update notification is processed. If the **obj** parameter has a non-null value, the display is refreshed so that the requested object is included in the visible portion of the list box or table.

For a combo box or list box, the **listIndex** property is set to the list entry for that object. For a table, the current row of the sheet is set to the row for that entry.

Note Calling this method discards any outstanding automatic notifications that would repeat the refreshing process. (It does not discard any future update notifications.)

You could use this method in the following situations.

- When logic has just added a new entry to the collection that is required to be displayed. If the control performs an automatic update on the display, the notification will not arrive until current event logic completes or an **app.doWindowEvents** is performed, and it cannot be guaranteed that it will arrive immediately if the server is very busy.

  Calling the **refreshEntries** method updates the display and ensures that the new entry is included in the list.

- To force a refresh of the entries displayed when data for an entry or entries has been changed so that it will not result in a notification (the collection is unchanged).

  Calling **refreshEntries** refreshes the displayed list using the updated objects.

refreshNow

Signature `refreshNow();`

The **refreshNow** method of the **Window** class forces an *immediate* repaint or update of a form or control. This paint is completed on return from this method. This immediate repaint can cause problems in situations where the parent of a control is awaiting an ordinary paint and the **clipControls** property is not set. The effects may be that the non-client area (border) of the control is overwritten by the paint of the parent.

Use this method to force a complete repaint of a form or control when you want a form to display completely while another form is loading, or when you want to update the contents of a control.

Generally, painting a form or control is handled automatically. The paint occurs at a point when other events have completed. However, there may be situations where you want the form or control updated immediately.

Use the **refreshNow** method instead of the **doWindowEvents** method of the **Window** class when the object is to update the image only; for example, when displaying a running status.

The **doWindowEvents** method lets in any waiting Windows message, which can result in some unwanted side effects as the processing of the current message has not yet been completed.

Note If a parent of the window that is refreshed still has an outstanding **paint** event, the child window is also painted again after the parent is painted.

The methods in the following examples show the use of the **refreshNow** method.

```
btnOK_click(btn: Button input) updating;
vars
    obj   : Object;
    count : Integer;
begin
    if class01.value then
        if beginNote.value  then
            foreach obj in ClassNoteClass_01.instances do
                classNoteClass_01 := obj.ClassNoteClass_01;
                beginNotification(obj, Any_System_Event,
                                  Response_Continuous, 11199);
                count := count + 1;
            endforeach;
            staLine.caption := count.String & ' notifications set';
        elseif endNote.value then
            foreach obj in ClassNoteClass_01.instances do
                classNoteClass_01 := obj.ClassNoteClass_01;
                endNotification(obj, Any_System_Event);
                count := count + 1;
            endforeach;
            staLine.caption := count.String & ' notifications ended';
        endif;
    endif;
end;

btnInitialize_click(btn: Button input) updating;
vars
    num   : Number;
    count : Integer;
begin
```

```
        // Creates 5000 Number objects & enters them into the numbers collection
        staLine.caption := "Initializing data...";
        staLine.refreshNow;
        beginTransaction;
            foreach count in 1 to 5000 do
                create num;
                num.key := count;
                numbers.add(num);
            endforeach;
        commitTransaction;
        // Resets the status line and enables the two buttons.
        // The initialize button is then disabled.
        staLine.caption       := "";
        btnClientExec.enabled := true;
        btnServExec.enabled   := true;
        btnInitialize.enabled := false;
    end;
```

For more details, see "Windows Events and JADE Events", later in this document.

registerFormKeys

Signature `registerFormKeys(array: IntegerArray);`

The **registerFormKeys** method of the **Form** class enables you to establish the entire set of key codes in which key events of the receiver form are interested. The **array** parameter is populated with all key codes.

The **keyUp**, **keyPress**, and **keyDown** events are available for forms and for some controls (for example, **TextBox** and **Table** controls). By default, key events are received by the form and the control that has focus. The **registerFormKeys** method applies only to key event methods implemented for the form. The **registerKeys** method applies only to key event methods implemented for the control.

By default, if a form has key event methods defined, those key events are sent for any key event for any control on that form. In most situations, the form key events are interested only in specific keys (for example, the Tab key, arrow keys, and function keys).

After calling the **registerFormKeys** method with a key code list, the form key events are called on the form only if the key that is pressed is in the supplied list. This results in not having to call the form key events for every key action, which reduces the number of events that must be sent and processed.

Note You can use this process in both standard (fat) client and for thin client mode.

In the **load** method for a form shown in the following example, the Tab and the F2 function keys are registered as the only keys for which the form key events are called.

```
load();
vars
    aray : IntegerArray;
begin
    create aray transient;
    aray.add(9);            // Tab key
    aray.add(113);          // F2 key
    registerFormKeys(aray);
    delete aray;
end;
```

Calling the **registerFormKeys** method with a **null** value or with an empty key code list results in all keys that are pressed being sent to the form key events again. Each call to the **registerFormKeys** method entirely replaces any key code list that is currently in effect.

Note The key codes represent the actual physical keys. Registering the key code of **187** traps both the **+** and the **=** characters because they are the same key.

The **Window** class provides the class constants listed in the following table, which you can add to key code **array** parameter value to indicate that the key event should be sent only if the indicated special keys are also down when a specified key is pressed.

Window Class Constant	Integer Value
RegisterKeys_Alt	(#80000000)
RegisterKeys_Ctrl	(#40000000)
RegisterKeys_Shift	(#20000000)

You can use these class constants with a zero (**0**) key code array value, to indicate that the events for any key pressed are sent when that combination of special keys is also down (except for the special keys themselves). The method in the following example shows the use of these class constants.

```
load();
vars
    aray : IntegerArray;
begin
    create aray transient;
    aray.add(9);
    // Tab key, regardless of whether the Alt, Ctrl, or Shift keys are down
    aray.add(113 + RegisterKeys_Shift);
    // F2 key, only when Shift is also down
    aray.add(114 + RegisterKeys_Ctrl + RegisterKeys_Shift);
    // F3 key, only when Ctrl and Shift keys are both also down
    aray.add(RegisterKeys_Ctrl);
    // Any key pressed while the Ctrl key is down
    aray.add(RegisterKeys_Alt + RegisterKeys_Shift);
    // Any key pressed while Alt and Shift keys are both down
    registerFormKeys(aray);
    delete aray;
end;
```

registerKeys

Signature `registerKeys(array: IntegerArray);`

The **registerKeys** method of the **Control** class enables you to establish the entire set of key codes in which key events of the receiver control are interested. The **array** parameter is populated with all key codes. (For an example of registering keys, see the **Form** class **registerFormKeys** method.) This method applies only to controls that can get the focus and therefore receive key events.

By default, if a control has **keyDown**, **keyPress**, or **keyUp** event methods defined, those key events are sent for any key event for that control. In most situations, the control key events are interested only in specific keys (for example, the Tab key, arrow keys, and function keys).

The **keyUp**, **keyPress**, and **keyDown** events are available for forms and for some controls (for example, **TextBox** and **Table** controls). By default, key events are received by the form and the control that has focus. The **registerFormKeys** method applies only to key event methods implemented for the form. The **registerKeys** method applies only to key event methods implemented for the control.

After calling the **registerKeys** method with a key code list, the control events are called on the control only if the key that is pressed is in the supplied list. This results in not having to call the control key events for every key action, which reduces the number of events that must be sent and processed.

Note You can use this process in both standard (fat) client and for thin client mode.

Calling the **registerKeys** method with a **null** value or with an empty key code list results in all keys that are pressed being sent to the control key events again.

Each call to the **registerKeys** method entirely replaces any key code list that is currently in effect.

Note The key codes represent the actual physical keys. Registering the key code of **187** traps both the **+** and the **=** characters because they are the same key.

The **Window** class provides the class constants listed in the following table, which you can add to key code **array** parameter value to indicate that the key event should be sent only if the indicated special keys are also down when a specified key is pressed.

Window Class Constant	Integer Value
RegisterKeys_Alt	(#80000000)
RegisterKeys_Ctrl	(#40000000)
RegisterKeys_Shift	(#20000000)

You can use these class constants with a zero (**0**) key code array value, to indicate that the events for any key pressed are sent when that combination of special keys is also down (except for the special keys themselves). For an example of the use of these class constants, see the **Form** class **registerFormKeys** method.

registerWindowMsg

Signature
```
registerWindowMsg(msgName:    String;
                  methodName: String): Integer;
```

The **registerWindowMsg** method of the **Form** class registers a Windows message with the JADE GUI environment so that when the message specified in the **msgName** parameter is received by that form, the method specified in the **methodName** parameter is called on the form object.

The parameters of the **registerWindowMsg** method are listed in the following table.

Parameter	Description
msgName	String used by the JADE GUI environment to call the **RegisterWindowMessage** Windows Application Programming Interface (API). This function defines a new Windows message number that is guaranteed to be unique for that workstation environment based on the specified name. Each subsequent call to **RegisterWindowMessage** by any application on that workstation returns the same message number.

Parameter	Description
methodName	The name of the method that is called when the JADE GUI environment receives a message with the specified number from Windows for that form. The method must be defined and expect a single **Integer** parameter. The **wParam** parameter of the received Windows message is passed as the **Integer** parameter to the method. (For additional information about **PostMessage** or **SendMessage**, refer to the Windows documentation.)

The **registerWindowMsg** method returns the message number (**msgNum**) generated by the **msgName** parameter call on the **RegisterWindowMessage** API. For example, the following code fragment results in **form1.specialCallBack(val: Integer)** being called when a Windows message of **msgNum** is received by **form1** in the GUI environment, where **val** is the value passed in the **wParam** value of the Windows message.

```
msgNum := form1.registerWindowMsg("My Special Call Back", "specialCallBack");
```

The **PostMessage** or **SendMessage** Windows call to generate the message must use the **hwnd** parameter of the form. Using an external method definition, for example, the message could be posted to JADE as follows.

```
call postMessage(form1.hwnd, msgNum, val, 0);
```

Note Multiple forms can register the same message name and each form can register multiple messages using different names.

Calling the **registerWindowMsg** method with the same message name a second time replaces the previous registration.

To remove the registered message, call the **registerWindowMsg** method with a null value (**""**) in the **methodName** parameter.

releaseMouse

Signature `releaseMouse();`

The **releaseMouse** method of the **Window** class releases the mouse capture from a window and restores normal mouse input processing.

A window that has captured the mouse receives all mouse input.

This method is called by an application after calling the **captureMouse** method.

removeItem

Signature `removeItem(index: Integer);`

The **removeItem** method of the **ComboBox**, **ListBox**, and **Table** classes removes an item from a combo box or list box control or removes a row from a table control for the current sheet at run time. For a combo box or list box control, if the item to be removed has subitems then these are removed as well.

The **index** parameter is an integer value that represents the position within the control of the item or row to be removed. The value of the index is **1** for the first row in a table control and for the first item in a list box or combo box control.

The following examples show the use of the **removeItem** method.

```
comboBoxColMoveTo.removeItem(comboBoxColMoveTo.listCount);

instancesTable.removeItem(row);    // remove from display
```

```
btnDelete_click(button: Button) updating;
vars
    emp : Employee;
begin
    emp := listBoxEmps.listObject.Employee;
    if emp.allEmployees.size <> 0 then
        app.msgBox("Employee has employees - cannot delete", "Error",
                   MsgBox_OK_Only);
        return;
    endif;
    beginTransaction;
    delete emp;
    listBoxEmps.removeItem(listBoxEmps.listIndex);
    commitTransaction;
end;
```

removeSystemTrayEntry

Signature `removeSystemTrayEntry();`

The **removeSystemTrayEntry** method of the **Form** class removes the system tray entry for the form. This method does nothing if there is no system tray entry (defined by using the **Form** class **setSystemTrayEntry** method). See also the **hasSystemTrayEntry** method.

removeWebEventMapping

Signature `removeWebEventMapping(eventName: String): Boolean;`

The **removeWebEventMapping** method of the **Window** class removes the Web event mapping specified in the **eventName** parameter from the receiver. For example, the following code fragment removes the **lostFocus** event from the **textBox1** control.

```
textBox1.removeWebEventMapping("onLostFocus");
```

This method returns **false** if the specified event does not exist.

replace

Signature

```
replace(find:    String;                    (JadeRichText)
        replace: String;
        start:   Integer;
        finish:  Integer;
        options: Integer): Integer;

replace(replacement:    String;             (JadeTextEdit)
        interpretation: Integer): Integer;
```

The **replace** method of the **JadeRichText** class replaces text in the receiver. (For an example of the use of this method, see "JadeRichText Control Method Example", earlier in this document.)

The **replace** method returns the number of replacements that are made.

The **JadeRichText** class **replace** method parameters are listed in the following table.

Parameter	Description
find	Text to be located.
replace	The text that replaces instances of text in the receiver matching that specified by the **find** parameter.
start	Start of the search range, as a character index into the text or specified as **Find_BeginningOfText**.
finish	End of the search range, as a character index into the text or specified as **Find_EndOfText**.
options	One or more of the following values, separated by the plus symbol (**+**). ■ **Find_MatchCase**, which finds only text with the matching case; otherwise search is case-insensitive. ■ **Find_WholeWord**, which finds only whole words; otherwise parts of words satisfy the search. ■ **Find_SearchBack**, which searches backwards through the text; otherwise the search direction is forward to the end of the text. ■ **Replace_ReplaceAll**, which replaces all occurrences of the located (found) text in the receiver.

Instances of text in the receiver matching the text specified in the **find** parameter are replaced by text specified in the **replace** parameter. The **replace** method of the **JadeTextEdit** class replaces the most recent find match in the text editor with the replacement text specified in the **replacement** parameter.

Use the **interpretation** parameter to specify one of the following **JadeTextEdit** class constants.

- **FIND_INTERP_NONE** (0), to replace text as is
- **FIND_INTERP_POSIXREGEXPR** (3), to replace text after converting backslash control characters and inserting tagged regions (**\n**, where the **n** value is in the range **1** through **9**)
- **FIND_INTERP_REGEXPR** (2), to replace text after converting backslash control characters and inserting tagged regions
- **FIND_INTERP_UNSLASH** (1), to replace text after converting backslash control characters

Each replace action in the text editor can be undone separately. A successful replace action automatically calls the **findAgain** method and returns the result of that find. If no recent find action was performed, the replace action is not performed.

For more details about interpretation, see the **find** method.

replaceAll

Signature

```
replaceAll(match:          String;
           replacement:    String;
           range:          Integer;
           direction:      Integer;
           caseSensitive:  Boolean;
           wholeWord:      Boolean;
           wordStart:      Boolean;
           interpretation: Integer): Integer;
```

The **replaceAll** method of the **JadeTextEdit** class searches for specified text in the text editor and replaces it with the specified replacement text.

This method performs a single undo action, which is a looped **find** method call followed by a **replace** method call until the replace action returns a *not found* result.

The **replaceAll** method parameters are listed in the following table.

Parameter	Description
match	Mandatory value, which specifies the text to be located.
replacement	The text that replaces instances of text in the receiver matching that specified by the **match** parameter.
range	Range of search. One of **FIND_RANGE_ALL** (**0**), **FIND_RANGE_CARET** (**1**), or **FIND_ RANGE_SELECTION** (**2**). The **FIND_RANGE_SELECTION** (**2**) can be a stream selection, line selection, or rectangular selection.
direction	Direction in which to search. Specify **-1** to search backwards towards the top of the text editor, zero (**0**) or **+1** to search forwards towards the bottom of the text editor. The search is always performed starting at the position closest to the top of the text.
caseSensitive	If **true**, finds only text with the matching case. If **false** (the default), the search is case-insensitive.
wholeWord	If **true**, finds only whole words. If **false** (the default), finds parts of words that satisfy the search.
wordStart	If **true**, the match must occur at the start of a word (that is, the matched text must be preceded by a non-word character).
interpretation	One of the following values, represented by **JadeTextEdit** class constants. ■ **FIND_INTERP_NONE** (0), to search for text as is ■ **FIND_INTERP_POSIXREGEXPR** (3), to search for a POSIX regular expression ■ **FIND_INTERP_REGEXPR** (2), to search for a regular expression ■ **FIND_INTERP_UNSLASH** (1), to search for backslash control characters

For details about interpretation, see the **find** method.

The caret is positioned as close as possible to its apparent original location.

The **replaceAll** method returns the number of replacements that were made. If the selection is empty and **FIND_ RANGE_SELECTION** is specified, it returns **-2**.

The code fragment in the following example shows the use of the **replaceAll** method to replace all occurrences of **"fred"** with **"harry"** between the caret and the end of the text.

```
count := self.theJadeTextEdit.replaceAll("fred", "harry",1 /*range*/,
            0/*direction*/
            false/*caseSensitive*/,
            false/*wholeWord*/,
            false/*wordStart*/,
            0 /*interpretation*/ );
if count < 1 then
    app.msgBox("No matches","Replace text",0);
endif;
```

replyAsBinary

Signature `replyAsBinary(header:  String;`
`              message: Binary): String;`

The **replyAsBinary** method of the **Form** class returns the Binary message contained in the **message** parameter to the Web browser without modification. Use this method to send a binary reply without UTF-8 encoding to a Web message.

The message uses the **String** value in the **header** parameter as the HTTP headers. The headers are encoded with UTF-8, as is expected for HTTP headers.

The code fragment in the following example shows the use of the **replyAsBinary** method in the **MyWebForm** class.

```
message := f.readBinary(f.fileLength);
header := 'Content-Type: application/pdf' & CrLf &
          'Content-Disposition: attachment;filename=' &
                              downloadFile.fileName.stripToFileName & CrLf &
          'Content-Length: ' & message.length.String & CrLf & CrLf;
replyAsBinary(header, message);
```

resetAllHyperlinks

Signature `resetAllHyperlinks();`

The **resetAllHyperlinks** method of the **Table** class clears all HyperText links that were set using the **setHyperlinkCell** method.

resetFirstChange

Signature `resetFirstChange();`

The **resetFirstChange** method of the **Form** class resets the first change status of the referenced form and all **TextBox**, **JadeRichText**, and **JadeEditMask** controls on the form. After calling the **resetFirstChange** method, the next change made to a control on the form listed in the following table again generates a **firstChange** event for the form.

Control Class	Change
TextBox	Change to the value of the **text** property

Control Class	Change
JadeEditMask	Change to the value of the **text** property
JadeRichText	After a user change but before the **change** event (same rules as those that apply to the **JadeRichText** class **firstChange** event)
CheckBox	Change to the value of the **value** property
OptionButton	Change to the value of the **value** property
ComboBox	Different entry selected (this does not apply to the text box portion of the combo box unless the **text** property change causes a different list entry to be selected)
ListBox	Different list entry selected
Table	Cell updated by the **Control** class **automaticCellControl** property, including default **inputType** property entry of **CheckBox**, **TextBox**, **JadeEditMask**, and **ComboBox** controls (but not when a different row or column is selected)

Similarly, the next change made on each **TextBox**, **JadeRichText**, and **JadeEditMask** control on the form again generates a **firstChange** event on that control.

The **resetFirstChange** method of the **Control** class resets the first change status of the control and all children of the control. As only the **TextBox**, **JadeRichText**, and **JadeEditMask** controls have a **firstChange** event, all other controls ignore the method other than to call the method on any children. The code fragment in the following example resets the **firstChange** event status on the text box.

```
textBox1.resetFirstChange();
```

The code fragment in the following example ignores the method for the group box, but resets the **firstChange** event status for any child (or children of children) **TextBox**, **JadeRichText**, or **JadeEditMask** controls.

```
grpAddress.resetFirstChange();
```

You could use the **resetFirstChange** method, for example, so that when a user clicks an update button and the logic associated with that button performs the required updating, the form remains active. Calling **resetFirstChange** will reset the **firstChange** status of the form and its controls to the same state as when the form was loaded. The next change made to a control on the form and to any **TextBox**, **JadeRichText**, and **JadeEditMask** controls on that form again generates a **firstChange** event and therefore signals that a new change has been made to the data. (To accomplish this for **TextBox**, **JadeRichText**, and **JadeEditMask** controls without calling the **resetFirstChange** method, you must use logic to reset the text of all of those controls.)

Note The **resetFirstChange** method for the **TextBox**, **JadeEditMask**, **JadeRichText**, **Control**, and **Form** classes is not available from a Web browser.

resetHyperlinkCell

Signature
```
resetHyperlinkCell(hyperlinkRow:    Integer;
                   hyperlinkColumn: Integer);
```

The **resetHyperlinkCell** method of the **Table** class clears the HyperText link that was previously set by the **setHyperlinkCell** method for the row specified in the **hyperlinkRow** parameter and the column specified in the **hyperlinkColumn** parameter.

resort

Signature `resort();`

The **resort** method of the **Table** class resorts the contents of the current sheet of a table control. This method uses the sort parameters specified in the **sortColumn**, **sortAsc**, and **sortCased** properties.

Use this method, for example, to resort the table by a particular column when the user clicks on the fixed column cell. The logic required to perform this action is shown in the code fragment in the following example.

```
if table1.row = 1 then
    table1.sortColumn[1] := column;
    table.resort;
endif;
```

The resorting of a table may cause rows of the table to be reordered. The current row of the table is adjusted if it is affected.

Notes The **resort** method is ignored if no columns are set.

When the text of a sorted column changes, the automatic sorting of rows occurs only when the **Table** class **addItem** method adds a new row or the **Table** class **resort** method is used.

The method in the following example shows the use of the **resort** method.

```
vars
    indx  : Integer;
    count : Integer;
begin
    if table.row = 1 and table.column > 1 then
        table.sortColumn[1] := table.column;
        table.resort;
        count := table.rows - 1;
        indx  := 1;
        while indx <= count do
            table.setCellText(indx + 1, 1, indx.String);
            indx := indx + 1;
        endwhile;
        table.row    := 1;
        table.column := 1;
        table.clearAllSelected;
    endif;
end;
```

restyleText

Signature `restyleText(): Integer;`

The **restyleText** method of the **JadeTextEdit** class clears all styling information from the text editor then recalculates the styling of the text using the current language setting. Call this method to recalculate text styles and fold points after changing the programming language. This method returns zero (**0**).

Calling this method without changing the current language has the useful side effect of expanding all collapsed fold points.

The code fragment in the following example shows the use of the **restyleText** method to display the text with highlighting defined by a newly selected language.

```
jteSource.language := JadeTextEdit.SCLEX_CPP;
jteSource.applySettings();
jteSource.restyleText();
```

resume

Signature `resume();`

The **resume** method of the **MultiMedia** class resumes playback or recording content from the paused mode. When playing content, this method is usually equivalent to using the **play** method.

rgb

Signature
```
rgb(red:   Integer;
    green: Integer;
    blue:  Integer): Integer;
```

The **rgb** method of the **JadeTextEdit** class returns an Integer value containing the encoded red, green, and blue color values specified in the respective **red**, **green**, and **blue** parameters.

To determine **Integer** value of a color from the RGB values:

```
int:= RedValue + (GreenValue * 256) + (BlueValue * 256 * 256);
```

The valid range for a normal RGB color is **0** through **16,777,215** (#FFFFFF). The high byte of an integer in this range equals **0**; the lower three bytes (from least to most significant byte) determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is **128**, JADE uses the system colors, as defined in the Control Panel of the user.

The code fragment in the following example shows the use of the **rgb** method.

```
jte.setLinemarkAttributes(JadeTextEdit.MARKER_JAD_LINEMARK, 0,
                          jte.rgb(255,180,180),
                          jte.rgb(128,255,255));
```

save

Signature `save(filename: String);`

The **save** method of the **MultiMedia** class saves the content currently used by a device into the file specified in the **filename** parameter.

This method is mostly used after recording content, allowing the content to be saved into a file.

Notes Not all devices support the saving of multimedia content to a file.

If the **useDotNetVersion** property is set to **true**, the **save** method is not available and it generates exception 1068 (*Feature not available in this release*).

The **mediaData** property does not contain the recorded data.

See also the **canRecord**, **canSave**, **newFile**, and **record** methods.

saveInFile

Signature
```
saveInFile(fileName:  String;
           selection: Integer;
           options:   Integer);
```

The **saveInFile** method of the **JadeRichText** class saves the contents of the receiver to a file.

The **saveInFile** method parameters are listed in the following table.

Parameter	Description
fileName	Existing file in which the contents of the control are to be saved. If you do not specify a path, JADE looks in the current directory and raises an exception if the specified file is not in the directory or if it does not exist.
selection	A **JadeRichText** class constant value that indicates the contents saved to the file, as follows. ■ **SaveInFile_All**, which saves the entire contents of the control to the file. ■ **SaveInFile_Selection**, which saves only the selected text of the receiver to the file.
options	A **JadeRichText** class constant value that indicates the format of the loaded text, as follows. ■ **SaveInFile_PlainText**, which is saved in plain text format. ■ **SaveInFile_UnicodeText**, which is saved as plain Unicode text. ■ **SaveInFile_RTF**, which is saved as rich text format text if the text starts with a valid RTF header sequence (for example, **"{\rtf "**).

savePicture

Signature
```
savePicture(fileName: String);
```

The **savePicture** method of the **ActiveXControl** class saves the image of a picture to the external file specified in the **fileName** parameter, which can be a valid file name or it can be the fully qualified path and name of a valid picture file. A **null** value is returned if a file name is not specified or it is invalid.

saveProperties

Signature
```
saveProperties(persistCtl: Control input);
```

The **saveProperties** method of the **Control** class is called by the Painter to save the properties edited by the property page dialogs of a control. (For more details, see the **hasPropertyPage** method.)

The **saveProperties** method is defined in ActiveX controls so that the ActiveX internal property setting can be used by the JADE Painter. The Painter sends the **saveProperties** message to a transient object, passing a persistent instance of the same class, which is used to store property values.

The application is in transaction state for the duration of the method.

Note If you want your subclassed controls to save their own internal property data, you must reimplement this method so that it can be called by Painter. Your subclassed controls must also call **inheritMethod**, to ensure superclasses save their data.

saveTextToFile

Signature `saveTextToFile(fileName: String): Integer;`

The **saveTextToFile** method of the **JadeTextEdit** class saves the text in the text editor to the fully qualified name of the text file specified in the **fileName** parameter. This file name includes the full path of the JADE application if the file is not located in the default directory. End-of-line sequences in the text buffer are converted to match the platform.

If the file cannot be opened, the **saveTextToFile** method returns the appropriate exception code rather than raising exceptions to report errors (for example, *5003 - Requested file not found*).

If the file write operation was successful, this method returns zero (**0**). In a Unicode system, the file is written with the **kind** property of the **File** object set to **Kind_Unicode_UTF8**, meaning wide characters (for example, **UTF-16** in a Windows environment); otherwise, it is written with the **kind** value of **Kind_ANSI**.

You can use this method to determine if the **JadeTextEdit** support library (that is, the **jadedit** DLL) is present, by calling it with the **fileName** parameter set to null (**""**). If the library is present and it is usable, the **saveTextToFile** method returns error **JTE_FILENAME_EMPTY**.

screenToWindow

Signature
```
screenToWindow(x: Real io;
               y: Real io);
```

The **screenToWindow** method of the **Window** class converts an absolute screen position into a position relative to the top and left of the window. For example, this method can be used in conjunction with the **windowToScreen** method to calculate a position of one window relative to another.

The method in the following example shows the use of the **screenToWindow** method.

```
vars
    x, y : Real;
begin
    cntrl.windowToScreen(x, y);       // Convert 0, 0 to screen coordinates.
    cntrl.form.screenToWindow(x, y);  // x, y now positioned within the
                                      // form of the control.
end;
```

selectedCount

Signature `selectedCount(): Integer;`

The **selectedCount** method of the **Table** class returns the number of selected cells in the current sheet of a table.

The value that is returned is equivalent to the code fragment in the following example.

```
foreach row in 1 to table1.rows do
    foreach col in 1 to table1.cols do
        if table1.getCellSelected(row, col) then
            count := count + 1;
        endif;
    endforeach;
endforeach;
return count;
```

The **selectedCount** method of the **ListBox** class returns the number of entries selected in the list box.

selectedNext

Signature
```
selectedNext(r: Integer io;
             c: Integer io): Boolean;
```

The **selectedNext** method of the **Table** class returns the next selected cell following the row and column specified in the **r** and **c** parameters, respectively, for the current sheet of a table.

The **selectedNext** method allows logic to step through the selected cells, and it returns **true** if another selected cell is located or **false** if none is located.

The following example steps through all of the selected cells of the current sheet of a table.

```
vars
    row : Integer;
    col : Integer;
begin
    while table1.selectedNext(row, col) do
        ...                                 // do some processing here
    endwhile;
end;
```

selectAll

Signature `selectAll();`

The **selectAll** method of the **JadeTextEdit** class selects all text in the text editor.

sendString

Signature `sendString(str: String): String;`

The **sendString** method of the **MultiMedia** class allows commands to be issued directly to the device driver associated with the **MultiMedia** control. You could perform all of the interfaces to the **MultiMedia** control by using the appropriate commands, if required. The commands are device-dependent, with each device having its own command set. However, there are common commands that are available with any device.

The **sendString** command sends the command specified in the **str** parameter and may return a string containing the reply. An exception is raised if the command is not recognized or is not relevant to the current device. For examples of the commands that are available in a Windows GUI environment, see the "Multimedia Command Strings" section under "Reference", in the *Microsoft Developer Network* product documentation.

If the **useDotNetVersion** property is set to **true**, the **sendString** method is not available and it generates exception 1068 (*Feature not available in this release*).

Some examples of commands that are available are listed in the following table.

Command	Result
mm.sendString("capability device type")	Returns the device type name
mm.sendString("capability has audio")	Returns **true** or **false**
mm.sendString("delete from 23 to 46")	Deletes file data from position 23 through 46
mm.sendString("set audio all off")	Disables all audio output
mm.sendString("set file format mpeg")	Sets the file format for save calls

Command	Result
mm.sendString("status media present")	Returns **true** or **false** to indicate if media present

The method in the following example shows the use of the **sendString** method.

```
trackLength(track: Integer): Integer;
vars
    str : String;
begin
    str := cd.sendString("status length track " & track.String);
    return str.Integer;
end;
```

setApplicationSkin

Signature `setApplicationSkin(skinapp: JadeSkinApplication);`

The **setApplicationSkin** method of the **Form** class sets the skin for a specific form and its controls.

When you call this method to set a skin for a form and all of its controls, the skins used to draw the form and its controls are from the application skin specified in the **skinapp** parameter and any skin set by calling the **Application** class **setApplicationSkin** is ignored for this form and its controls.

If the application skin of the form does not include an appropriate skin for a control type, that control is not drawn with a skin.

Note Any control that has had a specific skin set by calling the **Control** class **setSkin** method continues to use that specific skin. The form application skin is used only if **setSkin(null)** is subsequently called on that control.

When the **setApplicationSkin** method is first called, the collected skin data is stored as a blob on the **JadeSkinApplication** instance. Subsequent calls use this stored information and do not need to retrieve the skin information.

In addition, a presentation client caches the skin information. As a result, subsequent calls of the **setApplicationSkin** method only need to request the creation of the skin from the presentation client cache file without having to transmit the skin data.

When you change a skin definition using the **JadeSkinMaintence** form or by loading a form and data definition (**.ddb** or **.ddx**) file, the timestamp of all **JadeSkinApplication** instances is updated, which requires a rebuild of the skin information the first time each skin is set for an application or form. If you change JADE skin information by any other means, you must call the **updateSkinTimeStamp** method on the **JadeSkinApplication** instance, to reset the instance timestamp and cause the skin build data to be rebuilt.

setBackDrop

Signature
```
setBackDrop(pic:              Binary;
            type:             Integer;
            transparentColour: Integer);
```

The **setBackDrop** method of the **Form** class sets the backdrop binary picture image for the form. Use the **pic** parameter to specify the picture image for the form and the **type** parameter to specify zero (**0**) if the specified picture is to be stretched to the entire size or **1** if the picture is to be centered in the middle of the MDI frame.

Use the **transparentColour** parameter to specify the transparent color (that is, the bitmap is displayed so that the image of the parent shows through the bitmap anywhere that this color appears in the bitmap).

Note A **transparentColour** value of zero (**0**) sets the transparent color to black.

If you do not want a transparent color at all, set the value of **transparentColour** to **-1**.

The following example shows the use of the **setBackDrop** method. The first example uses class constants to make the second and third parameters more meaningful.

```
self.setBackDrop(self.picBackDrop.picture, BackDrop_Centred,
                 Black_Transparent);
begin
    logo.backColor := backColor;
    picture1.borderStyle := Window.BorderStyle_None;
    if not process.isUsingThinClient then
        setBackDrop(logo.createPicture(false, true, 24), 1, -1);
    endif;
end;
```

setCellSelected

Signature
```
setCellSelected(r:   Integer;
                c:   Integer;
                val: Boolean);
```

The **setCellSelected** method of the **Table** class sets the selected status of the cell specified in the **r** and **c** parameters of the current sheet of a table.

The **setCellSelected** method achieves the same as setting the **selected** property, except that the **row** and **column** properties do not need to be set.

Use the **val** parameter to specify the selected status of the cell; that is, set this parameter to **true** to select the cell or to **false** to deselect the cell.

This method has no impact on the current values of the **row** and **column** properties. See also the **getCellSelected**, **selectedCount**, and **selectedNext** methods.

setCellText

Signature
```
setCellText(r:   Integer;
            c:   Integer;
            str: String);
```

The **setCellText** method of the **Table** class sets the text of the cell specified in the **r** and **c** parameters of the current sheet of a table.

This method does the same as the **text** property, except that you do not need to set the values of the **row** and **column** properties, as this method has no impact on the values of those properties.

The method in the following example shows the use of the **setCellText** method.

```
cdtable_rowColumnChg(table: Table input) updating;
vars
    indx, count : Integer;
begin
```

```
        if table.row = 1 and table.column > 1 then
            table.sortColumn[1] := table.column;
            table.resort;
            count := table.rows - 1;
            indx  := 1;
            while indx <= count do
                table.setCellText(indx + 1, 1, indx.String);
                indx := indx + 1;
            endwhile;
            table.row    := 1;
            table.column := 1;
            table.clearAllSelected;
        endif;
    end;
```

The following code fragment shows the use of concatenation with the **Tab** character to store text in cells to the right of the specified cell.

```
    // Set up the column headings
    table.setCellText(1, 1, "Name" & Tab & "Address" & Tab & "Phone");
```

See also the **getCellText** method.

setCharacterFormat

Signature

```
setCharacterFormat(selection:  Boolean;
                   faceName:   String;
                   size:       Real;
                   color:      Integer;
                   bold:       Integer;
                   italic:     Integer;
                   strikethru: Integer;
                   underline:  Integer);
```

The **setCharacterFormat** method of the **JadeRichText** class sets common character formatting attributes of the receiver for the selected text or new text about to be typed. (For an example of the use of this method, see "JadeRichText Control Method Example", earlier in this document.)

The **setCharacterFormat** method parameters are listed in the following table.

Parameter	Description
selection	Specify **true** to set the character formatting attributes of the selected text, or if no text is selected, of the insertion point and specify **false** to set the default character format of the control
faceName	Sets the value of the **selFontName** property
size	Sets the value of the **selFontSize** property
color	Sets the value of the **selTextColor** property
bold	Sets the value of the **selFontBold** property
italic	Sets the value of the **selFontItalic** property
strikethru	Sets the value of the **selFontStrikethru** property
underline	Sets the value of the **selFontUnderline** property

You can specify the **CharacterFormat_Undefined** value for a font parameter if you do not want to apply a new value but retain the existing value of the attribute.

This method sets only the attributes of the currently selected text or the next inserted text at that point. It does not become the default for subsequent inserted or appended text.

When the value of the **selection** parameter is:

- **True**, the attributes of the text inserted at that point only (for example, an append) are inserted. When the selection point is moved at the end of the insertion (an append, and so on), the attributes revert to the default values for the control. If you therefore call the **setCharacterFormat** method followed by two calls to the **append** method, only the first of those append actions with those attributes is formatted.

- **False**, all text in the control with those attributes is formatted (that is, it sets only the characteristics of the text next entered by the user).

Calling the **load** method with a **replace** parameter value of **Load_ReplaceAll** empties the control by clearing the text contents of the control but leaving the **setCharacterFormat** method setting for the first character in place so that it becomes the default for the control. To clear the entire contents of the control, set the **text** property to null (**""**).

Although individual properties enable you to set character formatting attributes, you should consider the number of requests made to the control, particularly when running the JADE application in thin client mode. For example, calling the **setCharacterFomat** method to set all character format values involves one request from the application server to the presentation client but setting values individually requires seven calls to the presentation client to set the same information.

setClipBuffer

Signature
```
setClipBuffer(buffnum: Integer;
              text:    String);
```

The **setClipBuffer** method of the **JadeTextEdit** class sets the contents of the editor text (clip) buffer specified in the **buffnum** parameter with the text specified in the **text** parameter.

The value of the **buffnum** parameter can be in the range zero (**0**) through **JadeTextEdit.CLIPBUFFER_MAX**.

For details about getting the contents of the buffer, see the **getClipBuffer** method.

setCollectionObject

Signature
```
setCollectionObject(obj: Object);
```

The **setCollectionObject** method of the **Table** class sets the object in the collection attached to the table to the value specified in the **obj** parameter.

This method ensures that the object referenced is in the displayed list of collection entries for a table sheet. This is equivalent to setting **listbox.listObject := object;** when a collection is attached to a list box by using the **displayCollection** method.

If the object is already one of the row entries in the displayed table (the matching row object), the **setCollectionObject** method call just results in setting the **row** property of the sheet to that row.

If the object is not in the displayed table rows, the current table rows from the collection are discarded. The iterator position within the collection is then adjusted so that the rebuilt display rows include the requested object. The **row** property of the sheet is then set to the row of the requested object.

An exception is raised if a collection is not currently attached to the sheet that is being referenced or the object cannot be found within the collection.

setCurrentSchema

Signature `setCurrentSchema(schema: Schema);`

The **setCurrentSchema** method of the **JadeEditor** class sets the text editor keyword lists for classes, system and user-defined global constants, imported packages, and JADE interfaces for the schema specified in the schema parameter so that recognized keywords in method source can be displayed in the appropriate color.

Note Call the **initializeJadeEditor** method before you call the **setCurrentSchema** method, to ensure that the form displays entities in the correct color.

setDefaultPainterControlProperties

Signature `setDefaultPainterControlProperties() userHook, updating;`

The **setDefaultPainterControlProperties** method of the **Window** class is called by the JADE Painter whenever a new control is created.

You can re-implement this method on the **Control** class or a **Control** subclass, to set any user default properties you require.

The **Form** class **isPrinterForm** method returns whether the form was declared as a printer form on the New Form dialog in the JADE Painter; that is, the **Printer** option button was selected in the Form Style group box. The **isPrinterForm** method then allows the **setDefaultPainterControlProperties** method re-implementation to set properties only on a printer-style form, for example.

The following is an example of the re-implementation of the **setDefaultPainterControlProperties** method.

```
setDefaultPainterControlProperties();
begin
    if self.form.isPrinterForm() then
        self.fontName := "Arial";
    endif;
end;
```

Applies to Version: 2018.0.02 (Service Pack 1) and higher

setDragAndDropFiles

Signature `setDragAndDropFiles(method: Method);`

The **setDragAndDropFiles** method of the **Window** class enables files and folders to be dragged and dropped onto a form or control, by establishing the window as an allowed drop target. When files and folders are dropped onto that window or one of its children, JADE calls the method specified in the **method** parameter, passing the list of files and directories dropped.

Notes It is up to the specified method to process the list of files or directories.

When the user drags the files or folders over the window, Microsoft changes the cursor to a plus (**+**) symbol in a box, indicating that the window will accept dropped file and folders.

Windows does not allow you to drag and drop files or folders between processes with different levels of administration rights, so this method will therefore not be invoked if it is attempted.

The method passed to the **setDragAndDropFiles** method must be a method defined on the current form or a method defined on the class of the targeted control (or a superclass of the current form or targeted control).

If the method is defined on the form of the targeted window, the required signature is:

```
method-name(win: Window; aray: HugeStringArray);
```

The **win** parameter specifies the window on which the files were dropped and the **aray** parameter specifies an array of file or directory names that were dropped.

If the method is defined on the class of the targeted control targeted, the required signature is:

```
method-name(aray: HugeStringArray);
```

The **aray** parameter specifies an array of file or directory names that were dropped (**self** is the control instance on which the files were dropped).

Exception 1000 *(Invalid parameter type)* occurs if the method:

- Is not on the current **Form** class or on the targeted **Control** class
- Signature is not a format specified earlier in this topic
- Has a return type defined

If the **setDragAndDropFiles** method is called with a null value in the **method** parameter, the ability of the window to accept dropped files and directories is cancelled.

Note You cannot drag and drop files onto external .NET or ActiveX controls using this mechanism.

The following example shows the use of the **setDragAndDropFiles** method. (As the files are by definition of the client machine, in thin client mode, the files and directories must be accessed using **usePresentationClient := true**.)

```
listWilbur1.setDragAndDropFiles(MyForm::acceptFiles);

acceptFiles(dropWindow: Window; nameList: HugeStringArray);
vars
    file   : File;
    folder : FileFolder;
    str    : String;
begin
    create file transient;
    create folder transient;
    foreach str in nameList do
        file.fileName := str;
            // file?
        if file.isAvailable() then
            processDroppedFile(file);
        else
```

```
                folder.fileName := str;
                if folder.isAvailable() then
                    processDroppedFolder(folder);
                else
                    ... invalid entry?
                endif;
            endif;
        endforeach;
    epilog
        delete file;
        delete folder;
    end;
```

Applies to Version: 2016.0.01 and higher

setEventMapping

Signature
```
setEventMapping(eventName: String;
                mappedName: String);
```

The **setEventMapping** method of the **Window** class and **ActiveXControl** class enables the method that is executed for an event in this window to be dynamically set at run time. This method enables a form to be dynamically constructed and the event methods to be defined at run time.

Tip This method is equivalent to the **setEventMappingEx** method but it is less efficient, as it must find the methods by name. You should therefore use the **setEventMappingEx** method to improve performance.

By default, the JADE development environment allows the definition of event methods for a form (*event-name*), and for controls on the form (*control-name_event-name*). For example:

```
bInvestors_click(btn: Button input) updating;
```

The parameters of the **setEventMapping** method are listed in the following table.

Parameter	Description
eventName	Must be a defined event name for the form or control; for example, **click**.
mappedName	The name of the method that is to be called. This method must exist on the form that is the parent of the control for which you are calling the **setEventMapping** method or a method of the form that is calling the **setEventMapping** method.

The method checks that:

- The event method is valid for the window
- The method to be called exists
- The signature of the method matches the event method signature

setEventMappingEx

Signature
```
setEventMappingEx(eventMethod: Method;
                  mappedMethod: Method);
```

The **setEventMappingEx** method of the **Window** class enables the method that is executed for an event in this window to be dynamically set at run time and the mapping cached on each JADE node. This method enables a form to be dynamically constructed and the event methods to be defined at run time.

Repeat calls for a mapping that has been previously used is recognized and the signature check is not repeated unless the timestamp of the mapped method has changed since the previous signature check. The cost of reloading a form that assigns event mappings is therefore subsequently less expensive on that node. If an application server is involved, only the first assignment by any user performs the signature check. Subsequent repeat calls for any user on that application server avoid that overhead.

The event method being mapped and the mapped method are passed as parameters. The underlying logic, therefore, does not have to find the methods by name, making the execution more efficient; for example:

```
btnAction.setEventMethodEx(Button: click, Dialog::myButtonClick);
```

Tip This method is equivalent to the **setEventMapping** method but as it is more efficient, you should use the **setEventMappingEx** method to improve performance.

By default, the JADE development environment allows the definition of event methods for a form (*event-name*), and for controls on the form (*control-name_event-name*). For example:

```
bInvestors_click(btn: Button input) updating;
```

The parameters of the **setEventMappingEx** method are listed in the following table.

Parameter	Description
eventMethod	Specifies the event method, which must belong to the class of the receiver of the **setEventMappingEx** call, and must be a defined event name for the form or control; for example, **click**.
mappedMethod	The method that is to be called. This method must exist on the form that is the parent of the menu or control for which you are calling the **setEventMappingEx** method.

The method checks that:

- The event method is valid for the window
- The method to be called exists
- The signature of the method matches the event method signature

Applies to Version: 2016.0.02 (Service Pack 1) and higher

setFocus

Signature `setFocus();`

The **setFocus** method of the **Window** class and **Control** class sets the focus to a window or control.

If the window is a form (that is, a **form.setFocus** method call), focus is restored to the last control that had the focus on that form, unless the form has never been activated, in which case the focus is given to the first enabled visible control in the tab order.

If the control cannot have the focus (for example, it is a **Label**, **GroupBox**, **Frame**, **Picture**, or **Folder** control) or if the control is disabled or is not visible, the focus is set to the next enabled visible control in the tab order. After the **setFocus** method is executed, any user input is directed to the control that gained the focus.

You can set focus to a visible form or control only. Setting the focus to a control on a sheet of a folder control that is not the current top sheet causes that sheet to become the current top sheet (that is, visible).

The methods in the following examples show the use of the **setFocus** method.

```
mnuDataPeriod_click(menuItem: MenuItem input) updating;
vars
    form   : Form;
    lpForm : ListPeriodsForm;
begin
    form := app.getForm("ListPeriodsForm");
    if form = null then
        create lpForm transient;
        lpForm.show;
    else
        lpForm := form.ListPeriodsForm;
        lpForm.setFocus;              // set focus to this form
        lpForm.zOrder(1);
        lpForm.windowState := WindowState_Normal;
    endif;
end;

btnSearch_click(btn: Button input) updating;
begin
    topObject := positionCollectionByKey(txtStartName.text,
                                         theCollectionIterator);
    refreshTable;
    txtStartName.setFocus;
end;
```

setFontProperties

Signature
```
setFontProperties(fontName:    String;
                  fontSize:    Real;
                  fontBold:    Boolean);
```

The **setFontProperties** method of the **Control** class sets the values of the **Control** class **fontName**, **fontSize**, and **fontBold** properties.

Use this method to set the value of the **fontName**, **fontSize**, and **fontBold** properties of the control in one action; that is, instead of defining the example shown in the following code fragment.

```
listBox1.fontName := "Arial";
listBox1.fontSize := 9;
listBox1.fontBold := true;
```

Using the **setFontProperties** method can be more efficient than setting the properties individually, because the three properties are all set in the same action. When each property is set individually in JADE logic, a new font is created each time, and any impacts that the changed font has on the control size are applied; for example, auto-sizing, **parentAspect** positioning, or aligning controls. The **setFontProperties** method sets all three properties before applying any impacts.

For a **Table** control, the **setFontProperties** method uses the current setting of the **accessMode** property to determine whether the method will set the font for the current sheet, row, column, or cell.

setFormSkin

Signature `setFormSkin(skin: JadeSkinForm);`

The **setFormSkin** method of the **Form** class sets the current skin for the form regardless of the setting of the **Window** class **skinCategoryName** property. However, if the value of the **Window** class **ignoreSkin** property is set to **true**, the skin is still ignored.

The skin has no impact on the controls of the form other than potentially the **JadeSkinArea** class **backColor** property (for more information, see "JadeSkinForm Class" and "JadeSkinControl Class and Subclasses", in Chapter 1).

The controls continue to use the any application-defined skins. Setting a specific skin for a form takes precedence over any defined application skin for that form.

Note Changing a skin object after the **setFormSkin** method is called has no impact on the displayed skin. To apply any skin changes dynamically, you must call the **Form** class **setFormSkin** method again.

To clear the form skin (cancel the skin display), call the **setFormSkin** method again with a null value, as follows.

```
Form.setFormSkin(null);
```

The form then reverts to the use of an appropriate form skin set by the application.

For details about using skins to enhance your runtime applications, see Chapter 2 of the *JADE Runtime Application Guide*.

setFormParent

Signature `setFormParent(form: Form);`

The **setFormParent** method of the **Form** class sets the parent of a form to the form specified in the **form** parameter. This causes the child form to always be placed above the parent form in the order specified by the **zOrder** method.

Sibling controls that have the **alignContainer** property set do not occupy the same area on their parent. If so, use the **visible** property to control the window that is currently displayed.

The child form does not sit inside the parent as it does for an MDI child. The child is also closed when the parent is closed. As MDI forms cannot use this method, an exception is raised if you attempt to do so.

This method provides an alternative to running the form as a modal form (which causes all forms except the current modal form to be disabled) while still providing a dependency on the parent form. Passing a **null** value as the parent can clear the relationship.

The **getFormParent** method returns the parent of the form specified by using the **setFormParent** method or if the parent form was set directly by using a Windows API call.

setHyperlinkCell

Signature

```
setHyperlinkCell(hyperlinkRow:    Integer;
                 hyperlinkColumn: Integer);
```

The **setHyperlinkCell** method of the **Table** class sets up a HyperText link for a cell in a table.

Note The **setHyperlinkCell** method is ignored for JADE applications that are Web-enabled. For these applications, you should use the **hyperlinkColumn** array property of the **Table** class.

The **hyperlinkRow** and **hyperlinkColumn** parameters specify the cell in which the HyperText link is set. You can set HyperText links in more than one cell for a row.

The code fragment in the following example shows the use of the **setHyperlinkCell** method.

```
elseif checkbox.value then
    table1.setHyperlinkCell(table1.row, table1.column);
else
    table1.resetHyperlinkCell(table1.row, table1.column);
endif;
```

In a standard GUI application, the behavior of a cell with a HyperText link is as follows.

- The display changes only when the mouse moves over the cell; the cursor becomes a pointing hand and the text is underlined. The color of the text is blue, unless the **foreColor** property has been set specifically for the cell.
- If you want an action to occur when the HyperText link is clicked, you must provide the appropriate JADE logic for the **click** event for the table.
- A **rowColumnChg** event is not required for this process to function.

In a JADE forms Web-enabled application, the HyperText link is displayed in the Web browser in a standard way. There are two points to note:

- If you set the **inputType** property for the cell to any value apart from the default value (**InputType_None**) the hyperlink setting is ignored.
- You must implement the **rowColumnChg** event for the table.

setIndicatorAttributes

Signature

```
setIndicatorAttributes(indicatorNumber: Integer;
                       style:           Integer;
                       foreColor:       Integer);
```

The **setIndicatorAttributes** method of the **JadeTextEdit** class sets the text editor indicator attributes (for example, marking text to indicate syntax errors).

Use the **indicatorNumber** parameter to specify the number of the indicator whose style and color you want to set, in the range zero (**0**) through **7**.

The text editor has three indicators: zero (**0**), **1**, and **2**, whose default indicator styles and colors are listed in the following table.

Indicator Number	Style	Color	RGB Value
0	SC_INDIC_SQUIGGLE	Dark Green	0,127,0
1	SC_INDIC_TT	Light Blue	0,0,255
2	SC_INDIC_PLAIN	Light red	255,0,0

Use the **JadeTextEdit** class constants listed in the following table to specify the style of the indicator in the **style** parameter.

Constant	Value	Constant	Value
SC_INDIC_BOX	6	SC_INDIC_DIAGONAL	3
SC_INDIC_HIDDEN	5	SC_INDIC_PLAIN	0
SC_INDIC_SQUIGGLE	1	SC_INDIC_STRIKE	4
SC_INDIC_TT	2		

In the **foreColor** parameter, specify the color of the indicator.

setLinemarkAttributes

Signature
```
setLinemarkAttributes(markNumber: Integer;
                      style:      Integer;
                      foreColor:  Integer;
                      backColor:  Integer);
```

The **setLinemarkAttributes** method of the **JadeTextEdit** class sets the text editor linemark.

Use the **markNumber** parameter to specify the linemark number whose style and colors you want to set. You can assign style and color attributes to any of the 32 text editor linemarks, numbered zero (**0**) through **31**. (Values in the range **25** through **31** are used for folding.)

Use the **JadeTextEdit** class constants listed in the following table to specify the style of the indicator in the **style** parameter.

Constant	Value	Constant	Value
SC_MARK_ARROW	2	SC_MARK_ARROWDOWN	6
SC_MARK_ARROWS	24	SC_MARK_BACKGROUND	22
SC_MARK_BOXMINUS	14	SC_MARK_BOXMINUSCONNECTED	15
SC_MARK_BOXPLUS	12	SC_MARK_BOXPLUSCONNECTED	13
SC_MARK_CHARACTER	10000+*character*	SC_MARK_CIRCLE	0
SC_MARK_CIRCLEMINUS	20	SC_MARK_CIRCLEMINUSCONNECTED	21
SC_MARK_CIRCLEPLUS	18	SC_MARK_CIRCLEPLUSCONNECTED	19
SC_MARK_DOTDOTDOT	23	SC_MARK_EMPTY	5

Constant	Value	Constant	Value
SC_MARK_LCORNER	10	SC_MARK_LCORNERCURVE	16
SC_MARK_MINUS	7	SC_MARK_PLUS	8
SC_MARK_ROUNDRECT	1	SC_MARK_SHORTARROW	4
SC_MARK_SMALLRECT	3	SC_MARK_TCORNER	11
SC_MARK_ TCORNERCURVE	17	SC_MARK_VLINE	9

In the **foreColor** and **backColor** parameters, specify the foreground and background colors of the indicator. For details about the **SCI_MARKERDEFINE** and the associated **SC_MARKNUM_*** constants, see the Scintilla documentation at http://scintilla.sourceforge.net/ScintillaDoc.html.

The code fragment in the following example shows the use of the **setLinemarkAttributes** method.

```
// Set up bookmark linemark and bind keys
jteSource.setLinemarkAttributes(JadeTextEdit.MARKER_JAD_LINEMARK,
                               SC_MARK_ROUNDRECT, jteSource.rgb(255,180,180),
                               jteSource.rgb(128,255,255));
```

setLinemarkLines

Signature
```
setLinemarkLines(markNumber: Integer;
                 append:     Boolean;
                 lineList:   Boolean;
                 list:       IntegerArray);
```

The **setLinemarkLines** method of the **JadeTextEdit** class adds the linemark specified in the **markNumber** parameter to the lines (positions) specified in the **list** parameter.

If the value of the **append** property is **false**, all existing linemarks of the specified number are deleted before the entries in the list are added.

If the value of the **lineList** parameter is **true**, the list contains line numbers. If the value of the **lineList** parameter is set to **false**, the list contains character offsets that are limited to 128 entries. Values less than zero (**0**) or greater than the end of text are ignored.

The code fragment in the following example shows the use of the **setLinemarkLines** method to set linemark 1 on lines 1 and 25, and also remove the linemark from any other lines.

```
ia.add(1);
ia.add(25);
jteSource.setLinemarkLines(1, false, true, ia);
```

setNamedAttribute

Signature
```
setNamedAttribute(attName: String;
                  value:   Any);
```

The **setNamedAttribute** method of the **JadeTextEdit** class sets the named attribute specified in the **attName** parameter to the value specified in the **value** parameter.

The JADE text editor supports the following named attributes.

- **automatic.indenting**

  Boolean attribute that defaults to **true**, indicating that new lines are automatically indented to match the preceding (non-blank) line. Indentation of pasted lines is not changed.

- **caret.finder**

  Boolean attribute that defaults to **false**. When set **true**, pressing then releasing the Ctrl key causes the caret location to be highlighted.

- **caret.fore.color**

  Integer attribute that specifies the foreground color of the caret.

- **caret.line.back.color**

  Integer attribute that specifies the color that will override the background color of the line that currently contains the caret. The default value is **-1**, which disables the background color override functionality; that is, removes highlighting of the caret line. For example, the following code fragment sets the caret line background color to light green.

  ```
  jte.setNamedAttribute("caret.line.back.color", jte.rgb(200,255,200));
  ```

- **caret.width**

  Integer attribute that specifies the width of the caret in pixels, with the default value being **1**.

- **fold.compact**

  Boolean attribute that specifies if folding is compact or normal (the default). For more details, see the **folding** property.

- **fold.margin.color**

  Integer attribute that specifies the background color for the fold margin. It is an RGB value. The default value (**-1**) specifies that a system value be used.

- **linenumber.margin.width**

  Integer attribute that specifies the minimum line number margin width in characters, with the default value being **3** and the range **1** through **60**. The margin automatically expands if the last line number requires more space.

- **marker.margin.mask**

  Integer attribute that specifies a bit mask of the margin marker, with the default value being **SC_MASK_ FOLDERS**.

- **marker.margin.width**

  Integer attribute that specifies the width in pixels of the marker margin, with the default value being **20** pixels and the range **4** through **600**.

- **smart.indent.words**

  String attribute that defaults to null (**""**) and which is set to a list of words (for example, the **if**, **while**, and **foreach** JADE instructions). These words usually indicate the start of an indented block.

When **automatic.indenting** is set to **true** (that is, the **automatic.indenting** named attribute is set) and the list of smart words is not null, and a new line is inserted, and the previous non-blank line begins with one of the words in the *smart* list, the caret is indented one additional position. The JADE smart words are **constants**, **vars**, **begin**, **epilog**, **if**, **elseif**, **foreach**, and **while**.

The method in the following example shows the use of the **setNamedAttribute** method.

```
mnuEditSetAutoindent_click(menuItem: MenuItem input) updating;
begin
    menuItem.checked := not menuItem.checked;
    jteSource.setNamedAttribute("automatic.indenting", menuItem.checked);
end;
```

setOneColorText

Signature `setOneColorText(value: Boolean);`

The **setOneColorText** method of the **JadeEditor** class specifies whether text in the text editor is displayed as one single color or in the various colors specified for keywords, class, and constant names.

setParagraphFormat

Signature

```
setParagraphFormat(leftIndent:      Integer;
                   rightIndent:     Integer;
                   firstLineIndent: Integer;
                   alignment:       Integer);
```

The **setParagraphFormat** method of the **JadeRichText** class sets common paragraph formatting attributes of the receiver (that is, the paragraph that contains the insertion point). For an example of the use of this method, see "JadeRichText Control Method Example", earlier in this document.

The **setParagraphFormat** method parameters are listed in the following table.

Parameter	Description
leftIndent	Sets the value of the **leftIndent** property
rightIndent	Sets the value of the **rightIndent** property
firstLineIndent	Sets the value of the **firstLineIndent** property
alignment	Sets the value of the **alignment** property

You can specify the **ParagraphFormat_Undefined** value for a parameter if you do not want to apply a new value but retain the existing value.

Although individual properties enable you to set paragraph formatting attributes, you should consider the number of requests made to the control, particularly when running the JADE application in thin client mode. For example, calling the **setParagraphFormat** method to set all paragraph format values involves one request from the application server to the presentation client but setting values individually requires four calls to the presentation client to set the same information.

setPicture

Signature
```
setPicture(indx: Integer;
           pict: Binary);
```

The **setPicture** method sets the specified index entry of an array of pictures associated with the **Picture** control at run time.

The value specified in the **indx** parameter must be in the range **1** through **100**. Use the **pictureIndex** property to indicate the picture that is displayed. By default, the value of the **picture** property is used unless it is overridden by the **pictureDown** or **pictureDisabled** properties. See also the **pictureCount** property.

Setting the **picture** property to an animated GIF removes any pictures created by using the **setPicture** method. Similarly, changing the value of the **pictureCount** property or using the **setPicture** method closes any animated GIF operation (but leaves the value of the **picture** property unchanged).

setScrollRange

Signature
```
setScrollRange(scrollBar: Integer;
               min:       Integer;
               max:       Integer;
               smallChg:  Integer;
               largeChg:  Integer);
```

The **setScrollRange** method enables control of the scroll ranges for **Form**, **BaseControl**, and **Picture** controls.

The **setScrollRange** method changes the scroll range data. Scroll range data has no impact unless the window also has a corresponding scroll bar.

TextBox controls can obtain the current scroll range but cannot set the current scroll range. This range is determined automatically by the amount of text data in the control. **ListBox** and **Table** controls also offer the **getScrollRange** method only, as the ranges are set automatically by the control.

The following table lists the scroll data that is available by using the **setScrollRange** method parameters.

Parameter	Description
scrollBar	**1** for horizontal, **2** for vertical
min	Minimum scroll range
max	Maximum scroll range
smallChg	Size of scroll change when the user clicks a scroll arrow
largeChg	Size of scroll change when the user clicks elevator or uses the Page keys

All data units are in pixels.

For form and picture controls, the default scroll range data is listed in the following table.

Value	Default
min	0
max	1000
smallChg	20
largeChg	100

Control of the scroll bar position can be obtained by using the **scrollHorzPos** and **scrollVertPos** properties.

The code fragment in the following example shows the use of the **setScrollRange** method.

```
setScrollRange(ScrollBar_Vertical, 0, biggestOffset -
                (clientHeight/2).Integer, 20, 100);
```

setSkin

Signature

```
setSkin(skin: JadeSkin);            (Form)

setSkin(skin: JadeSkinControl);     (Control)
```

The **setSkin** method of the **Form** class sets the skin for the form, overriding any skin set for the application for that form.

Note Calling **form.setSkin(null);** causes that form to resume using the default skin of the application.

The **setSkin** method of the **Control** class sets the current skin for the control, regardless of the setting of the **Window** class **skinCategoryName** property and the **applyCondition** criteria. However, if the value of the **Window** class **ignoreSkin** property is set to **true**, the skin is still ignored. Setting a specific skin for a control takes precedence over any defined application skin.

Note that the skin object passed during this method call must correspond to the control type. For example, it must be of type **JadeSkinFrame** when the control is a **Frame** control class.

Note Changing a skin object after the **setSkin** method is called has no impact on the displayed skin. To apply any skin changes dynamically, you must call the **Control** class **setSkin** method again.

To clear the control skin (cancel the skin display), call this method again with a null value, as follows.

```
Control.setSkin(null);
```

The control then reverts to the use of an appropriate control skin set for the application.

For details about the **JadeSkinControl** class and its subclasses or the **JadeSkin** class, see the appropriate **JadeSkinEntity** subclass or "**JadeSkin** Class", in Chapter 1. For details about maintaining and using JADE skins, see Chapter 2 of the *JADE Runtime Application Guide*. See also the **Form** class **setApplicationSkin** method, earlier in this document.

setStyleAttributes

Signature

```
setStyleAttributes(styleNumber:   Integer;
                   fontName:      String;
                   fontSize:      Integer;
                   foreColor:     Integer;
                   backColor:     Integer;
                   fontBold:      Integer;
                   fontItalic:    Integer;
                   fontUnderline: Integer;
                   endOfLineFill: Integer);
```

The **setStyleAttributes** method of the **JadeTextEdit** class sets the attributes of a text editor style. Valid style numbers are in the range **0** through **127**. The actual style numbers that are used depends on the current language:

The style numbers that are common to all languages are the **JadeTextEdit** class constants listed in the following table.

Class Constant	Integer Value
STYLE_BRACEBAD	35
STYLE_BRACELIGHT	34
STYLE_CONTROLCHAR	36
STYLE_DEFAULT	32
STYLE_INDENTGUIDE	37
STYLE_LINENUMBER	33
STYLE_MAX	127

For the **STYLE_LINENUMBER** (**33**) style, the background color is the background color for the line number and linemark margins. (Use the **fold.margin.color** named attribute to set the background for the fold margin.)

For the **STYLE_INDENTGUIDE** (**37**) style, the indent guide is a dotted line alternating between the foreground and background colors of this style.

JADE language style numbers are the **JadeTextEdit** class constants listed in the following table.

Constant	Value	Constant	Value
SCE_JAD_BINARYLITERAL	22	SCE_JAD_COMMENT	6
SCE_JAD_COMMENTLINE	7	SCE_JAD_DEFAULT	4
SCE_JAD_DOCTEXT	21	SCE_JAD_DOLLARIDENT	23
SCE_JAD_GLOBALCONST	17	SCE_JAD_IDENTIFIER	11
SCE_JAD_INTERFACE	20	SCE_JAD_KEYWORD	12
SCE_JAD_METHODWORD	13	SCE_JAD_NUMBER	10
SCE_JAD_PACKAGE	18	SCE_JAD_PACKAGECLASS	19
SCE_JAD_PUNCTUATION	5	SCE_JAD_SINGLECOLOR	0
SCE_JAD_STRING1	8	SCE_JAD_STRING2	9
SCE_JAD_SYSTEMCLASS	15	SCE_JAD_SYSTEMVAR	14
SCE_JAD_USERCLASS	16		

To leave an attribute unchanged, specify null (**""**) in the **fontName** parameter and specify the **ATTRIB_NOCHANGE** (-2) **JadeTextEdit** class constant for the other parameters. To set an attribute to the same value as the attribute of the default style, specify **"*"** in the **fontName** parameter and specify the **ATTRIB_DEFAULT** (-1) class constant for the other parameters.

Valid values for the **fontSize** parameter are point sizes in the range **0** through **72**. The values in the range **990** through **1010** specify a size relative to the font size of the default style, in the range **-10** through **+10**. A relative size is converted to an absolute value by adding the font size to the default style font size. The font size is always rounded up to a minimum of **2**.

Set the **foreColor** and **backColor** parameters to an RGB color value, **ATTRIB_NOCHANGE**, or **ATTRIB_DEFAULT**.

Use the **JadeTextEdit** class constants listed in the following table to specify the values of the **fontBold**, **fontItalic**, **fontUnderline**, and **endOfLineFill** parameters, respectively.

Class Constant	Integer Value	Description
ATTRIB_DEFAULT	-1	Use the default style value
ATTRIB_FALSE	0	False
ATTRIB_NOCHANGE	-2	Do not change
ATTRIB_TRUE	1	True

The method in the following example shows the use of the **setStyleAttributes** method.

```
mnuEditSetMargcolor_click(menuItem: MenuItem input) updating;
vars
    val : Integer;
begin
    if self.askForColorSetting(self.caption, "Margin[0,1,2] Color", val)
        then
        jteSource.setStyleAttributes(JadeTextEdit.STYLE_LINENUMBER,"",
                 JadeTextEdit.ATTRIB_NOCHANGE, JadeTextEdit.ATTRIB_NOCHANGE,
                 val, JadeTextEdit.ATTRIB_NOCHANGE,
                 JadeTextEdit.ATTRIB_NOCHANGE,
                 JadeTextEdit.ATTRIB_NOCHANGE,
                 JadeTextEdit.ATTRIB_NOCHANGE);
        jteSource.setNamedAttribute("fold.margin.color", val);
    endif;
end;
```

setSystemTrayEntry

Signature
```
setSystemTrayEntry(str:  String;
                   icon: Binary);
```

The **setSystemTrayEntry** method of the **Form** class places an entry in the system tray of the Windows taskbar for this form. If you call this method at run time and the value of the **icon** parameter is **null**, the icon of the form is used.

Use the **str** parameter to specify a text value of up to 63 characters that is displayed in bubble help when the mouse is moved over the system tray. If you define a string longer than 63 characters, only the first 63 characters are displayed in the bubble help.

Note If the form is made invisible, an entry in the system tray is considered as having a visible form so that if the form is the only form in the application that is running, the *no visible form* warning message therefore does not occur.

The system tray icon entry can be used to provide the user with a means of signaling the form by using the icon, and it also provides the application with a way of signaling the user of an action (for example, that mail has arrived). The icon remains visible regardless of the state of the form (for example, if it is visible or minimized).

Clicking the icon in the system tray generates the **Form** class **trayIconClicked** method. Closing the form automatically removes the system tray entry.

A visible form still has an entry on the task bar if the form has a caption.

See also the **hasSystemTrayEntry** and **removeSystemTrayEntry** methods.

setTabStops

Signature `setTabStops(stops: IntegerArray);`

The **setTabStops** method of the **JadeRichText** class populates an array with the positions of the tab stops in the receiver control.

setTaskBarProgress

Signature
```
setTaskBarProgress(value:  Integer;
                   maxVal: Integer);
```

The **setTaskBarProgress** method of the **Form** class sets the extent of the progress to be displayed on the application icon in the taskbar.

If the method is called:

- On an MDI child form, the value is applied to the MDI frame form, because the MDI child form does not have an icon on the task bar.
- Before the form is shown (for example, in the **load** method), the value is applied when the form is shown. because there is no task bar icon before the form is made visible.

Notes This functionality is available only if the application displays an icon on the Windows taskbar. It does not apply to icons in the system tray.

If two forms call the **setTaskBarProgress** method while the taskbar progress is displayed, the progress displays the lowest value.

The method call can silently fail for some earlier versions of Windows (for example, Windows 8.0), even though the Microsoft documentation states that the functionality is available from Windows 7 and later.

Calling this method causes the progress indicator to be displayed in its set state. If the current state was **TaskBar_State_NoProgress** or **TaskBar_State_Indeterminate**, the state becomes **TaskBar_State_Normal**.

The **setTaskBarProgress** method **value** parameter specifies the current progress value and the **maxVal** parameter specifies the maximum value that will be reached. Both values are relative to zero (**0**).

To hide the progress state when the action is completed, call the **setTaskBarState** method with the **state** parameter set to **TaskBar_State_NoProgress**.

Applies to Version: 2018.0.01 and higher

setTaskBarState

Signature `setTaskBarState(state: Integer);`

The **setTaskBarState** method of the **Form** class sets the state of the taskbar icon for the application.

If the method is called:

- On an MDI child form, the value is applied to the MDI frame form, because the MDI child form does not have an icon on the task bar.
- Before the form is shown (for example, in the **load** method), the value is applied when the form is shown. because there is no task bar icon before the form is made visible.

Notes This functionality is available only if the application displays an icon on the Windows taskbar. It does not apply to icons in the system tray.

The method call can silently fail for some earlier versions of Windows (for example, Windows 8.0), even though the Microsoft documentation states that the functionality is available from Windows 7 and later.

The **state** parameter of this method can contain one of the **Form** class constants listed in the following table.

Form Class Constant	Value	Description
TaskBar_State_NoProgress	0	Hides the progress state on the icon
TaskBar_State_Indeterminate	1	Causes a continuous icon progress state, drawing the state in green from 0 through 100 percent, repeated, indicating the action is in progress but the progress completion time is unknown
TaskBar_State_Normal	2	Displays the current progress state in green
TaskBar_State_Error	4	Sets the progress state to be drawn in red to indicate an error state
TaskBar_State_Paused	8	Displays the current progress state in yellow, to indicate that the action has been paused

Any other value passed to the **setTaskBarState** method is treated as **TaskBar_State_Normal** (2).

Applies to Version: 2018.0.01 and higher

setTextFromCurrencyDecimal

Signature `setTextFromCurrencyDecimal(dec: Decimal);`

The **setTextFromCurrencyDecimal** method sets the **text** property value of a **TextBox** control to a **Decimal** value converted to a string in the currency format of the locale under which the control is running.

When the **dataType** property of a text box is set to **DataType_Currency**, the **decimals** property is set to **-1**, indicating that the number of decimal places for currency defined by the current locale is used.

To manually control how many decimal places are allowed, set the **decimals** property to a value other than **-1**.

The **setTextFromCurrencyDecimal** method of a **TextBox** control sets the **text** property to a **Decimal** value converted to a string in the format of the locale of the user application. If the text box is numeric (that is, the **dataType** property value of **DataType_Numeric** (1), **DataType_SignedNumeric** (2), or **DataType_Currency** (3)) and the value of **decimals** property is positive, the number of decimals places displayed is equal to the **decimals** property value. If the text box is not numeric or the value of the **decimals** property is less than zero (**0**), the number of decimal places specified in the **dec** parameter is retained. (Note that an exception is raised if the resulting string exceeds the **maxLength** property value of the text box.)

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Notes A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data. Setting decimal text by using the **setTextFromDecimal** method converts the numeric into the appropriate string for that locale. Retrieving the **Decimal** value by using the **getTextAsCurrencyDecimal** method does the reverse. You can therefore use the **getTextAsCurrencyDecimal** and **setTextFromCurrencyDecimal** methods to access decimal text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123**-).

setTextFromCurrencyReal

Signature `setTextFromCurrencyReal(real: Real);`

The **setTextFromCurrencyReal** method sets the text from the **text** property of a **TextBox** control to a **Real** value converted to a string in the format of the locale under which the control is running (for details, see the **languageId** property).

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The **setTextFromCurrencyReal** method of a **TextBox** control sets the **text** property value to a **Real** value converted to a string in the format of the locale of the user application. The number of decimal places displayed is determined by the value of the **decimals** property and the **Real** number is rounded to that precision. (Note that an exception is raised if the resulting string exceeds the **maxLength** property value of the text box.)

Notes A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** property and when the user enters data. Setting the text by using the **setTextFromCurrencyReal** method converts the number into the appropriate string for that locale. Retrieving the **Decimal** value by using the **getTextAsCurrencyReal** method does the reverse. You can therefore use the **getTextAsCurrencyReal** and **setTextFromCurrencyReal** methods to access numeric text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123**-).

setTextFromDate

Signature `setTextFromDate(date: Date);`

The **setTextFromDate** method of the **JadeEditMask** control class sets the text from the **textUser** property to a **Date** value converted to a string in the format of the locale under which the control is running (for details, see the **languageId** property). If the value of the **mask** property is null (**""**) or if it does not specify that the data is a full date field (for example, **ddMMMyyyy** or **dd/MM/yyyy**), an exception is raised.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default short date format of **dd/MM/yyyy**, and it has been overridden with a short date format of **yyyy-MM-dd**, this is returned in the default **dd/MM/yyyy** format.

A **JadeEditMask** control expects dates to be formatted according to the locale that the control is using. This applies when accessing the **text** or **textUser** property and when the user enters data. Setting date text by using the **setTextFromDate** method converts the date into the appropriate string for that locale.

Retrieving the **Date** value by using the **getTextAsDate** method does the reverse. You can therefore use the **getTextAsDate** and **setTextFromDate** methods to access date text so that JADE handles the locale format for you.

setTextFromDecimal

Signature `setTextFromDecimal(dec: Decimal);`

The **setTextFromDecimal** method sets the text from the **textUser** property of a **JadeEditMask** control to a **Decimal** value converted to a string in the format of the locale under which the control is running (for details, see the **languageId** property).

The string contains the number of decimal places specified by the **mask** property. If the **mask** property is null (**""**), the number of decimal places specified in the **dec** parameter is retained. (Note that an exception is raised if the resulting string does not conform to the data format specified by the **mask** property value.)

The **setTextFromDecimal** method of a **TextBox** control sets the **text** property to a **Decimal** value converted to a string in the format of the locale of the user application.

If the text box is numeric (that is, the **dataType** property value of **DataType_Numeric** (1), **DataType_SignedNumeric** (2), or **DataType_Currency** (3)) and the value of **decimals** property is positive, the number of decimals places displayed is equal to the decimals property value.

If the text box is not numeric or the value of the **decimals** property is less than zero (**0**), the number of decimal places specified in the **dec** parameter is retained. (Note that an exception is raised if the resulting string exceeds the **maxLength** property value of the text box.)

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Notes A numeric **JadeEditMask** control expects the negative sign, decimal place, and separator characters to be in the form defined for the locale under which the control is running. A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data. Setting decimal text by using the **setTextFromDecimal** method converts the numeric into the appropriate string for that locale. Retrieving the **Decimal** value by using the **getTextAsDecimal** method does the reverse. You can therefore use the **getTextAsDecimal** and **setTextFromDecimal** methods to access decimal text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **JadeEditMask** or **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123-**).

setTextFromInteger

Signature `setTextFromInteger(int: Integer);`

The **setTextFromInteger** method sets the text from the **textUser** property of a **JadeEditMask** control to an **Integer** value converted to a string in the format of the locale under which the control is running (for details, see the **languageId** property). Note that an exception is raised if the resulting string does not conform to the data format specified by the **mask** property value.

The **setTextFromInteger** method of a **TextBox** control sets the **text** property value to an **Integer** value converted to a string in the format of the locale of the user application. Note that an exception is raised if the resulting string exceeds the **maxLength** property value of the text box.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Notes A numeric **JadeEditMask** control expects the negative sign, decimal place, and separator characters to be in the form defined for the locale under which the control is running. A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data. Setting the text by using the **setTextFromInteger** method converts the numeric into the appropriate string for that locale. Retrieving the **Integer** value by using the **getTextAsInteger** method does the reverse. You can therefore use the **getTextAsInteger** and **setTextFromInteger** methods to access numeric text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **JadeEditMask** or **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123-**).

setTextFromInteger64

Signature `setTextFromInteger64(int64: Integer64);`

The **setTextFromInteger64** method sets the text from the **textUser** property of a **JadeEditMask** control to an **Integer64** value converted to a **String** in the format of the locale under which the control is running (for details, see the **languageId** property). Note that an exception is raised if the resulting string does not conform to the data format specified by the **mask** property value.

The **setTextFromInteger64** method of a **TextBox** control sets the **text** property value to an **Integer64** value converted to a **String** in the format of the locale of the user application. Note that an exception is raised if the resulting string exceeds the **maxLength** property value of the text box.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Notes A numeric **JadeEditMask** control expects the negative sign, decimal place, and separator characters to be in the form defined for the locale under which the control is running. A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data. Setting the text by using the **setTextFromInteger64** method converts the numeric into the appropriate **String** for that locale. Retrieving the **Integer64** value by using the **getTextAsInteger64** method does the reverse. You can therefore use the **getTextAsInteger64** and **setTextFromInteger64** methods to access numeric text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **JadeEditMask** or **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123-**).

setTextFromLongDate

Signature `setTextFromLongDate(date: Date);`

The **setTextFromLongDate** method of a **TextBox** control sets the **text** property value to a **Date** value converted to a **String** in the format of the locale of the user application. Note that an exception is raised if the resulting string exceeds the **maxLength** property value of the text box.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Note Retrieving the long date value by using the **getTextAsLongDate** method returns the text in long date format for the current locale. Setting date text by using the **setTextFromLongDate** method does the reverse.

You can therefore use the **getTextAsLongDate** and **setTextFromLongDate** methods to access long date text so that JADE handles the locale format for you.

setTextFromReal

Signature `setTextFromReal(real: Real);`

The **setTextFromReal** method sets the text from the **textUser** property of a **JadeEditMask** control to a **Real** value converted to a string in the format of the locale under which the control is running (for details, see the **languageId** property). The string contains the number of decimal places specified by the **mask** property. If the value of the **mask** property is null (**""**), the number of decimal places depends on the value of the **real** parameter. (Note that an exception is raised if the resulting string does not conform to the data format specified by the **mask** property value.)

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The **setTextFromReal** method of a **TextBox** control sets the **text** property value to a **Real** value converted to a string in the format of the locale of the user application. The number of decimal places displayed is determined by the value of the **decimals** property and the **Real** number is rounded to that precision. (Note that an exception is raised if the resulting string exceeds the **maxLength** property value of the text box.)

Notes A numeric **JadeEditMask** control expects the negative sign, decimal place, and separator characters to be in the form defined for the locale under which the control is running. A numeric **TextBox** control expects the negative sign and decimal place characters to be in the form defined for the locale under which the user is running. This applies when accessing the **text** or **textUser** properties and when the user enters data. Setting the text by using the **setTextFromReal** method converts the numeric into the appropriate string for that locale. Retrieving the **Decimal** value by using the **getTextAsReal** method does the reverse. You can therefore use the **getTextAsReal** and **setTextFromReal** methods to access numeric text so that JADE handles the locale format for you.

The position of a negative sign indicated by the locale is honored by a leading or trailing numeric in the **JadeEditMask** or **TextBox** control. However, locales that specify a negative value expressed by parentheses (for example, **(123)**) are treated as a leading dash symbol (**-**). In addition, locales that have a leading or trailing space associated with a negative sign are treated as the equivalent format without the space (for example, **- 123** will be **-123** and **123 -** will be **123**-).

setTextFromShortDate

Signature `setTextFromShortDate(date: Date);`

The **setTextFromShortDate** method of a **TextBox** control sets the **text** property value to a **Date** value converted to a **String** in the short date format of the current locale. Note that an exception is raised if the resulting string exceeds the **maxLength** property value of the text box.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Note Retrieving the long date value by using the **getTextAsShortDate** method returns the text in long date format for the current locale. Setting date text by using the **setTextFromShortDate** method does the reverse.

You can therefore use the **getTextAsShortDate** and **setTextFromShortDate** methods to access long date text so that JADE handles the locale format for you.

setTextFromTime

Signature `setTextFromTime(time: Time);`

The **setTextFromTime** method of the **JadeEditMask** control class sets the text from the **textUser** property to a **Time** value converted to a string in the format of the locale under which the control is running (for details, see the **languageId** property). If the value of the **mask** property is null (**""**) or if it does not specify that the data is a time field with at least an hour and minutes mask (for example, **hh:mm:ss**), an exception is raised. If the mask does not have a seconds mask, the seconds in the time are ignored.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeEditMask** class validates the setting and entry of text based on the **mask** property, using the current locale of the client with regional overrides on both the presentation client and the application server.

The **setTextFromTime** method of the **TextBox** control class sets the text of the text box to the specified **Time** value converted to a string in the format of the current locale.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default short time format of **HH:mm:ss** (24-hour clock), and it has been overridden with a short time format of **hh:mm:ss** (12-hour clock), this is returned in the default **HH:mm:ss** format.

A **JadeEditMask** control expects times to be formatted according to the locale that the control is using. This applies when accessing the **text** or **textUser** property and when the user enters data. Setting time text by using the **setTextFromTime** method converts the time into the appropriate string for that locale.

Retrieving the **Time** value by using the **getTextAsTime** method does the reverse. You can therefore use the **getTextAsTime** and **setTextFromTime** methods to access time text so that JADE handles the locale format for you.

setTextProtection

Signature
```
setTextProtection(start:     Integer;
                  length:    Integer;
                  protected: Boolean);
```

The **setTextProtection** method of the **JadeRichText** class enables you to mark a range of the content of a **JadeRichText** control from the position specified in the **start** parameter through to the length specified in the **length** parameter as protected or unprotected, by setting the **protected** parameter to **true** or **false**, respectively.

The method in the following example shows the use of the **setTextProtection** method.

```
vars
    start, length : Integer;
begin
    start := jadeRichText.selStart;
    length := jadeRichText.selLength;
    jadeRichText.setTextProtection(start, length, true);
end;
```

setTextRangeToStyle

Signature

```
setTextRangeToStyle(firstCharacter: Integer;
                    length:         Integer;
                    styleMask:      Integer;
                    styleValue:     Integer);
```

The **setTextRangeToStyle** method of the **JadeTextEdit** class updates the current text style setting for each character in the specified range with the values specified in the **styleMask** and **styleValue** parameters. You can use this method, for example, to set the text range so that it is displayed marked with a specified indicator.

In the **firstCharacter** parameter, specify the zero-based character offset (that is, the first character of text in the control is **0**, the second is **1**, and so on) of the first character whose text style you want to change.

In the **length** parameter, specify the number of characters whose text style you want to update.

In the **styleMask** parameter, you can specify one or a combination of the **JadeTextEdit** class constants listed in the following table to define the text style indicator; that is, which of the eight style bits you want to change.

Constant	Value	Constant	Value
SC_INDIC0_MASK	#20	SC_INDIC1_MASK	#40
SC_INDIC2_MASK	#80	SC_INDICS_MASK	#E0
SC_STYLES_MASK	#1F		

Note that indicator flags of each character are cleared when the text in a text editor control is restyled (for example, by calling the **restyleText** method).

In the **styleValue** parameter, specify the text style that you require; that is, the new value for the style bits indicated in the **styleMask** parameter.

As the text style is usually language-dependent, the style for the JADE language could be one of the **JadeTextEdit** class constants listed in the following table or it could be an indicator flag.

Constant	Value	Constant	Value
SCE_JAD_BINARYLITERAL	22	SCE_JAD_COMMENT	6
SCE_JAD_COMMENTLINE	7	SCE_JAD_DEFAULT	4
SCE_JAD_DOCTEXT	21	SCE_JAD_DOLLARIDENT	23
SCE_JAD_GLOBALCONST	17	SCE_JAD_IDENTIFIER	11
SCE_JAD_INTERFACE	20	SCE_JAD_KEYWORD	12
SCE_JAD_METHODWORD	13	SCE_JAD_NUMBER	10

Constant	Value	Constant	Value
SCE_JAD_PACKAGE	18	SCE_JAD_PACKAGECLASS	19
SCE_JAD_PUNCTUATION	5	SCE_JAD_SINGLECOLOR	0
SCE_JAD_STRING1	8	SCE_JAD_STRING2	9
SCE_JAD_SYSTEMCLASS	15	SCE_JAD_SYSTEMVAR	14
SCE_JAD_USERCLASS	16		

The code fragment in the following example shows the use of the **setTextRangeToStyle** method.

```
//Define indicator zero to be a red wavy underline symbol
setIndicatorAttributes(0, SC_INDIC_SQUIGGLE, Red);
//Change styling for characters 50 through 54 to set indicator 0
//(and to clear 1 and 2)
setTextRangeToStyle(50, 5, SC_INDICS_MASK, SC_INDIC0_MASK);
```

setValue

Signature
```
setValue(controlName: String;
         memberName:  String;
         paramList:   ParamListType);
```

The **setValue** method of the **JadeXamlControl** class sets the value of a WPF property for an entity of a XAML control

The parameters are combined to form a sequence of accesses to the WPF entities involved. The JADE method parameters are a mixture of property names, method names, and WPF method parameters, with the last parameter containing the primitive value to be set. as described in the following table.

Parameter	Description
controlName	Name of the WPF FrameworkElement involved. If the name is null or equal to the control name, the search for the **memberName** starts with the parent control; otherwise the search starts with the first child element with the specified name. The search succeeds when the entity or one of its children is found to have the specified **memberName** value. An exception is raised if the **controlName** or **memberName** is not found.
memberName	Name of the first method or property being accessed.
paramList	Remaining property, methods, and parameters used in sequence. The final parameter is the value being set.

The code fragments in the following examples show how these parameters are used.

```
jadeXamlCtl.setValue("list", "SelectedIndex", 3);
     // sets the integer value of the currently selected item of the
     // list box entity named "list" to 3 by executing the WPF sequence
     // list.SelectedIndex = 3.

jadeXamlCtl.setValue("list", "SelectedItem", "Content", "new text");
     // sets the content of the currently selected item of the list box
     // to "new text" by executing the WPF sequence
     // list.SelectedItem.Content = "new text"
```

```
jadeXamlCtl.setValue("list", "Items", "GetItemAt", 3, "Content", "Customer")
     // sets the content of the list box entry with an index of 3 to
     // "Customer", by executing the WPF sequence
     // list.Items.GetItemAt(3).Contents= "Customer"
```

Note the following restrictions.

- Only JADE primitives types are supported as parameters to WPF method calls.
- Access to static WPF properties and methods is not supported.
- The last parameter is the value to be set.
- The penultimate parameter is the name of a WPF property that can be set.
- Parameter types must match the same basic type; that is, an **Integer** parameter must be passed as an **Integer**, a floating point or real number as a **Real**, a byte as a **Byte**, and so on.
- For a thin client, calls to this method are buffered (the application server does not wait for a reply). If a call fails, an exception is raised. However the current line of logic listed as causing the exception does not indicate where the method was called.

setWordCharactersets

Signature

```
setWordCharactersets(wordChars:        String;
                     punctuationChars: String;
                     whitespaceChars:  String);
```

The **setWordCharactersets** method of the **JadeTextEdit** class sets the characters that are treated as part of a word (specified in the **wordChars** parameter) and those that are treated as white space (specified in the **whitespaceChars** parameter) and those that are treated as punctuation (specified in the **punctuationChars** parameter) in the text editor.

The keyboard command for the next word (that is, Ctrl+right arrow) and for the previous word (that is, Ctrl+left arrow) use these character sets. (See also the **moveCaret** method.)

As white space characters are skipped when positioning the caret before the previous or next word, you can use the **setWordCharactersets** method to specify additional characters that are ignored when searching for the next or previous word. Characters can be in the ranges **a** through **z**, **A** through **Z**, and **0** through **9**, as well as the period (**.**), comma (**,**), exclamation point (**!**), and question mark (**?**) symbols (for example, **".,!? "**).

Each **setWordCharactersets** method call restores the character sets to the default values. The default character sets are:

- Word character set

  Uppercase and lowercase letters, digits, underscore characters, and any character with a decimal value greater than **127**.

- Whitespace

  Space, tab, and all control characters (decimal value less than **32**).

- Punctuation

  All other character sets.

Characters specified in each parameter are added to the appropriate character set. A specified character can be in one character set only.

Limit the specified characters to the decimal equivalent range of zero (**0**) through 127. Extended and Unicode characters default to the word character set.

setXamlEventMethod

Signature
```
setXamlEventMethod(control:    String;
                   xamlEvent:  String;
                   jadeMethod: Method);
```

The **setXamlEventMethod** method of the **JadeXamlControl** class registers the control for other Window Presentation Foundation (WPF) control events apart from the standard JADE events.

The parameters of the **setXamlEventMethod** are described in the following table.

Parameter	Name of the …
control	WPF FrameworkElement. If the name is null or the same as the name of the control, the event is for the parent XAML element; otherwise the event is for the child element with the supplied name.
xamlEvent	WPF event to be registered.
jadeMethod	JADE method on the form that is to be called when the event occurs. The method must have the ***method-name*****(xamlControl: JadeXamlControl, itemName: String);** signature, where the **xamlControl** value is the control receiving the event and the **itemName** value is the WPF element invoking the event. If this parameter is null, an existing event registered for the **controlName**, **xamlEvent** combination is removed.

An exception is raised if is the **xamlEvent** parameter is not a defined WPF event for the control, or if the **jadeMethod** parameter is not a defined method on the form or a superclass.

If the control is a subclass of **JadeXamlControl** and has a method with the same name as the **jadeMethod** parameter, the event calls the subclass method rather than the form method.

The following code fragment shows the use of the **setXamlEventMethod** method to register the **myTextChanged** method to receive the **TextChanged** event.

```
jadeXamlCtl.setXamlEventMethod("text1","TextChanged",MyForm::myTextChanged);
```

shareDocumentFrom

Signature
```
shareDocumentFrom(original: JadeTextEdit);
```

The **shareDocumentFrom** method of the **JadeTextEdit** class specifies the **JadeTextEdit** object whose text buffer is shared by this object.

Use the **original** parameter to specify the text editor that this object is to share at run time.

This method links the receiver control to the text edit control specified in the **original** parameter so that both controls display the same text. Use this method to implement split windows, as shown in the following example.

```
mnuEditSplit_click(menuItem: MenuItem input) updating;
begin
    menuItem.checked := not menuItem.checked;
    if menuItem.checked then
        jteSecond.shareDocumentFrom(jteSource);
        jteSecond.visible := true;
    else
        jteSecond.shareDocumentFrom(null);
        jteSecond.visible := false;
    endif;
end;
```

sheets

Signature `sheets(): Integer;`

The **sheets** method returns the number of sheets for a **Folder** control. See also the **sheets** property of the **Table** control.

show

Signature `show();`

The **show** method of the **Form** class makes the form visible in its current state; that is, minimized, maximized, or normal.

The **show** method is invoked and starts execution before the **load** event method is invoked. Within the **show** method, the presence of an **inheritMethod** call causes the **load** method to be invoked. Consequently, any user logic positioned prior to the **inheritMethod** call is executed before the **load** event method executes.

Use the **windowState** property to control the state of the form. The first **show** statement for a created form executes its **load** event if it has not already been executed. Setting the value of the **visible** property to false within the **load** event is ignored when the **show** method is performed. The form must have been created before it can be shown. The **create** method constructs the windows and menus for the forms and its controls, as defined in the JADE development environment.

For a form defined as a Web page, the **show** method executes the **load** event, generates HTML, and returns this HTML back to the Web server. Unlike the **showModal** method, the logic returns to the statement following the **show** method after the form is made visible. Other forms can also receive user input (from the keyboard or mouse) while that form is active.

The methods in the following examples show the use of the **show** method.

```
menuCustomerAdd_click(menuItem: MenuItem input) updating;
vars
    form : CustomerDetailsForm;
begin
    create form transient;
    form.show;
end;

listBoxCustomers_dblClick(listbox: ListBox input) updating;
vars
    cust  : Customers;
    custs : CustomersByContactNameDict;
    form  : CustMaintForExternalDB;
begin
    create custs;
    cust := custs.getAtKey
        (listBoxCustomers.text[1:listBoxCustomers.text.pos(",", 1)-1]);
    create form transient;
    form.textBoxName.enabled := false;
```

```
        form.myExtCustomers       := cust;
        form.textBoxName.text     := cust.contactName;
        form.textBoxCity.text     := cust.city;
        form.textBoxAddress.text  := cust.address;
        form.show;
    end;
```

Note The **show** method cannot be used for a form created with the **createPrintForm** method of the **GUIClass** class.

showDropDown

Signature `showDropDown();`

The **showDropDown** method of the **ComboBox** class opens (shows) the drop-down list of the combo box control. (Use the **isDroppedDown** method to determine if the drop-down list is currently open.)

Note The **showDropDown** method is ignored if the combo box is not on the currently active form.

showHelp

Signature `showHelp(): Boolean;`

Call the **showHelp** method on a **Window** class object to invoke Windows help on that object if its **helpKeyword** or **helpContextId** property is set.

The **helpFile** property of the application determines the help file that is used. This method returns **true** if Windows help was initiated or **false** if it was not (that is, the **Window** class object had no help information).

When help is invoked directly via the **Window** class **showHelp** method or via the user pressing the help key (`F1`), a URL is created and the default browser is invoked to display the URL if the **helpFile** property or the **Window** class **helpKeyword** property is set to a base URL.

If the **helpKeyword** property was set, a partial key search is performed using that keyword. If the **helpKeyword** property is not set but the **helpContextId** property is set, that context id is used to access the help file.

The method in the following example shows the use of the **showHelp** method.

```
    buttonHelp_click(btn: Button input) updating;
    begin
        if listBoxMethods.text.pos ("::", 1) = null then
            helpKeyword := listBoxMethods.text;
        else
            helpKeyword := listBoxMethods.text[(listBoxMethods.text.pos
                                               ("::",1)+ 2) : end];
        endif;
        showHelp;
    end;
```

showInsertForm

Signature
```
showInsertForm(allow:   Integer,
               initDir: String;
               filter:  String): Integer;
```

The **showInsertForm** method of the **OleControl** class initiates a modal dialog, enabling the user to select the OLE object to be loaded into the control.

The code fragment in the following example shows the use of the **showInsertForm** method.

```
beginTransaction;
if oleReview1.showInsertForm(1, null, "All File |*.*|Notepad
             (*.txt)|*.txt") <> -1 then
    create obj;
    obj.copy(oleReview1.oleObject);
endif;
if oleReview2.showInsertForm(1, null, "All File |*.*|Notepad
             (*.txt)|*.txt") <> -1 then
    create obj;
    obj.copy(oleReview2.oleObject);
else
    create obj;
    obj.copy(oleReview3.oleObject);
endif;
commitTransaction;
```

The **allow** parameter values are listed in the following table.

Verb	Description
0	**embed & link** option enabled (new)
1	**embed** option only allowed
2	Linked object only allowed

The **initDir** parameter specifies the initial directory for browse options. The initial directory defaults to the current directory if an empty string is passed.

The **filter** parameter specifies the file type filter available during browse options. The filter defaults to all files if an empty string is passed.

The **showInsertForm** method returns the values listed in the following table.

Value	Description
-1	The user cancelled the dialog
1	An embedded object was selected
2	A link object was selected

If the user selected an OLE object, that OLE object is loaded into the **oleObject** property of the **OleControl** class.

Note When an OLE object is selected, the OLE server handling that object is initiated asynchronously. The **showInsertForm** method completes before the OLE object is loaded into the control.

showModal

Signature `showModal(): Integer;`

The **showModal** method of the **Form** class makes the form visible in its current state; that is, minimized, maximized, or normal. (Use the **windowState** property to control the state of the form.) The first **showModal** statement for a created form executes its **load** event if it has not already been executed.

Setting the **visible** property to **false** within the **load** event is ignored when the **showModal** method is performed. When a modal form is displayed, no logic after the **showModal** method is executed until the form is unloaded or made invisible. See also "Windows Events and JADE Events", later in this document.

The **showModal** method is not supported for a form defined as a Web page. If the **showModal** method is invoked for a Web page form, an error is raised at run time or the behavior of the **show** method is adopted, depending on the setting of the **Treat showModal as show** check box in the **Web Options** sheet of the Define Application dialog of your JADE application. (For details about interfacing to the Internet, see "Implementing Web Applications", in the *JADE Web Application Guide*.)

When displaying a modal form, no user input (from the keyboard or mouse) can occur in any other form until the modal form is unloaded. The logic must unload a modal form (usually in response to some user action) before further user input can occur. MDI forms cannot be shown modally. Although other forms in your application are disabled when a modal form is displayed, other applications are not.

The modal form is not deleted until the method that creates it is terminated (even when an unload is performed). User logic can therefore access any portion of the form and its controls, except that no events for that form occur.

The **showModal** method returns the value of the **modalResult** form variable, which can be set in the called modal form to indicate an action to be taken upon return from the form. The form must have been created before it can be shown. The **create** method constructs the windows and menus for the forms and its controls, as defined in the JADE development environment.

Unlike the **show** method, the logic does not return to the statement following the **showModal** method until after the form is unloaded or made invisible. Other forms cannot receive user input (from the keyboard or mouse) while that form is active.

Note The **showModal** method cannot be invoked from a server method.

The methods in the following examples show the use of the **showModal** method.

```
listProducts_dblClick(listBox: ListBox);
vars
    form : ProductDisplay;
begin
    if listProducts.itemText[listProducts.listIndex] <> "" then
        create form;
        form.fault := app.myCompany.allProducts.getAtKey(getProductNumber);
        form.centreWindow;
        form.showModal;
    endif;
end;

callCDEntry(doDisplay: Boolean) updating;
vars
    cdentry : CDEntry;
    str     : String;
begin
```

```
        str := cd.sendString("info identity");
        create cdentry;
        cdentry.doDisplay    := doDisplay;      // display if present
        cdentry.cdIdentity   := str.Integer;
        cdentry.numberTracks := numberOfTracks;
        cdentry.showModal;
        // update cdplayer data
        userObject := cdentry.userObject;
        displayCDInfo;
    end;
```

showMySheet

Signature `showMySheet();`

The **showMySheet** method of the **Control** class provides a control with the ability to make the sheet on which it is placed the top sheet of a **Folder** control. The control does not have to be a direct child of the folder.

Folders provide the ability to share the same screen space for a number of control images.

The sheets of the folders are special group boxes on which controls are placed. The caption of each sheet is displayed in selectable tabs above the images. Only one sheet is displayed at any time.

If the control is not on a sheet, this method does nothing. If the sheet that contains this control is not visible, the sheet is then made visible, as shown in the code fragment in the following example.

```
    if not text1.isMySheetVisible then
        text1.showMySheet;
    endif;
```

showPropertyPage

Signature `showPropertyPage(nam: String): Boolean;`

The **showPropertyPage** method of the **Window** class is called by the Painter when the **hasPropertyPage** method indicates that the window has its own property-editing dialog for editing the property specified in the **nam** parameter. If the property name is an empty string, the global Properties dialog is requested. The **showPropertyPage** method returns whether property changes were made by the user in the dialog, so that the Painter knows whether to save the changed data.

The Painter calls the **saveProperties** method when the edited form is being saved, so that the window can save data edited in the dialog.

The method in the following example shows the use of the **showPropertyPage** method.

```
    showPropertyPage(nam: String): Boolean updating;
    vars
        form              : MyTableForm;
        result, maxIndex  : Integer;
        loopIndex         : Integer;
    begin
        if nam = 'layout' then
           sizeColumnArrays;
           create form transient;
           form.transientTable := self;
           result := form.showModal;
```

```
            if result = 0 then
               if persistTable <> null then
                  beginTransaction;
                  persistTable.columnHeadingArray.clear;
                  persistTable.columnWidthArray.clear;
                  persistTable.columnVisibleArray.clear;
                  persistTable.columnPictureArray.clear;
                  persistTable.columnLengthArray.clear;
                  persistTable.columnAlignmentArray.clear;
                  persistTable.columnInputTypeArray.clear;
                  self.columnHeadingArray.copy(persistTable.columnHeadingArray);
                  self.columnWidthArray.copy(persistTable.columnWidthArray);
                  self.columnVisibleArray.copy(persistTable.columnVisibleArray);
                  self.columnPictureArray.copy(persistTable.columnPictureArray);
                  self.columnLengthArray.copy(persistTable.columnLengthArray);
                  self.columnAlignmentArray.copy(
                                       persistTable.columnAlignmentArray);
                  self.columnInputTypeArray.copy(
                                       persistTable.columnInputTypeArray);
                  persistTable.columnHeight := self.columnHeight;
                  myCollectionOid := null;
                  commitTransaction;
               endif;
               layout := 'Defined';
               buildTableColumns;
               return true;
            else
               return false;
            endif;
         else
            return inheritMethod(nam);
         endif;
     end;
```

Notes Although this method is primarily used for ActiveX controls, you can also use it for subclassed (user-defined) controls. If you want your subclassed controls to have their own property-editing dialog, you must reimplement the **showPropertyPage** method so that it can be called by Painter.

The initiated dialog runs under the JADE application (**app** is the JADE application).

stepRelative

Signature `stepRelative(increment: Integer);`

The **stepRelative** method of the **MultiMedia** class moves the current position in the content forwards or backwards by the amount specified in the **increment** parameter. Specify the increment in the units of the **timeFormat** property.

See also the **position** property.

startDrawingCapture

Signature `startDrawingCapture();`

The **startDrawingCapture** method of the **Picture** class causes the picture control to enter a special mode in which each subsequent mouse down on the control draws a dot. While the mouse is down, each mouse move event draws a line between the previous point and the current mouse position.

When the method is called, any previous drawing on the control is cleared and the value of the **autoRedraw** property is set to **true**. All subsequent user drawing activity is captured and saved.

The points and lines are drawn using default values for the **foreColor** property (**Black**), the **drawStyle** property (**DrawStyle_Solid**), and the **drawWidth** property (**1**) of the picture.

While the picture control is in this special mode, any defined **mouseDown**, **mouseUp**, **click**, and **dblClick** event methods on the control are not called.

Call the **stopDrawingCapture** method to exit from the special drawing mode.

stop

Signature `stop; (MultiMedia)`

`stop; (Picture)`

The **stop** method of the **MultiMedia** class stops playing or recording of the medium associated with the control. (See also the **MultiMedia** class **pause** method.)

The method in the following example shows the use of the **stop** method.

```
stop_click(btn: Button input) updating;
begin
    if cd.getMode = MultiMedia.Mode_Playing then
        cd.stop;
        cd.position := cd.getStartPosition;
    endif;
end;
```

The **stop** method of the **Picture** class causes the current animation process started by the **Picture** class **play** method to be terminated. If the **play** method is not in effect, this method does nothing.

When running the JADE application in thin client mode, this method executes on the presentation client by default.

stopDrawingCapture

Signature `stopDrawingCapture(): Binary;`

The **stopDrawingCapture** method of the **Picture** class turns off a special drawing mode that was entered using the **startDrawingCapture** and returns a binary image of the client area of the control.

A 1-bit monochrome **png** image is returned if the **Picture** control has no defined **picture** property, the effective value of the **backColor** property is white, and the value of the **foreColor** property is black. If any of these conditions is false, a 24-bit PNG image (.**png**) is returned.

The **stopDrawingCapture** method sets the value of the **autoRedraw** property to **false** and deletes the saved drawing history. When the control is repainted after calling the **stopDrawingCapture** method, the drawn image is not displayed.

If you want a different image style, use the following actions instead of calling the **stopDrawingCapture** method.

```
bin := picture1.createPictureAsType(....);
picture1.autoRedraw := false; // also turns off the special drawing mode
```

After the special drawing mode is turned off, any defined **mouseDown**, **mouseUp**, **click**, and **dblClick** event methods for the picture are called if they occur. If the **stopDrawingCapture** method is called and the control is not in the special drawing mode, an exception is not raised and the result is equivalent to calling the **createPictureAsType** method.

tabNext

Signature `tabNext();`

From the control that has the focus, the **tabNext** method of the **Form** class tabs to the next control that is visible and can receive the focus, in the tab order that is enabled. This method performs the same action as the Tab key.

tabPrior

Signature `tabPrior();`

From the control that has the focus, the **tabPrior** method of the **Form** class tabs to the prior control that is visible and can receive the focus, in the tab order that is enabled. This method performs the same action as the Shift+Tab key combinations.

undo

Signature `undo(): Boolean; (JadeRichText)`

`undo(): Integer; (JadeTextEdit)`

The **undo** method of the **JadeRichText** class undoes the last edit operation in the receiver control and returns **true** if the operation was undone or it returns **false** if the undo operation failed. You can undo up to 100 edit or format actions in a rich text control.

The **undo** method of the **JadeTextEdit** class undoes the last operation that was undone in the receiver control and returns zero (**0**). There is no limit to the number of undo actions in a text edit control.

unloadForm

Signature `unloadForm(): Boolean;`

The **unloadForm** method of the **Form** class attempts to unload the form and returns whether the unload operation was successful, as listed in the following table.

Return Value	Description
true	The form was unloaded
false	The request is denied by the **queryUnload** event generated by this method call

The form must have been created before it can be unloaded.

Use this method to cause the **queryUnload** and **unload** events of the form to be executed, even if the unload is performed in the **load** event of the form. The **queryUnload** method may reject the unload operation. If a form is unloaded before it is displayed and the **load** event has not been executed, the **queryUnload** and **unload** events are not executed.

If a **queryUnload** method is aborted, **queryUnload** is not called for any other forms affected by the unload and all affected forms are unloaded without calling the **unload** method. If an **unload** method is aborted, all other affected forms are unloaded without calling the **unload** method.

Unloading a form queues the window for deletion, but if another window is deleted before the next idle point, previously queued deleted windows are re-evaluated for deletion. If the queued window or its children have no outstanding Windows message, there are no incomplete event methods for that window or its children, and the method that created the window has exited or the window was deleted, the physical window is deleted.

If the form is defined as a Web page, the form is removed from the list of open forms maintained by the **WebSession** class. If the form is an MDI child and it is the last child of the default MDI frame form (supplied by JADE in **jade.exe**), the frame is also unloaded. If the form is the last running form, the application is also terminated.

The methods in the following examples show the use of the **unloadForm** method.

```
buttonUnload_click(btn: Button input) updating;
begin
    // Deletes the data and unloads the form.
    statusLine1.caption := "Deleting data...";
    statusLine1.refreshNow;
    beginTransaction;
        if numbers <> null then
            NumberDict.instances.purge;
            Number.instances.purge;
        endif;
    commitTransaction;
    unloadForm;
end;

cancelButton_click(btn: Button input) updating;
begin
    if app.checkForTransients(self) then
        app.msgBox("Transients remain", 'Warning', MsgBox_Information_Icon);
    endif;
    unloadForm;
end;
```

update

Signature `update();`

The **update** method of the **OleControl** class initiates the server application to ensure that the OLE object is up-to-date. The image is refreshed and stored if it is not up-to-date.

updateAppSettings

Signature `updateAppSettings(settings: String);`

The **updateAppSettings** method of the **JadeTextEdit** class adds one or more entries specified in the **settings** parameter to the application text editor settings table.

Use the application settings table to add to or override the default global settings.

Use the **applySettings** method to apply the combined application a global settings to the receiver control, as shown in the example in the following code fragment.

```
jte.updateAppSettings("smart.indent.words.*.jade=if while foreach");
jteSource.applySettings();
```

See also the **Application** class **getJadeTextEditGlobalSettings**, **getJadeTextEditOneSetting**, and **updateJadeTextEditAppSettings** methods and the **JadeTextEdit** class **applySettings** method.

useImage

Signature `useImage(image: Binary);`

The **useImage** method of the **Picture** class assigns an image to the picture of the picture control without creating a transient copy of that image, therefore avoiding the possibility of filling transient cache with a copy of the image, particularly for very large images. The result is the same as the **picture1.picture := <*binary*>** logic except that a copy of the image is not created in transient cache.

Note Calling the **useImage** method or assigning to the **picture** property discards the previous image. Both these logic statements update the same GUI object so that only the last set image is retained.

Calling the **useImage** method also causes the **picture** property to be set to **null**.

The following code fragment is an example of the use of this method.

```
vars
    bin : Binary;
begin
    bin := getImage; /* somehow retrieve the image; for example,
                        app.loadPicture */
    picture1.useImage(bin);
    /* Note that the following statement will then return 'null', even
       though the image has been set above */
    bin := picture1.picture;
```

usesFiles

Signature `usesFiles(): Boolean;`

The **usesFiles** method of the **MultiMedia** class returns **true** if the data storage used by the device is a file. If the storage is not a file, this method returns **false**.

If the **useDotNetVersion** property is set to **true**, the **usesFiles** method returns a fixed value of **true**.

windowToScreen

Signature
```
windowToScreen(x: Real io;
               y: Real io);
```

The **windowToScreen** method of the **Window** class converts a position relative to the left and top of the window into an absolute position on the screen. By using the **screenToWindow** method, this position can then be converted into a position relative to another window.

See also the **screenToWindow** method for an example of the use of these methods.

writeHTML

Signature `writeHTML();`

For report-type Web pages, the **writeHTML** method of the **Frame** class generates HTML code when a Web session is active.

If there is no active Web session, this method outputs the frame to the printer. For details, see the **print** method of the **Printer** class.

zOrder

Signature `zOrder(pos: Integer);`

The **zOrder** method of the **Window** class places a specified form or control at the front or back of the z-order within its graphical level. This affects only windows that are siblings. (Sibling controls that have the **alignContainer** property do not occupy the same area on their parent. If so, use the **visible** property to control the window that is currently displayed.)

For non-MDI forms, this method sends the form to either the front or the back of the screen, depending on the value of the **pos** parameter. Consequently, forms can be displayed in front of or behind other running applications. The **pos** parameter indicates the position of the control relative to other controls. If the position is **1**, the control is positioned at the front of the z-order. If the position is zero (**0**), the control appears at the back of the z-order.

Within MDI forms, the **zOrder** method sends MDI child forms to the front or the back of the MDI client area, depending on the value of the **pos** parameter.

When bringing a control to the front, if the control is already above all other overlapping siblings, the request is ignored. Conversely, when pushing a control to the back, if the control is already below all other overlapping siblings, the request is ignored. (The **zOrder** method does not generate a **paint** event if the position is unchanged.)

The methods in the following examples show the use of the **zOrder** method.

```
btnShowPicture_click(btn: Button input) updating;
begin
    if picture1.visible = false then
        picture1.visible := true;
        picture1.zOrder(1);
        buttonShowPicture.caption := "Hide Picture";
    else
        picture1.visible := false;
        buttonShowPicture.caption := "Show Picture";
    endif;
end;

btnInvestors_click(btn: Button input) updating;
vars
    iqForm : InvestorInquiry;
    form   : Form;
begin
    form := app.getForm("InvestorInquiry");
    if form = null then
        create iqForm;
        iqForm.centreWindow;
        iqForm.show;
    else
        iqForm := form.InvestorInquiry;
        iqForm.setFocus;              // set focus to this form
        iqForm.zOrder(1);
    endif;
end;
```

Notes A child is always drawn on top of its parent.

For details about the order of controls on touchscreens when accessibility is set, see "Control Order on Touchscreens" under "Changing the Runtime Tab Sequence", in Chapter 5 of the *JADE Development Environment User's Guide.*

It is not possible to handle the painting of transparent controls in the correct **zOrder** when it involves a mixture of controls that can be directly painted by JADE and those that can only be painted separately by Windows.

As a result, transparent sibling controls are always painted before any **JadeRichText**, **MultiMedia**, **JadeXamlControl**, **Ocx**, **OleControl**, **JadeDotNetVisualComponent**, or **ActiveXControl** controls, regardless of their **zOrder** settings.

Window, Form, and Control Events

This section describes the events generated by user interaction with the application in some way, usually by the mouse or a keyboard action.

One action may generate multiple events for that process. For example, clicking a different form can cause the following events.

- **deactivate** on the form losing the focus
- **deactivate** on the MDI frame form losing the focus
- **activate** on the new MDI frame form
- **activate** on the new form
- **mouseDown** on the new form or control (depending where it is clicked)
- **lostFocus** on the control that previously had the focus
- **gotFocus** on the control receiving the focus
- **mouseUp**
- **click**

Other development platforms take the approach of causing some of these events when properties of the controls or forms are altered; for example, causing the **change** event when text in a **TextBox** control changes. JADE does not do this. Sometimes these pseudo-events are an advantage, but sometimes they cause difficulties, with extra logic required to ignore the event, particularly when initializing the data in the control.

Tip Use the **Window** class **enableEvent** method to control at run time whether JADE logic associated with an event for a specific form or control is executed. You could use this method in thin client mode, for example, to speed up the data entry process for a **TextBox** control by disabling the **keyDown** event. (Event methods can be enabled or disabled in standard client mode and in thin client mode.)

In general, any events generated as a result of logic are not sent. The exception to this rule is that the following events are sent regardless of whether logic caused that event.

- **activate**
- **deactivate**
- **gotFocus**
- **lostFocus**

For details about the events that can be generated from JADE logic, see "Windows Events and JADE Events", in the following subsection.

When an event method defined at the **Control** class is deleted, the user is warned of this. If the deletion is confirmed, the event methods that correspond to this control event are also deleted.

Renaming a control event method automatically renames all event implementations.

Menu and control events have the menu or control causing the event passed as the first parameter. This need not be used if the event logic is designed only for a particular control. In this case, the logic can refer to the specific control. However, you should use the passed menu or control. By doing so, more than one control or menu can use the same event logic, where appropriate. In this situation, the event for these controls would have been mapped to the same method name.

Using the parameter is also of benefit when controls are cloned using the **loadControl** or **addControl** method. In this case, there is more than one control with the same name and type. They each share the same event methods. References to the control name directly from logic access only the original control created in the JADE development environment.

You can access all GUI properties and methods (which are marked as **clientExecution** methods) from a server method except for anything that brings up a modal-type dialog (that is, the common dialog class methods, the **app.msgBox**, and the **showModal** and **popupMenu** methods in the **Form** class). The other exceptions to this are the **app.doWindowEvents**, **app.checkPictureFile**, and **app.loadPicture** methods, which are executed relative to the server.

Caution Use of GUI methods and properties is *very* expensive in a server method. A **clientExecution** method requires that all transient objects passed to the server are passed back with the client execution (and passed back to the server after the client execution is complete). Accessing GUI properties and methods within a server execution should therefore be done only in exceptional circumstances.

Windows Events and JADE Events

Each application running in JADE is executed on its own thread and each thread effectively runs as a standalone program. When an application is idle, it waits for the arrival of Windows messages generated as a result of some user action; for example, the use of the keyboard or mouse, a timer expiry, or a message such as a JADE notification posted to the application. Any window of the current application can generate messages that are to be acted upon.

When a message is received, JADE internally performs the appropriate processing and generates the required JADE event calls.

Many messages generate a sequence of actions that are executed as part of the original message; for example, a **mouseDown** event on an inactive form may also generate an **activate** event on the new form, a **deactivate** event and a **mouseLeave** event on the old form, a **gotFocus** event on the control gaining focus, and a **lostFocus** and a **mouseLeave** event on the control losing focus.

When a message has begun to be processed, no further messages can be executed until logic again does a read on the input message queue. (For the exceptions to this rule, see "Window, Form, and Control Events", earlier in this document.) The actions that generate a read of this message queue are as follows.

- Processing of the current message, and all its resulting actions have been completed.
- When JADE logic executes a **form.showModal** and all of the logic associated with the loading and showing of that modal form has been completed. At this point, JADE starts a new windows loop, waiting and processing further window messages for the application until the form is unloaded.

  The method that executed the **showModal** method is suspended until that point.
- A message box is displayed. This is just another modal form, but one that does not have any JADE logic associated with it.
- JADE logic calls the **Application** class **doWindowEvents** method (**app.doWindowEvents**). This method immediately processes all messages for the entire application for a specified time, suspending the existing

method for that duration.

- An exception dialog is displayed, which is just another modal form.

Timer and notification events are executed only when the application is waiting for input.

When the application is in one of the wait situations described in the previous list, any window can generate events and any notifications and timers are processed, except that when an exception box is displayed, all timer and notification events are delayed until the exception is aborted or canceled. Paint events are still executed, otherwise the forms and controls would not be repainted.

Showing a form (including a message box) as modal has the following impact only.

- All other forms of that application are disabled until the modal form is unloaded. Those disabled forms can then generate timer, paint, and notification events only if specific logic makes changes to them; for example, an instance of a form is unloaded.
- The method issuing the **showModal** method call is suspended at that point, until the modal form is unloaded.

Some JADE events are generated immediately as a result of logic, as follows.

- Any logic that causes another form to be activated (including methods such as **show**, **showModal**, **showHelp**, **msgBox**, and **aboutBox**) generates **activate** and **deactivate** events.
- Any logic (including **setFocus**, **show**, **showModal** methods, and so on) that causes the focus to shift generates **gotFocus** and **lostFocus** events.
- The **refreshNow** method causes an immediate paint and generates a **paint** event unless the **autoRedraw** property is set. Any generated paint may also cause the **displayEntry** events associated with a list box collection to be executed when there are insufficient entries in the list box. (The **displayEntry** event does *not* re-evaluate existing entries in the list box.)
- The **show** and **showModal** methods usually generate the **load** event, but some other event such as a notification event can trigger the **load** event before the **show** or **showModal** method. (The **load** event is always executed before any other form event.)
- The **unloadForm** method generates a **queryUnload** event and an **unload** event, unless the **queryUnload** event rejects the closure.

  Unloading a form queues the window for deletion but if another window is deleted before the next idle point, previously queued deleted windows are re-evaluated for deletion. If the queued window or its children has no outstanding message, there are no incomplete event methods for that window or its children, and the method that created the window has exited or the window was deleted, the physical window is deleted.

  If a **queryUnload** method is aborted, **queryUnload** is not called for any other forms affected by the unload and all affected forms are unloaded without calling the **unload** method. If an **unload** method is aborted, all other affected forms are unloaded without calling the **unload** method.
- The **popupMenu** method generates the **select** and **click** events for the menu that is being displayed.
- The **terminate** instruction issues **queryUnload** and **unload** events to every form, unless a form rejects the termination.
- Changing properties of a **MultiMedia** control can cause **notifyMedia**, **notifyMode**, and **notifyPosition** events.
- **Ocx** events can be generated as a result of logic, but this is dependent on the control.
- User-generated events on subclassed controls can be generated by logic.

The JADE events that are generated immediately as a result of logic specifically do *not* include the following situations.

- Setting the **listIndex** property value of a **ListBox** or **ComboBox** control does not generate a **click** event.
- Changing the current row or column of a **Table** control does not generate a **rowColumnChg** event.
- Changing the **value** property of a **CheckBox** or **OptionButton** control does not generate a **change** or a **click** event.
- Changing the scroll position of a control does not generate a **scrolled** event.
- Changing the size or position of a **Form** or **Control** does not generate any event.

Note To reduce the complexity that can arise from generating these events immediately, it is your responsibility to handle these situations in your logic.

Call the **Application** class **doWindowEvents** method from a server method to process server notifications and timers.

activate

Signature `activate();`

The **activate** event of the **Form** class occurs when a form becomes the active window.

When an MDI frame containing MDI child forms is made active, the MDI frame and the active child both receive **activate** events (in that order).

When moving between child forms of an MDI frame, only the child forms receive an **activate** event. Similarly, if the last active child is destroyed, the MDI frame does not receive another **activate** event, as it is already active.

A form can become active by a user action, such as clicking a form or control, or by using the **show** or **setFocus** method in logic.

The **activate** event occurs only when a form is visible. For example, after a form loaded is created, it is not visible unless you use the **show** method or set the **visible** property of the form to **true**. The **activate** event occurs when moving the focus within an application (including MDI to non-MDI) and also to another application.

The method in the following example shows the use of the **activate** event.

```
activate() updating;
begin
    lCount.caption := theCollectionSize.String;
    self.resize;
end;
```

The handling of **activate** and **deactivate** events ensures that the following events occur once only.

- **deactivate** of the prior active form.
- **deactivate** of the MDI frame of the prior active form if it was an MDI child and the new form gaining focus is not the MDI frame or a child of the MDI frame.
- **activate** of the MDI frame if the form to be activated is an MDI child and the frame is not already active.
- **activate** of the form to be activated.

If another Windows or JADE application is activated while a JADE application is active, only **deactivate** events are generated for the deactivated JADE application.

If a JADE application is activated when another Windows or JADE application is active, only the **activate** events are generated for the activated JADE application.

browse

Signature
```
browse(browsebuttons: BrowseButtons input;
       browseType:    Integer);
```

The **browse** event occurs when the user presses the left mouse button over a browse button in a browse control. See the **BrowseButtons** class for details of the constants provided by browse buttons.

The value of the argument button reflects which browse button was depressed, as shown in the following table.

BrowseButtons Class Constant	Value	Button	Interpretation
Browse_First	1	Left	First
Browse_Prior	2	Second left	Prior
Browse_Next	3	Third left	Next
Browse_Last	4	Right	Last

cellInputReady

Signature
```
cellInputReady(table:        Table input;
               inputControl: Control input;
               cleft:        Real io;
               ctop:         Real io;
               cwidth:       Real io;
               cheight:      Real io);
```

The **cellInputReady** event of the **Table** control occurs after a cell containing a control referenced by the **cellControl** property becomes the current cell. The specified control is then resized to overlay the cell and is made visible before returning from this event method to initialize the **cellControl** control contained in the **inputControl** parameter for that cell.

Note If you create a table with a name 86 characters or longer and you then attempt to implement the **cellInputReady** event, a message box advises you that the name is too long. If this occurs, reduce the length of the **Table** control name before implementing the **cellInputReady** event.

The **cellInputReady** event method parameters are listed in the following table.

Parameter	Description
table	The **Table** control that has focus
inputControl	The control referenced by the **cellControl** property
cleft	The left position of the control
ctop	The top position of the control
cwidth	The width of the control
cheight	The height of the control

For details about automatically controlling a **ComboBox** or **TextBox** control assigned to a **Table** control as a **cellControl** property (for example, when performance is an issue when running in JADE thin client mode over a slow link), see the **Control** class **automaticCellControl** property.

change

Signature `change(control: control-type input);`

The **change** event indicates that the contents of a control have changed. The way in which the **change** event occurs varies with the control, as shown in the following table.

Control	Action
CheckBox	The **value** property is changed using the mouse or a keyboard action.
ComboBox	The text in the text box portion of the control has changed, only if the **style** property is set to **Style_DropDown** (0), **Style_Simple** (1), or **Style_DropDownComboList** (3) and the user changes the text portion of the combo box by using a keystroke. It does not occur if you change the **text** property from logic or by using arrow keys or the mouse.
Horizontal or vertical ScrollBar	The scroll box portion of the scroll bar has finished moving. This occurs when the user scrolls. It does not occur if you change the **value** property from logic.
JadeEditMask	The text contents of a text box have been changed by the user. It does not occur when you change the **text** property from logic (that is, dynamically).
JadeTextEdit	The text contents of the text editor have been changed by the user. It does not occur when you change the **text** property from logic (that is, dynamically).
OptionButton	The **value** property is changed using the mouse or a keyboard action. Both the option button that is turned on and the one that is turned off receive the **change** event.
Table	The current cell has an effective non-zero **inputType** property value and the user has changed the contents of the cell; that is, the check box is toggled by the mouse, space bar, or Enter key; the text box is changed by a key press, or a combo box list entry is selected.
TextBox	The text contents of the text box have changed. This occurs when the user changes the text. It does not occur when you change the **text** property from logic.

A **change** event can synchronize or coordinate the display of data between controls. For example, you can use the **change** event for a scroll bar to display a color scale based on the **value** property of the scroll bar.

click

Signature

```
click();                                 (Form)

click(control: control-type input);      (Control)

click(menuItem: MenuItem input);         (MenuItem)
```

The **click** event occurs when the user presses and then releases the left mouse button over an object.

For a menu item, this event occurs when the user clicks the menu item. If the menu item has a submenu, logic in this event allows the contents of the submenu to be changed before it becomes visible.

For a form, this event occurs when the user clicks a blank area or a disabled control. (The **click** event is lost if a form or another application is displayed between the **mouseDown** or **keyDown** event and the **mouseUp** or **keyUp** event. This occurs most commonly if **write** instructions invoke the display window or if the display window has *bring to top* enabled.)

For a control, this event occurs when the user:

- Clicks a control with the left mouse button. (Both the **click** and the **sheetChg** events are generated when the user clicks the tab of a sheet of a **Folder** or **Table** control that is not currently the top sheet.)
- Selects an item in a **ComboBox** or **ListBox** control, by pressing the arrow keys, by clicking the left mouse button, or by entering text into the edit area of a combo box that causes a list item to become selected.
- Presses the spacebar when a **Button**, **OptionButton**, or **CheckBox** control has the focus.
- Presses an access key combination for a control. For example, if the caption of a command button is **&Go**, pressing Alt+G triggers the event.
- Presses the Enter key and a button is defined as being the default.
- Presses the Esc key and a button is defined as the **Cancel** button.

For the **JadeTextEdit** class, a single **click** event positions the caret (insertion point) in the text editor, a double-click selects a word, a triple-click selects a line, and a single-click outside selected text cancels the selection.

For details about handling the **click** event method in check box, combo box, list box, and option button controls and user-defined subclasses of only those controls on Web pages, see "Handling Events on Web Pages", in the *JADE Web Application Guide*.

Typically, you attach a **click** event to a control or menu, to carry out commands and command-like actions, or to a change in the control.

You can use the **value** property of a control to test for the state of the control from logic (except for a push button control). When a **click** event is generated, JADE also calls the related events in the following order.

1. **mouseDown**
2. **mouseUp**
3. **click**

When you attach methods for these related events, ensure that their actions do not conflict.

Note To distinguish between the left, right, and middle mouse buttons, use the **mouseDown** and **mouseUp** events.

The methods in the following examples show the use of the **click** event method.

```
btnCancel_click(btn: Button input) updating;
begin
    unloadForm;
end;

theTable_dblClick(table: Table input) updating;
begin
    if xColSave <> 0 and yRowSave <> 0 then
```

```
            if theTable.row > 1 then
                btnChange.click(btnChange);
            endif;
        endif;
    end;
```

closeup

Signature `closeup(control: control-type input);`

When the list portion of a **ComboBox** control closes up, the **closeup** event is generated for the combo box object.

For **ComboBox** controls, as this event is called only after the **click** event when using the mouse and after the **keyPress** and **keyUp** events when the Enter key is pressed, users can modify the combo box entry selected during the **closeup** event.

A **Table** control also has the ability to be folded up so that it has only the height required to display the current row when it loses focus. The **closeup** event is generated when the table folds up.

This table control facility is controlled by the **dropDown** property. If the value of this property is **DropDown_Auto** (**2**), the table opens up when the table receives the focus. If the value is **DropDown_Click** (**1**), the user must specifically click the control to cause the event.

Use a **closeup** event to select the row (and columns) to be shown for the table in the folded up state.

contextMenu

Signature

```
contextMenu(conwin:  Window input;         (Form)
            mouse:   Boolean;
            x:       Integer;
            y:       Integer): Boolean;

contextMenu(control: control-type input;   (Control)
            conwin:  Window input;
            mouse:   Boolean;
            x:       Integer;
            y:       Integer): Boolean;
```

The **contextMenu** event occurs after the right **mouseUp** event and after the **keyUp** event.

If the control or form does not implement the event, the default processing continues.

This event returns **true** if processing is to continue or it returns **false** to terminate further processing.

For the **JadeRichText** control, the **contextMenu** event occurs after the right mouse click or a Shift+F10 key combination if the **contextMenuOptions** property has a value of **MenuOption_Custom**. If the control does not implement the event, no action is taken and the context menu is not displayed. This event returns **true** if the **JadeRichText** control displays its built-in menu or it returns **false** if it displays the custom popup menu.

The **contextMenu** event parameters are listed in the following table.

Parameter	Description
control	Control for which the event is being called

Parameter	Description
conwin	The window being right-clicked or the control with the focus when the keyboard generates the event
mouse	Contains **true** if the mouse generated the event or **false** if the keyboard generated it
x	Client left position in pixels of the window specified by the **conwin** parameter for the mouse click, or unused if mouse is **false**
y	Client top position in pixels of the window specified by the **conwin** parameter for the mouse click, or unused if mouse is **false**

The **contextMenu** event is called when the right mouse button is clicked over the window or when a control has focus and the accelerator menu key or the Shift+F10 shortcut keys of the application are pressed. You would most often implement this event to display a popup menu when the user performs that action, or you could implement it to suppress an existing automatically displayed popup menu (for example, that which occurs for a text box control).

For a **JadeRichText** or **JadeTextEdit** control, you would most often implement this event to display a popup menu when the user performs that action, or you could implement it to suppress the automatic display of the built-in context menu in a rich text control and display the custom menu instead.

The **contextMenu** event is also called for each parent of the window unless a previous child window in the parent-child chain terminated that process by returning a value of **false** from the event method. The **contextMenu** event has no impact on existing events and occurs in addition to the normal mouse and keyboard events.

Note Displaying a form, popup menu, or changing focus during mouse or keyboard events results in **contextMenu** events not being sent.

If you want to suppress the default context menu, you must implement the **contextMenu** event for the text box, and it should return **false**. This also means that the event is not called on any of its parents.

The method in the following example shows the use of the **contextMenu** event, positioning the popup menu on the currently selected list entry when the keyboard is used.

```
listBox1_contextMenu(control: Control;
                     conwin:  Window;
                     mouse:   Boolean;
                     x, y:    Integer): Boolean updating;
vars
    x1 : Real;
    y1 : Real;
begin
    x1 := x;
    y1 := y;
    if not mouse then  // if keyboard, use current listIndex value
        x1 := 20;
        if listBox1.listIndex < 0 then
            y1 := 5;
        else
            y1 := listBox1.positionTop(listBox1.listIndex);
        endif;
    endif;
    listBox1.windowToScreen(x1, y1);
    self.screenToWindow(x1, y1);   // convert x, y relative to form
    self.popupMenu(menuOptions, x1.Integer, y1.Integer);
```

```
        return false;                   // cancel further processing
    end;
```

dblClick

Signature `dblClick();` (Form)

`dblClick(`*`control: control-type`*` input);` (Control)

The **dblClick** event occurs when the user presses and releases the left mouse button and then presses and releases it again over an object.

For a **Form**, the **dblClick** event also occurs when the user double-clicks a disabled control or a blank area of a form.

For a **Control**, this event occurs when the user double-clicks a control.

For a **ListBox** control, the event occurs when the user double-clicks the text of an item. When the user double-clicks in a word in a **JadeTextEdit** control, that word is selected.

You can use a **dblClick** event for an implied action; for example, double-clicking an icon to open a window or document. This type of event is also useful for carrying out multiple steps with a single action; for example, double-clicking to select an item in a list box.

For those objects that receive mouse events, the events occur in the following order.

1. **mouseDown**
2. **mouseUp**
3. **click**
4. **dblClick**
5. **mouseUp**

If the **dblClick** event does not occur within the double-click time limit of the system, the object recognizes another **click** event. The double-click time limit may vary, because the user can use the Windows Control Panel to set the double-click speed. When you attach methods for these related events, ensure that their actions do not conflict.

Controls that do not receive **dblClick** events may receive two clicks instead of a **dblClick** event.

Note To distinguish between the left, right, and middle mouse buttons, use the **mouseDown** and **mouseUp** events.

deactivate

Signature `deactivate();`

The **deactivate** event of the **Form** class occurs when a different form becomes the active window.

When a non-MDI form or another MDI frame is then made active, both the active child and the MDI frame receive **deactivate** events (in that order). When moving between child forms of an MDI frame, only the child forms receive a **deactivate** event.

The **deactivate** event occurs when moving the focus within an application (including MDI to non-MDI) and to another application. The **deactivate** event does not occur when unloading a form.

The handling of **activate** and **deactivate** events ensures that the following events occur once only.

- **deactivate** of the prior active form.
- **deactivate** of the MDI frame of the prior active form if it was an MDI child and the new form gaining focus is not the MDI frame or a child of the MDI frame.
- **activate** of the MDI frame if the form to be activated is an MDI child and the frame is not already active.
- **activate** of the form to be activated.

If another Windows or JADE application is activated while a JADE application is active, only **deactivate** events are generated for the deactivated JADE application.

If a JADE application is activated when another Windows or JADE application is active, only the **activate** events are generated for the activated JADE application.

displayEntry

Signature

```
displayEntry(control:  control-name input;
             obj:      Any;
             lstIndex: Integer): String;
```

The **displayEntry** event occurs when a **ComboBox** or **ListBox** control is attached to a collection object and the text to be displayed for an entry in the collection is required.

When a collection is attached to a combo box or list box control by the **listCollection** method, an entry in the collection is accessed only when that entry is to be displayed in the list box for the control, or when logic accesses the object from the control (for example, accessing the **itemBackColor** property of the control).

You therefore do not need to write logic to fill the control, and only that portion of the control viewed or referred to needs to be accessed. For example, if the user views only the first 15 entries of the collection, the remaining entries (which may be hundreds or even thousands) are never accessed and substantial overheads are potentially saved. These processes *can* be done from standard logic, but where possible, this technique can save both logic and processing overheads.

To get the text to be displayed in the list box for each entry, the **displayEntry** event is called, which must return a string that is placed in the list box. If the **displayEntry** event is not implemented, a string informing the user that the event was not handled is placed in the list box instead.

If the **displayEntry** event returns an empty string, that entry is ignored and is not included in the list box. (The **listCount** method return value is adjusted when the **displayEntry** event method returns a null string.)

The object to be displayed is passed as a parameter (it must be cast to the required type for actual access). This would normally be a member of the **Object** class. However, as it is possible to attach a collection of primitive types to the list box, this object would then be of that primitive type (for example, an array of strings).

The text to be displayed is returned as a string. The position at which the entry is placed in the list box is also passed so that colors, levels, and so on can also be set.

Notes If the **sorted** property of the combo box or list box is set to **true**, the entire contents of the collection is accessed during the **listCollection** call, with calls to the **displayEntry** event for each object. The **lstIndex** parameter is always valid and can be used to set properties of the item that is being added, even when the list is sorted (for example, **itemBackColor**). For sorted lists, an empty entry is added, which is sorted into its correct position after the **displayEntry** event returns.

When the **listCollection** method is used with the **listCount** method, the value returned by the **listCount** method is the logical number of entries in the list box or combo box (that is, it returns a total of the number of entries for which the **displayEntry** event has already returned a string and the number of entries in the collection that are yet to be accessed).

When an exception is raised by the **displayEntry** or **displayRow** event method, the size of the collection is treated as being one less than the number of entries already processed. No further attempts are made to access the additional entries from the collection until a new **displayCollection** or **listCollection** method is executed against the control. The method in the following example shows the use of the **displayEntry** event method.

```
list1_displayEntry(listbox:  ListBox;
                   obj:      Any;
                   lstIndex: Integer): String updating;
begin
    return obj.Customer.surname & " " & obj.Customer.firstnames;
end;
```

displayRow

Signature

```
displayRow(control:   control-name input;      (Table)
           theSheet:  Integer;
           obj:       Object;
           theRow:    Integer;
           bcontinue: Boolean io): String;

displayRow(control:   control-name input;      (ComboBox or ListBox)
           obj:       Object;
           theRow:    Integer;
           bcontinue: Boolean io): String;
```

The **displayRow** event method occurs for each entry in the collection attached to the current sheet or list of a **ComboBox**, **ListBox**, or **Table** control by using the **displayCollection** method, to display the contents of the row. The **displayRow** event method parameters are listed in the following table.

Parameter	Description
control	The combo box, list box, or table that is calling the event method.
theSheet	The index of the sheet in the table.
obj	The object to be displayed, which must be cast to the appropriate type for actual access.
theRow	The row into which the entry is being placed (a blank row already exists).

Parameter	Description
bcontinue	Returns **false** if all entries in the collection beyond this point are to be ignored or returns **true** if processing is to continue. This **io** parameter is initialized to **true**. When a collection is attached to **ListBox** and **Table** controls using the **displayCollection** method and the collection is an **Array**, **Dictionary**, or **Set** and the **bcontinue** parameter returns **false**, the list size is adjusted and remembered but users can drag the list past that entry without the **displayRow** event being called for that entry, and JADE is therefore unaware that the display size was to be limited. You therefore cannot rely on the **bcontinue** parameter to limit the display up to the required point.

This event method returns a string that is used to set the contents of the row, starting with the first column regardless of whether it is a fixed column of a **Table** control or not. Tab characters must be placed between the text values of each cell in a table.

The value of the **itemObject** property for the row is set to the object value contained in the **obj** parameter.

If a **null** string (**""**) is returned, no row is added to the table or list.

The method in the following example shows the use of the **displayRow** event method.

```
tbProducts_displayRow(table:     Table input;
                      theSheet:  Integer;
                      obj:       Object;
                      theRow:    Integer;
                      bcontinue: Boolean io): String;
vars
    prod : Product;
begin
    prod := obj.Product;
    if prod.superseded then
        return null;
    endif;
    if prod.date < 010198.Date then
        bcontinue := false;        // have all the entries required
        return null;
    endif;
    return prod.code.String & Tab & prod.name;  // Return text for each cell
                                                // separated by a tab
end;
```

As this event method can set the **bcontinue** parameter to **false** if processing of further collection entries is not required and therefore provides the ability to terminate the load process, this event method enables you to perform a load of all products whose names start with the letter **s**, for example.

Note The scroll bar may not necessarily show the correct extent of scrolling available, as the number of entries to be displayed may not be known accurately.

If the **update** parameter of the **displayCollection** method is set to **true**:

- Deleting the collection results in the non-fixed rows of the sheet or list being deleted, and the collection is no longer associated with the sheet or list.
- Any changes to the collection cause the contents of the non-fixed rows of a sheet or list to be refreshed. New entries are inserted, deleted entries are removed, and the text of entries that are still valid is refreshed. Any other properties that are set (for example, the status of the **selected** property) are retained.

If the **update** parameter is set to **false**, the table is not updated when the collection changes and may contain outdated information.

Note The value of the **update** parameter must be **false** for transient collections, as JADE does not issue system notifications for the addition, change, or deletion of a transient collection.

When using the **displayRow** event method, you should also be aware of the following.

- Changes to the non-fixed rows of the table made outside the **displayRow** event method are permitted but are lost after scrolling.
- The collection object is saved as a row object for a table. You can use the **itemObject** property to retrieve the collection object, as shown in the method in the following example.

```
Table.accessRow(indx).itemObject;
vars
    prod    : Product;
    company : Company;
begin
    if listProducts.listIndex = -1 then
       app.msgBox("You must make a selection", "Error", 0);
       return;
    endif;
    beginTransaction;
       prod := listProducts.itemObject[listProducts.listIndex].Product;
       delete prod;
       listProducts.clear;
       company := Company.firstInstance;
       listProducts.listCollection(company.allProducts, true, 0);
    commitTransaction;
end;
```

- The **clear** method clears the list or non-fixed rows of the sheet and detaches the collection. (This behavior differs from that of tables that do not have an attached collection.)
- When the **update** parameter of the **displayCollection** method is set to **true**, the sheet or list has a **beginNotification** condition established (for all events). These events are also passed to the **sysNotify** event of the table or list.
- Each sheet of a table can have its own collection attached.
- The **topRow** property of a table or the **topIndex** property of a list is always set to **1** and cannot be changed.
- The **Table** class **dropDown** property is ignored.
- When an exception is raised by the **displayEntry** or **displayRow** event method, the size of the collection is treated as being one less than the number of entries already processed. No further attempts are made to access the additional entries from the collection until a new **displayCollection** or **listCollection** method is executed against the control.

docked

Signature

```
docked(dockbar: control-name input);     (JadeDockBar)

docked(dockcont: control-name input);    (JadeDockContainer)
```

The **docked** event method of the **JadeDockBar** or **JadeDockContainer** class is called when the user has caused a control to be docked.

Use this event method to handle any implications that are not handled automatically. The parent and new alignment of the docking control are already set when this event method is called.

Note This event method is called even if the user drags the control to a new position within its existing parent. This event method is also called when a floating form is closed by its **Close** button after the parent of the control has been set to the JADE form and the control has been made invisible.

For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document. See also the docking examples under the **JadeDockContainer** class **allowDocking** property, earlier in this document.

dragDrop

Signature

```
dragDrop(win:      Window input;          (Form)
         x:        Real;
         y:        Real);

dragDrop(control: control-name input;    (Control)
         win:      Window input;
         x:        Real;
         y:        Real);
```

The **dragDrop** event occurs when a dragged window is dropped over a form or control belonging to the same application. The drop part of the drag and drop process occurs when the user causes a **mouseUp** event (that is, drag mode is terminated when the left, right, or middle mouse button is released) or when the logic sets the **dragMode** property of the window that is being dragged to **DragMode_Drop** (2).

See also the **dragListIndex**, **dragColumn**, **dragRow**, and **dragSheet** properties.

No **dragDrop** event is issued if the drag process is terminated by setting the **dragMode** property to **DragMode_None** (0).

The **win** parameter specifies the window that is being dragged.

The **x** and **y** parameters specify the physical horizontal and vertical positions where the window was dropped, respectively. The positions are in the units of the **scaleMode** property of the form or control.

For details about converting to logical positions within the window that has been scrolled, see the **scrollHorzPos** property.

dragOver

Signature

```
dragOver(win:      Window input;           (Form)
         x:        Real;
         y:        Real;
         state:    Integer);

dragOver(control: control-type input;    (Control)
         win:      Window input;
         x:        Real;
         y:        Real;
         state:    Integer);
```

The **dragOver** event occurs for each form or control over which a window is dragged.

A window is placed into drag mode by setting its **dragMode** property to **DragMode_Drag** (1).

The dragged-over window receives an indication with each **dragOver** event as to whether the drag process is entering, continuing, or leaving the window.

The **win** parameter specifies the window that is being dragged. The **x** and **y** parameters specify the physical horizontal and vertical positions where the window was dragged, respectively.

The positions are in the units of the **scaleMode** property of the form or control. For details about converting to logical positions within the window that has been scrolled, see the **scrollHorzPos** property.

The values of the **state** parameter are listed in the following table.

Window Class Constant	Value	Description
DragOver_Enter	0	If the drag process has just entered this window
DragOver_Continue	1	The second and subsequent calls over this window
DragOver_Leave	2	The drag process has just left this window

See also the **dragListIndex**, **dragColumn**, **dragRow**, and **dragSheet** properties.

For details about getting and setting the handling of the **dragOver** event to optimize performance on presentation clients in JADE thin client mode, see the **Application** class **getMouseMoveTime** and **setMouseMoveTime** methods.

dropDown

Signature `dropDown(combobox: ComboBox input): Boolean;`

The **dropDown** event occurs when the list portion of a **ComboBox** control is about to drop down. It does not occur if the **style** property of a combo box is set to **Style_Simple** (1).

The **dropDown** event returns the values listed in the following table.

Return Value	Action
True	Opens a drop-down list
False	Ignores the drop-down request

Use a **dropDown** event to make final updates to a combo box list before the user makes a selection. This enables you to add or remove items from the list, by using the **addItem** or **removeItem** methods. This flexibility is useful when you want some interplay between controls; for example, if what you want to load into a combo box list depends on what the user selects in an option button group. See also the **closeDropDown**, **isDroppedDown**, and **showDropDown** methods.

firstChange

Signature `firstChange(`*`control: control-type`*` input);`

The **firstChange** event of the **TextBox** control is generated when the user performs keyboard or cut and paste actions that result in any of the following conditions.

- The text box goes from an empty to a non-empty state
- The text box goes from a non-empty to an empty state
- The first change made by the user after the text has been set by the form build process or by JADE logic

Use this event to move logic from other key events when that logic is relevant only to the first time the text changes; for example, when the logic is recording that the text content has changed. As a result, removal of the other key events reduces the number of events that must be sent and processed for each key that is pressed.

The **firstChange** event method of the **JadeEditMask** class occurs when the user makes the first change on the displayed text or the **isEmpty** status of the control changes. It does not occur when you change the **text** property from logic (that is, dynamically).

For the **JadeRichText** or **JadeTextEdit** control class, a **firstChange** event occurs after a user change but before the **change** event, in the following cases.

- The first time the contents change
- The contents of the control are deleted
- The contents of the control change from being empty to having some content
- The **text** property or **textRTF** property changes the contents of the control (the **textRTF** property applies only to rich text control)
- After the value of the **readOnly** property changes from **true** to **false**

Note The **firstChange** event method for the **TextBox**, **JadeEditMask**, **JadeRichText**, **JadeTextEdit**, and **Form** classes is not available from a Web browser.

The **Form** class **firstChange** event is generated when the first change to the values listed in the following table is made by a user to a control on a form.

Control Class	Change
TextBox	Change to the value of the **text** property
JadeEditMask	Change to the value of the **text** property
JadeRichText and JadeTextEdit	After a user change but before the **change** event (same rules as those that apply to the **TextBox** class **firstChange** event)
CheckBox	Change to the value of the **value** property

Control Class	Change
OptionButton	Change to the value of the **value** property
ComboBox	Different entry selected (this does not apply to the text box portion of the combo box unless the **text** property change causes a different list entry to be selected)
ListBox	Different list entry selected
Table	Cell updated by the **Control** class **automaticCellControl** property, including default **inputType** property entry of **CheckBox**, **TextBox**, **JadeEditMask**, and **ComboBox** controls (but not when a different row or column is selected)

The control that has been changed is passed as the parameter to the event. By default, the **firstChange** event is generated once only, when the first of any of the changes listed in the above table has been made on a form. You can then determine that a data change has been made to a form without having to monitor changes to every control on that form.

The **Form** class **resetFirstChange** event enables you to reset all **firstChange** events on the form. In addition, you could use the **resetFirstChange** method inside the **Form** class **firstChange** event when the first change has been made to a control that is not to be counted as the first data change.

The **Control** class **resetFirstChange** event resets the **firstChange** event status of the control and all children of the control. As only the **TextBox**, **JadeRichText**, and **JadeEditMask** controls have a **firstChange** event, all other controls ignore the method other than to call the method on any children.

floated

Signature `floated(dockbar: `*`control-name`*` input);` (JadeDockBar)

`floated(dockcont: `*`control-name`*` input);` (JadeDockContainer)

The **floated** event of the **JadeDockBar** or **JadeDockContainer** class is called when the user has caused a control to float.

Use the **floated** event method to handle any implications that are not handled automatically.

The parent of the docking control is set to **null** before the event is called.

For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document. See also the docking examples under the **JadeDockContainer** class **allowDocking** property, earlier in this document.

formMove

Signature `formMove();`

The **formMove** event occurs when a **Form** is moved on the screen. This event is not called under the following circumstances.

- When logic changes the **left** or **top** property
- When logic calls the **move** method

The movement of a form is not normally of concern to the application, unless the application is a screen painter of some type.

As the **left** and **top** properties can be accessed directly, this event has no parameters.

gotFocus

Signature

```
gotFocus(cntrl: Control input);              (Form)

gotFocus(control: control-type input);   (Control)
```

The **gotFocus** event occurs when a control receives the focus, either by a user action such as tabbing to or clicking the object or by changing the focus in logic using the **setFocus** method. A copy of this event is first sent to the form.

This event is not returned for a form, as a form does not have focus unless there are no input controls. Use the **activate** event instead.

Typically, a **gotFocus** event method specifies the actions that occur when a control first receives the focus. For example, by attaching a **gotFocus** method to each control on a form, you can guide the user by displaying brief instructions or status bar messages.

You can also provide visual cues, by enabling, disabling, or showing other controls that depend on the control that has the focus.

Note An object can receive the focus only if its **enabled** and **visible** properties are set to **true**.

When focus is gained by a control, a **gotFocus** event is *also* sent to the form, identifying the control that gained focus. The form itself never gains the focus, but this event (in conjunction with the **lostFocus** event) provides a centralized place where all focus change events can be monitored in a similar way to the **keyDown**, **keyPress**, and **keyUp** events of the **Form** class.

keyDown

Signature

```
keyDown(keyCode: Integer io;              (Form)
        shift:   Integer);

keyDown(control: control-type input;    (Control)
        keyCode: Integer io;
        shift:   Integer);
```

The **keyDown** event occurs when the user presses a key while an object has focus. (To interpret ANSI characters, use the **keyPress** event.) This event uses the parameters listed in the following table.

Parameter	Description
keyCode	A key code; for example, **J_key_F1** (F1 key) or **J_key_Home** (Home key).
shift	The state of the Shift, Ctrl, and ALT keys when the key was pressed. This integer is a combination of values, as follows. ■ To test if the Shift value (1) is set, use **if shift.bitAnd(1) <> 0 then** ■ To test if the Ctrl value (2) is set, use **if shift.bitAnd(2) <> 0 then** ■ To test if the Alt value (4) is set, use **if shift.bitAnd(4) <> 0 then**

You can use the **Window** class constants listed in the following table to test the state of the Alt, Ctrl, and Shift keys of the **shift** parameter.

Window Class Constant	Bit Value
KeyState_Alt	#4
KeyState_Ctrl	#2
KeyState_Shift	#1

The **keyDown** event of the **Form** class first receives all keystrokes for all controls, followed by the object with the focus. Although the **keyDown** event can apply to most keys, it is most often used for:

- Extended character keys such as function keys
- Navigation keys
- Combinations of keys with standard keyboard modifiers
- Distinguishing between the numeric keypad and regular number keys

Tip Use the **Window** class **enableEvent** method to control at run time whether JADE logic associated with an event for a specific form or control is executed. You could use this method in thin client mode, for example, to speed up the data entry process for a **TextBox** control by disabling the **keyDown** event. (Event methods can be enabled or disabled in standard client mode and in thin client mode.)

Use the **keyDown** event for keyboard handlers if you need to respond to the pressing of a key.

When using a numeric text box (the **inputType** property is set to **InputType_TextNumeric**), the **Form** and **TextBox** class **keyDown** events are still fired when the entered key is invalid. This enables the form to process keys such as the Enter key in the **Form**::**keyDown** or **keyPress** events and the control to process these keys in its **keyDown** event, therefore allowing application users to use the numeric keypad when a form requires a large amount of numeric data entry.

Note Changing the **keyCode** parameter effects only the **Form** class. For control classes, Windows has already predetermined the actions taken.

keyPress

Signature

```
keyPress(keyCharCode: Integer io);              (Form)

keyPress(control:     control-type input;       (Control)
         keyCharCode: Integer io);
```

The **keyPress** event occurs when the user presses and releases an ANSI key.

The **keyPress** event uses the parameter listed in the following table.

Parameter	Description
keyCharCode	Returns a standard numeric ANSI key code. The **keyCharCode** parameter is passed by reference; changing it sends a different character to the object. (See the global constants in the **KeyCharacterCodes** category for a list of the global constants for printable key codes.)

Changing the value of the **keyCharCode** parameter changes the character that is displayed. Changing the **keyCharCode** parameter to zero (**0**) cancels the keystroke so that the object receives no character.

The form first receives the **keyPress** event and then the object with the focus. A **keyPress** event can involve any printable keyboard character, the Ctrl key combined with a character from the standard alphabet or a special character, and the Enter or Backspace key. Use a **keyPress** event for intercepting keystrokes entered in a text box or combo box. It enables you to immediately test keystrokes for validity or to format characters as they are typed.

Use the **keyDown** and **keyUp** events to handle any keystroke not recognized by the **keyPress** event; for example, function keys, editing keys, navigation keys, and any combinations of these with keyboard modifiers. Unlike the **keyDown** and **keyUp** events, the **keyPress** event does not indicate the physical state of the keyboard but passes a character.

Tip Use the **Window** class **enableEvent** method to control at run time whether JADE logic associated with an event for a specific form or control is executed. You could use this method in thin client mode, for example, to speed up the data entry process for a **TextBox** control by disabling the **keyPress** event. (Event methods can be enabled or disabled in standard client mode and in thin client mode.)

When using a numeric text box (the **inputType** property is set to **InputType_TextNumeric**), the **Form** class **keyPress** event is still fired when the entered key is invalid. This enables the form to process keys such as the Enter key in the **keyDown** or **keyPress** events of the **Form** class and the control to process these keys in its **keyDown** event, therefore allowing application users to use the numeric keypad when a form requires a large amount of numeric data entry.

The **keyPress** event interprets the uppercase and lowercase of each character as separate key codes, and therefore as two separate characters.

The **keyDown** and **keyUp** events interpret the uppercase and lowercase of each character by means of two parameters: **keyCode**, which indicates the physical key (therefore returning **"A"** and **"a"** as the same key), and **shift**, which indicates the state of Shift+ key and therefore returns **"A"** or **"a"**.

keyUp

Signature

```
keyUp(keyCode: Integer io;           (Form)
      shift:   Integer);

keyUp(control: control-type input;   (Control)
      keyCode: Integer io;
      shift:   Integer);
```

The **keyUp** event occurs when the user releases a key while an object has the focus. (To interpret ANSI characters, use the **keyPress** event.)

The **keyUp** event uses the parameters listed in the following table.

Argument	Description
keyCode	A key code; for example, **J_key_F1** (F1 key) or **J_key_Home** (Home key).
shift	The state of the Shift, Ctrl, and Alt keys when the key was pressed. This integer is a combination of values, as follows. ■ To test if the Shift value (1) is set, use **if shift.bitAnd(1) <> 0 then** ■ To test if the Ctrl value (2) is set, use **if shift.bitAnd(2) <> 0 then** ■ To test if the Alt value (4) is set, use **if shift.bitAnd(4) <> 0 then**

You can use the **Window** class constants listed in the following table to test the state of the Alt, Ctrl, and Shift keys of the **shift** parameter.

Window Class Constant	Bit Value
KeyState_Alt	#4
KeyState_Ctrl	#2
KeyState_Shift	#1

The **keyUp** event of the **Form** class first receives all keystrokes for all controls, followed by the object with the focus. Although the **keyUp** event can apply to most keys, it is most often used for:

- Extended character keys such as function keys
- Navigation keys
- Combinations of keys with standard keyboard modifiers
- Distinguishing between the numeric keypad and regular number keys

Tip Use the **Window** class **enableEvent** method to control at run time whether JADE logic associated with an event for a specific form or control is executed.

You could use this method in thin client mode, for example, to speed up the data entry process for a **TextBox** control by disabling the **keyUp** event. (Event methods can be enabled or disabled in standard client mode and in thin client mode.)

Use the **keyUp** event for keyboard handlers if you need to respond to releasing a key.

Note Changing the **keyCode** parameter only effects the **Form** class. For control classes, Windows has already predetermined the actions taken.

For details about the context menu event that occurs after the **keyUp** event, see the **contextMenu** event method. If a called window does not implement the **contextMenu** event, the default processing continues.

linkClicked

Signature
```
linkClicked(textbox: JadeRichText input;
            link:    String);
```

The **linkClicked** event of the **JadeRichText** class occurs when the user clicks on a URL within the text of the control.

The **link** parameter contains the URL text.

The following example attempts to open the URL specified by the **link** parameter in the default browser.

```
jrtResults_linkClicked(textbox: JadeRichText input; link: String) updating;
vars
    result : Integer;
begin
    result := call josShellExecute(getWindowHandle(), 'open', link, null, null, 1);
    if result <= 32 then
        app.msgBox('You do not have a default browser or the URL '
                   & link & ' could not be found', 'Error',
                   MsgBox_Exclamation_Mark_Icon);
        return;
    endif;
end;
```

load

Signature `load();`

The **load** event of the **Form** class is always the first event called for the form and its controls (except the **windowCreated** event for the controls).

The **load** event is normally called when the form is displayed using the **show** or **showModal** method.

If the form or one of its controls receives an event before the form being displayed (for example, a **sysNotify** event), the **load** event is called before processing that event.

For the startup form, a **show** method is issued automatically after the form is created.

Typically, use a **load** event to include initialization logic for a form; for example, specifying default settings for controls, indicating contents to be loaded into combo boxes or list boxes, and initializing form-level variables.

Note When you create methods for related events such as **activate**, **gotFocus**, **paint**, and **resize**, ensure that their actions do not conflict and that they do not cause recursive events.

If a JADE logic exception occurs during the **load** event and the user responds by selecting the **Abort** option, the form is destroyed without any further events being issued.

lostFocus

Signature

```
lostFocus(cntrl: Control input);             (Form)

lostFocus(control: control-type input);      (Control)
```

The **lostFocus** event occurs when a control loses the focus, either by a user action such as tabbing to or clicking another object or by changing the focus in logic using the **setFocus** method.

A copy of this event is first sent to the form. As a form does not have focus unless there are no input controls and this event is not returned for a form, use the **deactivate** event for a form.

Use the **lostFocus** event for verification and validation updates. Using the **lostFocus** event can cause validation to take place as the user leaves the control.

You can also use the **lostFocus** event to enable, disable, hide, or display other objects, as in a **gotFocus** event. You can also reverse or change conditions that you set up in the **gotFocus** event of an object.

When focus is lost by a control, a **lostFocus** event is *also* sent to the form, identifying the control that lost focus. The form itself never has the focus, but this event (in conjunction with the **gotFocus** event) provides a centralized place where all focus change events can be monitored in a similar way to the **keyDown**, **keyPress**, and **keyUp** events of the **Form** class.

mdiDocked

Signature `mdiDocked();`

The **mdiDocked** event occurs after a user docks an MDI child **Form** back into its MDI frame.

This event is not called if the **dockMdi** method is called to do the docking.

Applies to Version: 2020.0.01 and higher

mdiFloated

Signature `mdiFloated();`

The **mdiFloated** event occurs when an MDI child **Form** is floated by the user.

This event is not called if the **floatMdi** method is called to do the docking.

Applies to Version: 2020.0.01 and higher

mouseDown

Signature

```
mouseDown(button:  Integer;                (Form)
          shift:   Integer;
          x:       Real;
          y:       Real);

mouseDown(control: control-type input;     (Control)
          button:  Integer;
          shift:   Integer;
          x:       Real;
          y:       Real);
```

The **mouseDown** event occurs when the user presses a mouse button. The **mouseDown** event uses the parameters listed in the following table.

Parameter	Description
button	The button that was pressed to cause the event. ■ 1 (left button) ■ 2 (right button) ■ 3 (middle button)
shift	The state of the Shift, Ctrl, and Alt keys when the mouse button was pressed. This integer is a combination of values: ■ To test if the Shift value (1) is set, use **if shift.bitAnd(1) <> 0 then** ■ To test if the Ctrl value (2) is set, use **if shift.bitAnd(2) <> 0 then** ■ To test if the Alt value (4) is set, use **if shift.bitAnd(4) <> 0 then**
x	The current left physical position of the mouse pointer within the window. This is in the units of the **scaleMode** property of the form or control.
y	The current top physical position of the mouse pointer within the window. This is in the units of the **scaleMode** property of the form or control.

For details about converting the horizontal (**x**) and vertical (**y**) physical pixel positions to logical positions within the window that has been scrolled, see the **scrollHorzPos** property.

Use a **mouseDown** event to specify actions to occur when a specified mouse button is pressed. Unlike the **click** or **dblClick** event, the **mouseDown** event enables you to distinguish between the left, right, and middle mouse buttons. If a new form is created or if focus is shifted before the **mouseDown** event is completed, focus is not set. You can also write logic for mouse and keyboard combinations that use the Shift, Ctrl, and Alt keyboard modifiers.

See the **Window** class for details of the **MouseButton_** constants provided for window classes.

The following applies to both **click** and **dblClick** events.

- If a mouse button is pressed while the cursor is over a form or control, that object "captures" the mouse and receives all mouse events up to and including the last **mouseUp** event.

  This implies that the **x** and **y** mouse pointer coordinates specified by a mouse event may not always be in the client area of the object that receives them.

- If mouse buttons are pressed in succession, the object that captures the mouse after the first press receives all mouse events until all buttons are released.

Notes You can use a **mouseMove** or **mouseLeave** event to respond to an event caused by moving the mouse. The **button** parameter for the **mouseDown** event differs from the **button** parameter used for the **mouseMove** event. For the **mouseDown** event, the **button** parameter indicates exactly one button for each event; for the **mouseMove** event, it indicates the current state of all buttons.

Table control subclasses do not reference **mousePointer** in a **mouseDown** event to determine if the mouse is in the resize position. Use the **allowResize** property to specify whether users can resize the rows and columns of a table. (The **resizeColumn** and **resizeRow** event methods are called when a resize operation has been performed using the mouse on a table column or row, respectively.)

mouseEnter

Signature `mouseEnter();` (Form)

`mouseEnter(`*`control: control-type`*` input);` (Control)

The **mouseEnter** event occurs when the user moves the mouse onto a control or form.

Unlike the **mouseMove** event, which is generated continually as the mouse pointer moves across objects, the **mouseEnter** event occurs once only; that is, when the mouse moves onto the object.

To cause an event when the user moves the mouse pointer onto a control and then the mouse remains static for one second or longer (that is, the mouse pointer hovers over a control), use the **mouseHover** event.

Use this event method, for example, to draw the changed state of a button. (Use the **mouseLeave** event to restore the state of the button.)

mouseHover

Signature

```
mouseHover(control: control-type input;
           button:  Integer;
           shift:   Integer;
           x:       Real;
           y:       Real);
```

The **mouseHover** event occurs when the user moves the mouse onto a control and then the mouse remains static for one second or longer.

The parameters of the **mouseHover** event are listed in the following table.

Parameter	Description
button	The state of the mouse buttons as an integer, in which a bit is set if the button is down. ■ 0 (no button down) ■ 1 (left button) ■ 2 (right button) ■ 3 (middle button)
shift	The state of the Shift, Ctrl, and Alt keys when the button specified in the **button** parameter was pressed or released. A bit is set if the key is down. The **shift** parameter is a bit integer, with the least-significant bits corresponding to the Shift key (bit **0**), the Ctrl key (bit **1**), and the Alt key (bit **2**). The **shift** parameter indicates the state of these keys. Some, all, or none of the bits can be set, indicating which keys were pressed. For example, if both Ctrl and Alt were pressed, the value of shift would be **6**.
x	The current left physical position of the mouse pointer in the window. This is in the units of the **scaleMode** property of the form or control.
y	The current top physical position of the mouse pointer in the window. This is in the units of the **scaleMode** property of the form or control.

For details about converting the horizontal (**x**) and vertical (**y**) physical pixel positions to logical positions within the window that has been scrolled, see the **scrollHorzPos** property.

Notes This event can occur over the non-client parts of the control. The **x** and **y** positions are relative to the client area of the control and can be negative or greater than the client width and height. The event can also occur if a mouse button is down and the Shift or Ctrl key is down.

After the **mouseHover** event is generated, another event is generated for the same control each time the mouse is moved on the control and then becomes static again for one second or longer.

If the window implements the **mouseHover** event, bubble help is displayed after the **mouseHover** event has been executed. This allows the **mouseHover** event to set the **bubbleHelp** property text value that is appropriate for the mouse position; for example, a **ListBox** control the list entry that the mouse is over. The bubble help display is cancelled if the user moves the mouse away from the list item over which the mouse pointer was positioned.

Consider using the **mouseHover** event instead of the **mouseMove** event, as the **mouseHover** event can achieve what is required with one event instead of several **mouseMove** events.

Use the **mouseEnter** event to respond when the user moves the mouse pointer onto a control but the mouse does not then become static (that is, the mouse does not hover over the control).

Applies to Version: 2016.0.01 and higher

mouseLeave

Signature

```
mouseLeave();                                  (Form)

mouseLeave(control: control-type input);       (Control)
```

The **mouseLeave** event occurs when the user moves the mouse off a control or form. For example, you can use this event method when you have defined your own toolbar buttons to turn off button highlighting that was caused by the **mouseMove** event method.

mouseMove

Signature

```
mouseMove(button:  Integer;                 (Form)
          shift:   Integer;
          x:       Real;
          y:       Real);

mouseMove(control: control-type input;     (Control)
          button:  Integer;
          shift:   Integer;
          x:       Real;
          y:       Real);
```

The **mouseMove** event occurs when the user moves the mouse.

The parameters of the **mouseMove** event are listed in the following table.

Parameter	Description
button	The state of the mouse buttons as an integer, in which a bit is set if the button is down. ■ 0 (no button down) ■ 1 (left button) ■ 2 (right button) ■ 3 (middle button)
shift	The state of the Shift, Ctrl, and Alt keys when the button specified in the **button** parameter was pressed or released. A bit is set if the key is down. The **shift** parameter is a bit integer, with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). The **shift** parameter indicates the state of these keys. Some, all, or none of the bits can be set, indicating which keys were pressed. For example, if both Ctrl and Alt were pressed, the value of shift would be **6**.
x	The current left physical position of the mouse pointer in the window. This is in the units of the **scaleMode** property of the form or control.
y	The current top physical position of the mouse pointer in the window. This is in the units of the **scaleMode** property of the form or control.

For details about converting the horizontal (**x**) and vertical (**y**) physical pixel positions to logical positions within the window that has been scrolled, see the **scrollHorzPos** property.

The **mouseMove** event is generated continually as the mouse pointer moves across objects. Unless another object has captured the mouse, an object recognizes a **mouseMove** event whenever the mouse position is within its borders.

Note You can use the **mouseDown** and **mouseUp** events to respond to events caused by pressing and releasing mouse buttons, or the **mouseLeave** event to respond to the user moving the mouse off the control.

Any time you move a window inside a **mouseMove** event, it can cascade, as **mouseMove** events are generated when the window moves underneath the cursor. A **mouseMove** event can be generated even if the mouse is stationary.

The first **mouseMove** event received after left-clicking a control in thin client mode immediately generates a **mouseMove** event call to the application server (when that control has logic defined for that event). The **mouseMove** time processing then starts with the next **mouseMove** event that is received.

For details about getting and setting the handling of the **mouseMove** event to optimize performance on presentation clients in JADE thin client mode, see the **Application** class **getMouseMoveTime** and **setMouseMoveTime** methods.

Tip Consider using the **mouseHover** event to reduce the number of event calls to the application server.

mouseUp

Signature

```
mouseUp(button:  Integer;               (Form)
        shift:   Integer;
        x:       Real;
        y:       Real);

mouseUp(control: control-type input;    (Control)
        button:  Integer;
        shift:   Integer;
        x:       Real;
        y:       Real);
```

The **mouseUp** event occurs when the user releases a mouse button.

The **mouseUp** event uses the parameters listed in the following table.

Parameter	Description
button	The button that was released to cause the event. ■ 1 (left button) ■ 2 (right button) ■ 3 (middle button)
shift	The state of the Shift, Ctrl, and Alt keys when the mouse button was released. This integer is a combination of values, as follows. ■ To test if the Shift value (1) is set, use **if shift.bitAnd(1) <> 0 then** ■ To test if the Ctrl value (2) is set, use **if shift.bitAnd(2) <> 0 then**

Parameter	Description
	■ To test if the Alt value (4) is set, use **if shift.bitAnd(4) <> 0 then**
x	The current left physical position of the mouse pointer in the window. This is in the units of the **scaleMode** property of the form or control.
y	The current top physical position of the mouse pointer in the window. This is in the units of the **scaleMode** property of the form or control.

For details about converting the horizontal (**x**) and vertical (**y**) physical pixel positions to logical positions within the window that has been scrolled, see the **scrollHorzPos** property.

Use a **mouseUp** event to specify actions to occur when a specified mouse button is released. Unlike the **click** or **dblClick** event, the **mouseUp** event enables you to distinguish between the left, right, and middle mouse buttons. (For details about the context menu event that occurs after the right **mouseUp** event, see the **contextMenu** event method.)

If a called window does not implement the **contextMenu** event, the default processing continues. Drag mode is terminated (that is, the **dragDrop** event is received) by any **mouseUp** event (that is, when the left, right, or middle mouse button is released).

You can also write logic for mouse and keyboard combinations that use the Shift, Ctrl, and ALT keyboard modifiers.

See the **Window** class for details of the **MouseButton_** constants provided for window classes.

The following applies to both **click** and **dblClick** events.

- If a mouse button is pressed while the cursor is over a form or control, that object "captures" the mouse and receives all mouse events up to and including the last **mouseUp** event. This implies that the **x** and **y** mouse-pointer coordinates specified by a mouse event may not always be in the client area of the object that receives them.
- If mouse buttons are pressed in succession, the object that captures the mouse after the first press receives all mouse events until all buttons are released.

Note You can use a **mouseMove** event to respond to an event caused by moving the mouse or the **mouseLeave** event to respond to the user moving the mouse off the control. The **button** parameter for the **mouseUp** event differs from the **button** parameter used for the **mouseMove** event.

For the **mouseUp** event, the **button** parameter indicates exactly one button for each event; for the **mouseMove** event, it indicates the current state of all buttons.

notifyMedia

Signature `notifyMedia(cntrl: MultiMedia input);`

The **notifyMedia** event of the **MultiMedia** class is called whenever there is a change in the medium assigned to the **MultiMedia** control.

The **mediaName** property contains the name of the new medium.

Use the **sendString("capability device type")** method call to obtain the device type name.

notifyMode

Signature
```
notifyMode(cntrl: MultiMedia input;
           nmode: Integer);
```

The **notifyMode** event of the **MultiMedia** class is called whenever there is a change in the status of the **MultiMedia** control.

The value passed in the **nmode** parameter indicates the new status of the device, as shown in the following table.

MultiMedia Class Constant	Mode	Description
Mode_Not_Ready	1	Not ready
Mode_Open	7	Door open
Mode_Paused	6	Paused
Mode_Playing	3	Playing
Mode_Recording	4	Recording
Mode_Seeking	5	Seeking
Mode_Stopped	2	Stopped

notifyPosition

Signature
```
notifyPosition(cntrl: MultiMedia input;
               pos:   Integer);
```

The **notifyPosition** event of the **MultiMedia** class is called whenever there is a change in the position of the content of the **MultiMedia** control.

The value passed in the **pos** parameter indicates the position in the content of the device, in the units of the **timeFormat** property. The time between the arrival of **notifyPosition** events is controlled by the **timerPeriod** property.

The code fragment in the following example shows the use of the **notifyPosition** event method.

```
cd.notifyPosition(cd, cd.position);
```

openup

Signature
```
openup(table: Table input);
```

A **Table** control can be folded up so that it has only the height needed to display the current row when it loses focus. The **openup** event is generated when the table opens to its full size.

This facility is controlled by the **dropDown** property. If the value of this property is **DropDown_Auto** (**2**), the table opens up when the table receives the focus. If the value of the property is **DropDown_Click** (**1**), the user must specifically click the control to cause the event.

Use the **openup** event to make any necessary changes to the table and what is displayed in its full display state.

paint

Signature

```
paint();                               (Form)

paint(control: control-type input);    (Control)
```

The **paint** event occurs when:

- Part or all of a form or control is exposed after it has been moved or enlarged
- A window that was covering the object has been moved
- A visual aspect of the window has changed

Use a **paint** event if you need to coordinate the painting of a form or control with some external activity. The **paint** event is also invoked when the **refresh** and **refreshNow** methods are used. Using a **refresh** method in a **resize** event forces repainting of the entire object every time a user resizes the form.

Notes The **paint** event is not called if the form or control has the **autoRedraw** property set to **true** or when running a Web-enabled JADE application, as no JADE forms are created and displayed.

Java generates a paint request for the smallest rectangle that encloses the required paint areas. GUI changes in a **paint** event therefore cause continuous painting if the resulting paint rectangle includes the same window again.

Using a **paint** event for certain tasks can cause a cascading event. In general, avoid using a **paint** event for the following.

- Moving or sizing a form or control
- Changing any variables that affect size or visual appearance, such as setting the **backColor** property for an object
- Invoking a **refresh** or **refreshNow** method
- Invoking the **clearGraphics** method

A **resize** event may be more appropriate for some of these tasks.

The **paint** event is called after the normal painting of the form or control has completed, allowing this image to be drawn on.

It is not possible to handle the painting of transparent controls in the correct **zOrder** when it involves a mixture of controls that can be directly painted by JADE and those that can only be painted separately by Windows.

As a result, transparent sibling controls are always painted before any **JadeRichText**, **MultiMedia**, **JadeXamlControl**, **Ocx**, **OleControl**, **JadeDotNetVisualComponent**, or **ActiveXControl** controls, regardless of their **zOrder** settings.

pictureClick

Signature

```
pictureClick([control:    control-type input];
              picIndex:    Integer;
              whatClicked: Integer);
```

The **pictureClick** event is generated in a **ListBox** control or a **ComboBox** control when the picture area before the text of an item is clicked with the mouse. (See also the **itemPicture**, **hasPictures**, **hasTreeLines**, and **hasPlusMinus** properties.)

If the area clicked is before the pictures or the start of the treeline for the item, no events other than the **mouseDown** and **mouseUp** events are generated.

This event is also generated by pressing the Enter key when an entry in the list is selected.

Clicking the picture area does not expand or collapse an item.

The **pictureClick** event passes the index of the item being clicked and the picture that was clicked, as listed in the following table.

ComboBox or ListBox Class Constant	Index	Picture
PictureClick_PlusMinus	1	Plus or minus picture, controlled by the **hasPlusMinus** property
PictureClick_TreeLine	2	Tree line of item, controlled by the **hasTreeLines** property
PictureClick_Picture	3	Picture, controlled by the **hasPictures** property
PictureClick_ItemPicture	4	ItemPicture, set by the **itemPicture** property
PictureClick_KeyBoard	5	Keyboard action

Note If both the **hasPlusMinus** and **hasTreeLines** properties are set, a value of **1** is returned when the plus or minus picture is clicked, as this overlays the treeline area.

The picture image that is displayed depends on whether the item has subitems and whether the item is expanded or collapsed. See also the **pictureClosed**, **pictureOpen**, and **pictureLeaf** properties.

Clicking the picture does not select that entry.

pictureDblClick

Signature

```
pictureDblClick([control:    control-type input];
                picIndex:    Integer;
                whatClicked: Integer);
```

The **pictureDblClick** event is generated in a **ListBox** control or a **ComboBox** control when the picture area before the text of an item is double-clicked with the mouse. (See also the **itemPicture**, **hasPictures**, **hasTreeLines**, and **hasPlusMinus** properties.)

If the area clicked is before the pictures or the start of the treeline for the item in a list box or a combo box, no events other than the **mouseDown** and **mouseUp** events are generated. Clicking this area does not expand or collapse an item.

The **pictureDblClick** event passes the index of the item being clicked and the picture that was double-clicked, as listed in the following table.

ComboBox or ListBox Class Constant	Index	Picture
PictureClick_ItemPicture	4	ItemPicture, set by the **itemPicture** property
PictureClick_KeyBoard	5	Keyboard action
PictureClick_Picture	3	Picture, controlled by the **hasPictures** property

ComboBox or ListBox Class Constant	Index	Picture
PictureClick_PlusMinus	1	Plus or minus picture, controlled by the **hasPlusMinus** property
PictureClick_TreeLine	2	Tree line of item, controlled by the **hasTreeLines** property

Note If both the **hasPlusMinus** and **hasTreeLines** properties are set, a value of **1** is returned when the plus or minus picture is double-clicked, as this overlays the treeline area.

The picture images that are displayed depend on whether the item has subitems and whether the item is expanded or collapsed. See also the **pictureClosed**, **pictureOpen**, and **pictureLeaf** properties. Clicking or double-clicking the picture does not select that entry.

protected

Signature
```
protected(textbox:     JadeRichText;
          start:       Integer;
          length:      Integer;
          allowChange: Boolean io);
```

The **protected** event is generated in a **JadeRichText** control when a user attempts to alter in any way (for example, to insert, delete, or change the size of the font) text that is marked as protected.

Your logic can determine if changes to the rich text control should be allowed. Set the value of the **allowChange** parameter to **true** to allow the changes or to **false** if changes cannot be made to text in the control. The values of the **start** and **length** parameters indicate the block of text that the user is attempting to change.

If you do not implement this event on a specific control, any attempt to alter protected text is ignored.

In JADE, you can set or clear text protection only by calling the **setTextProtection** method. Protected text in rich text format that was created externally and loaded into a **JadeRichText** control remains protected.

The **protected** event is not raised when you call the **setTextProtection** method. In addition, it is not raised when you change unprotected text.

In the following example of the use of the **protected** event method, the user is restricted to changing a single character of protected text at a time.

```
jadeRichText_protected(textbox: JadeRichText input; start: Integer;
       length: Integer; allowChange: Boolean io) updating;
begin
    allowChange := length < 2;
end;
```

queryColumnMove

Signature
```
queryColumnMove(table: Table;
                c:     Integer;
                cto:   Integer): Boolean;
```

The **queryColumnMove** event of the **Table** control is called when the user releases the mouse after dragging a fixed column to a new position when the **allowDrag** property is set to **true**.

The parameter values are listed in the following table.

Parameter	Description
table	The table in which the column is to be moved
c	The column that is to be moved
cto	The column before which the dragged column is inserted

The column is moved to the position to which it was dragged when this event method is not defined or if this event method is defined and it returns a value of **true**.

The move of the column is aborted when this event method is defined and it returns **false**.

The method in the following example shows the use of the **queryColumnMove** event method.

```
queryColumnMove(table: Table; c: Integer; cto: Integer): Boolean;
begin
    if c <= 2 or cto <= 2 then
        return false;         // don't let them move the first two columns
    endif;
    return true;
end;
```

queryDock

Signature

```
queryDock(dockbar: control-name input;     (JadeDockBar)
          newWin:  Window;
          dockpos: Integer): Boolean;

queryDock(dockcont: control-name input;    (JadeDockContainer)
          newWin:   Window;
          dockpos:  Integer): Boolean;
```

The **queryDock** event of the **JadeDockBar** or **JadeDockContainer** class is called when the user drags a docking control into a valid docking position that is accepted by the **JadeDockContainer** class **allowDocking** property of the window.

If the docking is permitted, this **queryDock** event method must return **true**. If docking is not permitted, it must return **false**.

Use this method to control the container into which the control can be docked and how it can be docked.

Note The method is called once only for each combination of the **newWin** and **dockpos** parameters for each dragging episode.

For details about floating and docking container controls, see "Floating a Docking Control" and "Docking a Control", under the "JadeDockBase Class", earlier in this document. See also the docking examples under the **JadeDockContainer** class **allowDocking** property, earlier in this document.

queryRowColChg

Signature
```
queryRowColChg(table:    Table input;
               newSheet: Integer;
               newRow:   Integer;
               newCol:   Integer): Boolean;
```

The **queryRowColChg** event occurs for the **Table** control when the user selects a cell that is not the current cell or a new sheet of the displayed table by using the mouse or the keyboard. Use this event method to prevent the user from moving the current cell until valid data has been entered.

If this event method is not implemented or it is implemented and returns **true**, the sheet, row, or column is changed to the selected table element and the **rowColumnChg** or the **sheetChg** event is called.

If this event is implemented and it returns **false**, the requested user action is ignored; that is, the current **sheet**, **row**, or **column** is not changed and the **rowColumnChg** or **sheetChg** event is not sent.

This event method is not called as a result of the table losing focus or when the **sheet**, **row**, or **column** is changed by logic.

When a collection is attached to a table and the current row is scrolled so that it will be discarded (outside of the displayed virtual window over the collection), the **queryRowColChg** and **rowColumnChg** event methods are called. If the **queryRowColChg** event is rejected, the scroll action is discarded. (Note that the **queryRowColChg** event passes the current **row** and **column** values as parameters, because the actual current row number remains unchanged after scrolling.)

queryRowMove

Signature
```
queryRowMove(table: Table;
             r:     Integer;
             rto:   Integer): Boolean;
```

The **queryRowMove** event of the **Table** control occurs when the user releases the mouse after dragging a fixed row to a new position when the **allowDrag** property is set to **true**.

The parameter values are listed in the following table.

Parameter	Description
table	The table in which the row is to be moved
r	The row that is to be moved
rto	The row before which the dragged row is inserted

The row is moved to the position to which it was dragged when this event method is not defined or if this event method is defined and it returns a value of **true**.

The move of the row is aborted when this event method is defined and it returns **false**.

The method in the following example shows the use of the **queryRowMove** event method.

```
queryRowMove(table: Table; r: Integer; rto: Integer): Boolean;
begin
    if r <= 2 or rto <= 2 then
        return false;         // don’t let them move the first two rows
    endif;
```

```
        return true;
    end;
```

querySheetChg

Signature
```
querySheetChg(folder:   Folder;
              newSheet: Sheet): Boolean;
```

The **querySheetChg** event of the **Folder** class occurs before the user changes a sheet using the keyboard or the mouse, to give the logic the opportunity to reject the change of sheet.

The actions determined by the return values for this event are listed in the following table.

Return Value	Action
True	The **sheetChg** event is called
False	The sheet change request is ignored and the **sheetChg** event is not called

Use the **querySheetChg** event in wizards, to enable users to change sheets back again (for example, by using a **<Back** button).

queryUnload

Signature
```
queryUnload(cancel: Integer io;
            reason: Integer);
```

The **queryUnload** event occurs before a form closes, to give the logic the opportunity to reject the closure. Logic dealing with the closure should be placed in the **unload** event. When the MDI frame is closed, this event is sent to all child forms as well as the MDI frame. This event is not called if a form is unloaded before it was displayed and the **load** event has not been executed, or if the **delete** method was used to delete the running instance of the form.

When the JADE language **terminate** instruction is executed, all active forms have the **queryUnload** event sent to them.

If a **queryUnload** method is aborted, **queryUnload** is not called for any other forms affected by the unload and all affected forms are unloaded without calling the **unload** method. If an **unload** method is aborted, all other affected forms are unloaded without calling the **unload** method.

If any form involved in the close process rejects the request to unload, no further **queryUnload** events are sent and no form unloads are performed. When all forms involved in the closure agree to close (by the **queryUnload** events), the **unload** event is then sent to those forms.

Note If the form to which the event is sent is the active form, a **lostFocus** event is caused on the active control before the **queryUnload** event is sent. If the **queryUnload** event is rejected, focus is returned to that same control, unless the focus has been moved elsewhere in the processing of that event.

When the last MDI child form in the default MDI frame closes (supplied by the **jade.exe** program), the default MDI frame also closes. When the last form closes, the application closes.

The parameters of the **queryUnload** event are listed in the following table.

Parameter	Description
cancel	Setting this parameter to any value other than **0** (zero) stops the **queryUnload** event in all loaded forms and stops the form and application from closing. This parameter is ignored if the form database object was deleted.
reason	Indicates the reason for the **queryUnload** event.

The **reason** parameter can have the values listed in the following table.

Form Class Constant	Value	Description
QueryUnload_MdiChild	4	An MDI child form is closing because the MDI form is closing.
QueryUnload_TaskManager	3	The Microsoft Windows Task Manager is closing the application.
QueryUnload_UnloadMethod	1	The **unload** event has been invoked from logic.
QueryUnload_User	0	The user has selected the **Close** command from the Control-Menu icon on the form.
QueryUnload_Windows	2	The current Windows-environment session is ending.

Use the **queryUnload** event to ensure that there are no unfinished tasks in each form before an application closes. For example, if a user has not yet saved some new data in any form, your application can ask the user if the data should be saved.

resize

Signature `resize();`

The **resize** event of the **Form** class occurs when the size of a form is changed. This event is called under the following circumstances.

- When a form is loaded and displayed and its size changes because of environmental differences
- When a form is minimized, maximized, or restored by a user or through logic
- When a top-level menu item is added or removed through logic, causing a menu line to be added or removed (or changed from a single menu line to a double menu line)
- When a skin is applied or removed

Note The **resize** event generated by logic changes is called after the current logic event is completed.

This event is *not* called if the form is resized by logic in one of the following ways.

- Calling the **move** method
- Changing the **height**, **width**, **clientHeight**, or **clientWidth** property of the form
- Changing the **borderStyle** property of the form
- Adding or removing a scroll bar to the form

Use a **resize** event to move or resize controls when the parent form is resized or to recalculate variables or properties that may depend on the size of the form.

resizeColumn

Signature
```
resizeColumn(table:         Table input;
             column:        Integer input;
             previousWidth: Integer input);
```

The **resizeColumn** event of the **Table** class occurs after the user has resized a column on a **Table** control using the mouse.

The parameters for the **resizeColumn** event method are listed in the following table.

Parameter	Description
table	Table control whose column was resized
column	Index of the column that has been resized
previousWidth	Width of the column before the resize operation took place

resizeRow

Signature
```
resizeRow(table:          Table input;
          row:            Integer input;
          previousHeight: Integer input);
```

The **resizeRow** event method of the **Table** class occurs after the user has resized a row on a **Table** control using the mouse.

The parameters for the **resizeRow** event method are listed in the following table.

Parameter	Description
table	Table control whose row was resized
row	Index of the row that has been resized
previousHeight	Width of the row before the resize operation took place

rowColumnChg

Signature
```
rowColumnChg(table: Table input);
```

The **rowColumnChg** event occurs for the **Table** control when the user selects a different cell of the displayed table by either clicking a cell or by using the arrow keys to select a new cell.

The selected cell is reflected in the **row** and **column** properties.

When a collection is attached to a table and the current row is scrolled so that it will be discarded (outside of the displayed virtual window over the collection), the **queryRowColChg** and **rowColumnChg** event methods are called. If the **queryRowColChg** event is rejected, the scroll action is discarded. (Note that the **queryRowColChg** event passes the current **row** and **column** values as parameters, because the actual current row number remains unchanged after scrolling.)

The **sheet** is contained in the **topSheet** property.

scrolled

Signature
```
scrolled([control:  control-type input;]
          scrollBar: Integer);
```

The **scrolled** event occurs when the user scrolls a **Form** or a **BaseControl**, **ComboBox**, **ListBox**, **Picture**, **Table**, or **TextBox** control. For **ScrollBar** controls, this event occurs as the control is scrolled. When the user releases the mouse button, a **change** event is sent.

The **scrolled** event passes the scroll bar that is being scrolled, as follows.

- **ScrollBar_Horizontal** (1)
- **ScrollBar_Vertical** (2)

See "Window Class Constants", for the **ScrollBar_** constants that are provided.

The **scrolled** event gives logic the opportunity to coordinate other actions with the scrolling process. Use this event to perform calculations or to manipulate controls that must be coordinated with ongoing changes in scroll bars. Alternatively, use the **change** event when you want an update to occur only once, after a scroll bar position change is complete.

Use the **scrollHorzPos** and **scrollVertPos** properties to determine the scroll position. Use the **topIndex** property to control the scrolling position set for the list box control. Use of the **scrollHorzPos** and **scrollVertPos** methods do not generate a scrolled event when a new scroll position is set.

For details about scrolling in the row of a **Table** control to which a collection is attached, see the **queryRowColChg** and **rowColumnChg** event methods.

selChanged

Signature
```
selChanged(textbox:   JadeRichText input;
           selStart:  Integer;
           selLength: Integer);
```

The **selChanged** event of the **JadeRichText** class occurs when the selection of text within the control has changed, updating the **selStart** and **selLength** parameters with the starting position and the length (in characters) of the selection. (See also the **selStart** and **selLength** properties.)

A **selChanged** event can synchronize or coordinate the display of data between controls.

sheetChg

Signature
```
sheetChg(control: control-type input);
```

The **sheetChg** event occurs when the user clicks on the tab of a sheet that is not currently the top sheet of a **Folder** or **Table** control, to enable that sheet to become visible. (The **click**, **mouseLeave**, and the **sheetChg** events are generated when the user clicks the tab of a sheet that is not currently the top sheet or the Ctrl+Page Up or Ctrl+Page Down shortcut keys are used to move the focus of a folder to the prior or next enabled visible sheet when that folder or a child of the folder has focus.)

For a **Folder** control, see also the **isMySheetVisible** and **showMySheet** properties.

For a **Table** control, the sheet that is clicked is reflected in the **topSheet** property.

sysNotify

Signature

```
sysNotify(eventType: Integer;                   (Form)
          theObject: Object;
          eventTag:  Integer);

sysNotify(control:   control-type input;    (Control)
          eventType: Integer;
          theObject: Object;
          eventTag:  Integer);
```

A **sysNotify** event occurs when it is triggered when a specified event occurs on a JADE system object; for example, a customer object is updated. If a form or control has an attached window, a requested user notification is directed to the **userNotify** event and a requested system notification to the **sysNotify** event.

The **sysNotify** event of a window is invoked when a window notification is detected. Insert code into the **sysNotify** event to respond to or deal with the notification, so that any required tasks are performed when the event occurs.

The following examples show the use of the **sysNotify** event.

```
sysNotify(eventType: Integer; theObject: Object;
          eventTag:  Integer) updating;
    // Update the instance collection as instances of Cat are
    // created and deleted by other users.
    if Cat.hasInstance(theObject) then
        if eventType = Object_Create_Event then
            // This may be a notification from one we have just added!
            if not instanceCollection.includes(theObject) then
                instanceCollection.add(theObject);
            endif;
        elseif eventType = Object_Delete_Event then
            // This may be a notification from one we have just deleted!
            if instanceCollection.includes(theObject) then
                // If the instance on display is about to be deleted, tell
                // the user what is about to happen.  They cannot have
                // changed it yet, as it would have been locked if they had)
                if currentInstance = theObject then
                    app.msgBox('Another process has deleted the instance
                               displayed.' & CrLf & 'This instance will now
                               be removed.', self.name, MsgBox_OK_Only +
                               MsgBox_Information_Icon);
                endif;
                instanceCollection.remove(theObject);
                clearFormFields;
            endif;
        endif;
        showInstanceNumber;
    endif;
end;

sysNotify(eventType: Integer; notifyObject: Object;
          eventTag:  Integer) updating;
begin
    if eventType = Object_Create_Event then       // new fault
        createGraph;
```

```
            loadTable;
        endif;
    end;
```

The **sysNotify** event parameters, described in the following subsections, are listed in the following table.

Parameter	Contains …
eventType	The type of event received
theObject	The object for which the notification is to be received
eventTag	An integer value that is received for each notification

eventType

The **eventType** parameter of the **sysNotify** event contains the type of event received. The global constants for the types of system event that can be received are listed in the following table.

Global Constant	Value	Object has been ...
Object_Create_Event	4	Created
Object_Delete_Event	6	Deleted
Object_Update_Event	3	Updated

theObject

The **theObject** parameter of the **sysNotify** event contains the object for which the notification is to be received.

Caution Attempts to access features (that is, methods and properties) for the object of a notification of a delete event type (**Object_Delete_Event**) raise an exception.

eventTag

The **eventTag** parameter of the **sysNotify** event contains an integer value that is received with each notification.

trayIconClicked

Signature `trayIconClicked(clickType: Integer) updating;`

The **trayIconClicked** event of the **Form** class occurs when the user clicks a system tray entry created by the **Form** class **setSystemTrayEntry** method.

The **Form** class provides the class constants listed in the following table to indicate the types of clicks that occur on the system tray icon.

Form Class Constant	Value	Description
TrayIcon_LeftClick	1	Left mouse click
TrayIcon_RightClick	2	Right mouse click
TrayIcon_LeftDblClick	3	Left mouse double-click
TrayIcon_RightDblClick	4	Right mouse double-click

unload

Signature `unload();`

The **unload** event of the **Form** class occurs when a form is about to be removed from the screen. This event is triggered by a user action (closing the form using the Control-Menu icon) or by an **unloadForm** method.

The **unload** event and **queryUnload** events are not called if the form is unloaded before being displayed, the **load** event was never called, or if the **delete** method was used to delete the running instance of the form.

The **unload** event is always preceded by the **queryUnload** event, giving the opportunity to reject the requested closure. Use an **unload** event to specify actions to take place when the form is unloaded.

If a **queryUnload** method is aborted, **queryUnload** is not called for any other forms affected by the unload and all affected forms are unloaded without calling the **unload** event method. If an **unload** method is aborted, all other affected forms are unloaded without calling the **unload** method.

Unloading a form queues the window for deletion but if another window is deleted before the next idle point, previously queued deleted windows are re-evaluated for deletion. If the queued window or its children have no outstanding Windows message, there are no incomplete event methods for that window or its children, and the method that created the window has exited or the window was deleted, the physical window is deleted.

The **unload** event can be caused by using the **unloadForm** method or by the user taking an action; for example, selecting the **Close** command on the Control-Menu of a form, exiting from the application by using the **End Task** button on the Windows Task Manager, closing the MDI frame form for which the current form is a child form, or exiting the Windows environment while the application is running.

If the form is the last child of the default MDI frame (**jade.exe**-supplied), the MDI frame form is also closed. If the form is the last form for the application, the application is closed.

updated

Signature `updated(ole: OleControl input) updating;`

The **updated** event of the **OleControl** class occurs when an OLE object is updated in the JADE environment.

userNotify

Signature

```
userNotify(eventType: Integer;                 (Form)
           eventType: Integer;
           theObject: Object;
           eventTag:  Integer;
           userInfo:  Any);

userNotify(control:   control-type input;    (Control)
           eventType: Integer;
           theObject: Object;
           eventTag:  Integer;
           userInfo:  Any);
```

A **userNotify** event occurs when this event is triggered from the JADE Object Manager by a user call. If a form or control has an attached window, a requested user notification is directed to the **userNotify** event and a requested system notification to the **sysNotify** event.

The **userNotify** event of a window is invoked when a window notification is detected. Insert code into the **userNotify** event to respond to or deal with the notification, so that any required tasks are performed when the event occurs. (The **Object** class **causeEvent**, **sdeCauseEvent**, or **sdsCauseEvent** method informs the JADE Object Manager when the event coded in your method occurs.)

The method in the following example shows the use of the **userNotify** event.

```
userNotify(eventType: Integer; theObject: Object; eventTag:
           Integer; userInfo: Any) updating;
vars
begin
    if eventType = 17 then
        createGraph;
        loadTable;
    endif;
end;
```

The **userNotify** event parameters, described in the following subsections, are listed in the following table.

Parameter	**Contains ...**
eventType	The type of event received
theObject	The class for which the notification is to be invoked
eventTag	An integer value that is required for each notification
userInfo	A value of **Any** primitive type that is received when the event is notified

eventType

The **eventType** parameter of the **userNotify** event contains the type of event received. The global constants for the types of user event that can be received are listed in the following table.

Global Constant	**Value**
User_Base_Event	16
User_Max_Event	Max_Integer (#7FFFFFFF, equates to 2147483647)

The **JadeTextEdit** class event type can be one of the **JadeTextEdit** class constant values listed in the following table.

Constant	**Description**	**Event Tag**
EVENTTYPE_BOUNDKEY	User pressed a key that was bound to a notification.	Specified in the **eventTag** parameter of the **bindKeyToNotification** method
EVENTTYPE_ALTERREADONLY	An attempt was made to alter the document when **readOnly** is **true**	Zero (not applicable)
EVENTTYPE_CANCEL	One of the following: ■ Esc key was pressed ■ Control lost focus ■ Button pressed	Zero (not applicable)

Constant	Description	Event Tag
	■ Control deleted	
EVENTTYPE_CLIPBUFFCHG	The clip buffer has changed	Buffer number (**0** through **9**)
EVENTTYPE_CLIPBUFFETC	A clip buffer paste or copy command was not followed by a buffer number keystroke in the range **0** through **9**	Key code entered
EVENTTYPE_SELECTIONSTATE	Selection has changed between empty and not empty	Current state, with zero (**0**) being empty

theObject

The **theObject** parameter of the **userNotify** event contains the object for which the notification is to be received.

The value of this parameter for pre-defined pseudo-notifications in the **JadeTextEdit** class is always null.

eventTag

The **eventTag** parameter of the **userNotify** event contains an integer value (for example, an index into an array) that is received with each notification. This parameter corresponds to the **eventTag** parameter in the **Object** class **beginNotification** method that subscribes to the user event.

userInfo

The **userInfo** parameter of the **userNotify** event is a value of **Any** primitive type (that is, a string, integer, or character) that is received when the event is notified. This parameter corresponds to the **userInfo** parameter in the **Object** class **causeEvent**, **sdeCauseEvent**, or **sdsCauseEvent** method that triggered the notification.

userResize

Signature `userResize(control: control-type input);`

The **userResize** event occurs when the user drags the resize bar of the **JadeDockBar** or **JadeDockContainer** control to a new position or resizes a floating **JadeDockBar** or **JadeDockContainer** control.

If you want to differentiate between the **userResize** event being called when the user drags the controls resize bar and when the user has resized the floating window, call the **isFloating** method.

For details about adding resize bars to docking controls that are all aligned horizontally or vertically, see the **showResizeBar** property.

validate

Signature `validate(editMask: JadeEditMask): Boolean;`

The **validate** event method of the **JadeEditMask** class occurs when the user attempts to shift the focus to another control or form within the same application by using the mouse or keyboard. Your logic can validate the text field to determine what action to take when the data is incomplete.

This event is not called if the user switches to another displayed form of the application, to another application, or closes the form. In addition, the event is not called if logic shifts the focus.

This event returns **true** to allow focus to be shifted or it returns **false** to force the focus to remain on the control. If you do not implement this event, focus movement is permitted (that is, implement the **validate** event if you want the focus to remain on the control).

windowCreated

Signature

```
windowCreated(cntrl:        Control input;
              persistCntrl: Control);
```

A **windowCreated** event of the **Control** class is called for all controls when the window for the control is created. This event is called before the **load** event for the **Form** class, so that you can initialize a control (for example, creating a child window) when the window for that control is present.

The **cntrl** parameter passes the event the control itself, and the **persistCntrl** parameter passes the control object from which the window was created (usually the persistent control object). This enables the event logic to clone any reference information that is also required by the control. (The runtime control is created by a shallow clone, and the primitive properties only are copied.)

Notes Properties retrieved from the control object specified in the **persistCntrl** parameter reflect the persistent, not the transient, instance.

Avoid referring to user-defined properties or methods on the **Application** or **Global** classes in the **windowCreated** method or any methods called by the **windowCreated** method. When the form is displayed in the JADE Painter, the application used is not the user application, so these properties or methods are not available at run time. If the property or method must be referenced, the method must perform a runtime check to ensure that the application is the expected one; otherwise, an exception will be raised.

Graphics Properties and Methods

Graphical methods provide the ability to draw on forms and controls. The type of drawing includes points, lines, arcs, chords, pies, ellipses (and circles), and rectangles. As graphical methods and properties are defined in the **Window** class, they are available for any **Form** or **Control** class or subclass instance.

Note When drawing on a **Form** class or a **BaseControl** or **Picture** control, the drawing positions are relative to the scrolled position. The position for drawing on **TextBox**, **Table**, and **ListBox** controls is not affected by scrolling.

You can access all GUI properties and methods (which are marked as **clientExecution** methods) from a server method except for anything that brings up a modal type dialog (that is, the common dialog class methods, the **app.msgBox**, and the **showModal** and **popupMenu** methods in the **Form** class). The other exceptions to this are the **app.doWindowEvents**, **app.checkPictureFile**, and **app.loadPicture** methods, which are executed relative to the server.

Caution Use of GUI methods and properties is *very* expensive in a server method. A **clientExecution** method requires that all transient objects passed to the server are passed back with the client execution (and passed back to the server after the client execution is complete). Therefore accessing GUI properties and methods within a server execution should be done only in exceptional circumstances.

When you use the **drawFilledRectangle** or **drawSolidRectangle** method to draw a solid rectangle, previous figures that were drawn in the history but are covered and all property settings that have been made redundant are removed. In addition, the **drawLine** method removes any previous lines with the same co-ordinates where the same **width** and **drawStyle** property values apply.

The automatic process does not occur if you use flood-fill, after logic calls to the **drawSize** method, or if the value of the **drawMode** property is not **DrawMode_Copy** (13).

For details about the graphics properties and methods, see "Graphics Properties" and "Graphics Methods", in the following sections.

The form or control properties summarized in the following table also affect the graphical process.

Property	Description
backColor	Color property of a form or control causes the graphical image to be erased. If the **autoRedraw** property is set to **true**, the image is repainted using the new value of the **backColor** property.
clipControls	If the **clipControls** property of the **Form** class is set to **true**, controls are excluded from the painted area of the form or parent control, meaning that the graphics functions are not able to draw over child controls.
scaleHeight	Height of the drawing area expressed in the units of the **scaleMode** property.
scaleLeft	Horizontal coordinate of the left edge of the drawing area expressed in the units of the **scaleMode** property.
scaleTop	Vertical coordinate of the top edge of the drawing area expressed in the units of the **scaleMode** property.
scaleWidth	Width of the drawing area expressed in the units of the **scaleMode** property.
scaleMode	Units used to express the dimensions of the drawing area of the form or control (defaults to pixels).

Graphics Properties

The graphical properties defined in the **Window** class are summarized in the following table.

Property	Description
autoRedraw	Setting the **autoRedraw** property to **true** automatically redraws graphical output in a form or picture box when the object is resized or redisplayed after being hidden by another window, for example. Setting the value to **false** means that any graphical output is lost. However, you can place logic in the **paint** event of the form or control, which redraws the images.
drawFillColor	Determines the color of the fill.
drawFillStyle	Determines the style of the fill.
drawFontBold	Used when constructing the font used for drawing text.
drawFontItalic	Used when constructing the font used for drawing text.
drawFontName	Used when constructing the font used for drawing text.
drawFontSize	Used when constructing the font used for drawing text.
drawFontStrikethru	Used when constructing the font used for drawing text.
drawFontUnderline	Used when constructing the font used for drawing text.
drawMode	Determines the appearance and interaction with the existing graphic image while drawing.
drawStyle	Determines the line style used for drawing.
drawTextAlign	Determines the alignment of any text drawn.
drawTextCharRotation	Specifies the angle in degrees between each character base line and the *x* axis of the form or control
drawTextRotation	Specifies the angle in degrees between the base line of the text output and the *x* axis of the form or control.
drawWidth	Determines the width of the lines drawn.
drawWindow	Determines what part of the window is drawn on.

autoRedraw

Type: Boolean

Availability: Read or write at run time only

The **autoRedraw** property of the **Window** class applies to any form or control at run time only and specifies whether the drawing methods save a history of graphics commands issued for that form or control. This history of commands is replayed when the repainting of that window object is required, enabling the image to be automatically rebuilt. The **autoRedraw** property can be used to overcome the problem when graphics output on the object of a window is lost when that window is covered over and then uncovered.

To rebuild the image, you can:

- Place all calls to graphics methods in the **paint** event for the window
- Use the **autoRedraw** property

Using the **autoRedraw** property has the following advantages over placing the calls to graphics methods in the **paint** event.

- Calls to graphics events need not be in the **paint** event for the object.
- If the **scaleMode** property of the object is set to user scaling, resizing the form or control automatically redraws the image scaled according to the new size.
- Graphical methods function only after the control has been painted and is visible. Therefore, without the **autoRedraw** property, drawing in the **load** event of the form does not produce an image.
- Because a history is kept of the drawing and of any graphical attribute changes made while drawing, use of the **drawUndo** method enables the image to be "rolled back" to a previous point.
- If the drawn image is larger than the visible window, scrolling the window by using a scroll bar automatically repaints the image, showing the area of the image that is scrolled into view.

Caution Because use of the **autoRedraw** property saves a history, there is a cost to its use. Take care to ensure that the history is cleared at appropriate points, by using the **clearGraphics** method, otherwise the history list consumes an increasing amount of memory, and the graphical presentation gets slower and slower as it has to process an increasing amount of history when repainting.

When you use the **drawFilledRectangle** or **drawSolidRectangle** method to draw a solid rectangle, previous figures that were drawn in the history but are covered and all property settings that have been made redundant are removed. In addition, the **drawLine** method removes any previous lines with the same co-ordinates where the same **width** and **drawStyle** property values apply. The automatic process does not occur if you use flood-fill, after logic calls to the **drawSize** method, or if the value of the **drawMode** property is not **DrawMode_Copy** (**13**).

The settings of the **autoRedraw** property are listed in the following table.

Setting	Description
true	Enables automatic repainting of a form or control. Graphics are written to the screen, and the commands used to create the image are saved in a history "file" in memory. The **paint** event of the object is not called, and the object is repainted when necessary, by using the memory image.
false	Disables automatic repainting of an object, and writes graphics or print output to the screen only. The **paint** event of the object is invoked, allowing user logic to redraw the graphics (the default)

Changing the setting of the **autoRedraw** property from **true** to **false** automatically clears the graphics history but does not affect the image that is currently displayed.

Drawing properties are saved in the history only when they are changed if a drawing method has been called since the last call of the **clearGraphics** method. The history consists of:

- The initial value of all drawing properties.
- One entry for each drawing request and for each drawing property change.

If you set the **backColor** property, all graphics are erased. However, if the **autoRedraw** property is set, the image is repainted using the new **backColor**.

In general, all graphics instructions should normally be in a **paint** event unless the **autoRedraw** property is set to **true**.

See the **startDrawingCapture** and **stopDrawingCapture** methods, which cause the **autoRedraw** property to be set to **true** and **false**, respectively.

drawFillColor

Type: Integer

Availability: Read or write at run time only

The **drawFillColor** property of the **Window** class contains the color used to fill in shapes drawn with the graphics methods.

JADE uses the RGB scheme for colors. Each property can be set, by using the appropriate RGB value.

The valid range for a normal RGB color is **0** through **16,777,215** (#FFFFFF). The high byte of a number in this range equals zero (**0**); the lower three bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively.

The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is not zero (**0**), JADE uses the system colors, defined in the Control Panel of the user.

By default, the **drawFillColor** property is set to **0** (black).

When the **drawFillStyle** property is set to its default value of **DrawFillStyle_Transparent** (1), the setting of the **drawFillColor** property is ignored.

The code fragment in the following example shows the use of the **drawFillColor** property.

```
    table.drawFillColor := Green;
```

drawFillStyle

Type: Integer

Availability: Read or write at run time only

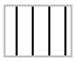
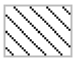
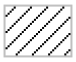
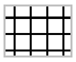
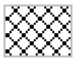
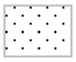
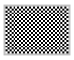
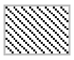
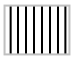
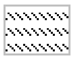
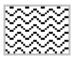
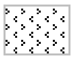
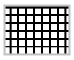
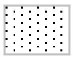
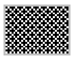
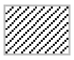
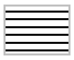
The **drawFillStyle** property of the **Window** class contains the pattern used to fill the shapes drawn with the graphics methods.

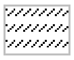
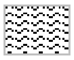
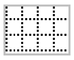
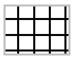
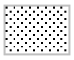
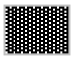
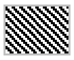
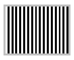
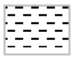
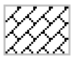
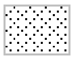
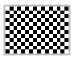
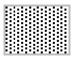
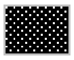
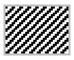
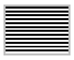
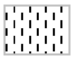
When the **drawFillStyle** property is set to **DrawFillStyle_Transparent** (1), the **drawFillColor** property is ignored.

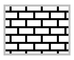
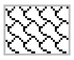
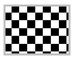
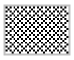
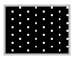
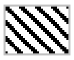
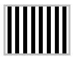
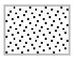
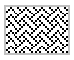
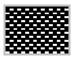
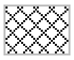
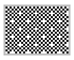
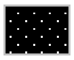
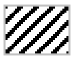
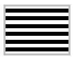
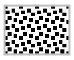
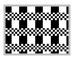
The **drawFillStyle** property values in the range **1** through **7** (the Microsoft standard fill styles) are drawn with a transparent background and the JADE-defined fill styles in the range 8 through 55 are drawn with a white background.

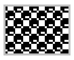
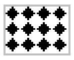
The settings of the **drawFillStyle** property are listed in the following table.

Window Class Constant	Integer Value	Image
DrawFillStyle_Solid (the default value)	0	■
DrawFillStyle_Transparent	1	□
DrawFillStyle_HorzLine	2	▤

Window Class Constant	Integer Value	Image
DrawFillStyle_VertLine	3	
DrawFillStyle_UpDiagonal	4	
DrawFillStyle_DownDiagonal	5	
DrawFillStyle_Cross	6	
DrawFillStyle_DiagonalCross	7	
DrawFillStyle_4DotDiamond99	8	
DrawFillStyle_EveryOther	9	
DrawFillStyle_UpDiagonal4	10	
DrawFillStyle_VertLine4	11	
DrawFillStyle_HalfUpDiagonal	12	
DrawFillStyle_HorzWaves4	13	
DrawFillStyle_Triangles	14	
DrawFillStyle_Cross55	15	
DrawFillStyle_4DotDiamond95	16	
DrawFillStyle_FilledDiamond	17	
DrawFillStyle_DownDiagonal4	18	
DrawFillStyle_HorzLine4	19	

Window Class Constant	Integer Value	Image
DrawFillStyle_HalfDownDiagonal	20	
DrawFillStyle_HorzWaves3	21	
DrawFillStyle_DottedCross	22	
DrawFillStyle_Cross99	23	
DrawFillStyle_4DotDiamond55	24	
DrawFillStyle_Checkered	25	
DrawFillStyle_DbleUpDiagonal	26	
DrawFillStyle_VertLine2	27	
DrawFillStyle_HorzDash	28	
DrawFillStyle_DownRectangle	29	
DrawFillStyle_8DotDiamond99	30	
DrawFillStyle_AltSquares2	31	
DrawFillStyle_4DotDiamond53	32	
DrawFillStyle_Rev4DotDiamond55	33	
DrawFillStyle_DbleDownDiag	34	
DrawFillStyle_HorzLine2	35	
DrawFillStyle_VertDash	36	

Window Class Constant	Integer Value	Image
DrawFillStyle_HorzRectangle	37	
DrawFillStyle_UpKeyShape	38	
DrawFillStyle_AltSquares4	39	
DrawFillStyle_8DotDiamond55	40	
DrawFillStyle_Rev4DotDiamond95	41	
DrawFillStyle_TripleDownDiag	42	
DrawFillStyle_DbleVertLine	43	
DrawFillStyle_Speckled	44	
DrawFillStyle_Interlocked	45	
DrawFillStyle_ReverseHorzDash	46	
DrawFillStyle_DiagonalHatch	47	
DrawFillStyle_InvertedCross	48	
DrawFillStyle_Rev4DotDiamond99	49	
DrawFillStyle_TripleUpDiagonal	50	
DrawFillStyle_DbleHorzLine	51	
DrawFillStyle_PatchworkSquares	52	
DrawFillStyle_Tartan	53	

Window Class Constant	Integer Value	Image
DrawFillStyle_ShinyBalls	54	
DrawFillStyle_Balls	55	

The code fragment in the following example shows the use of the **drawFillStyle** property.

```
    table.drawFillStyle := DrawFillStyle_DiagonalCross;
```

Note Printing the draw fill styles in the range **DrawFillStyle_4DotDiamond99** (8) through **DrawFillStyle_Balls** (55) using the Windows Enhanced Meta Files (EMF) format produces a solid gray hue only, because of an EMF problem that is outside JADE's control. To print these styles, you must use the Scalable Vector Graphics (SVG) format.

drawFontBold

Type: Boolean

Availability: Read or write at run time only

The **drawFontBold** property of the **Window** class, together with the **drawFontItalic**, **drawFontStrikethru**, **drawFontUnderline**, **drawFontName**, and **drawFontSize** properties, determines the font used for graphics text drawing methods.

The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

Note The **drawFonts** available in JADE vary according to your system configuration, display devices, and printing devices. Font-related properties can be set only to values for which actual **drawFonts** exist.

The font that is used is constructed only when a draw text method call is made, so the order of setting the property values does not matter as it does for controls.

drawFontItalic

Type: Boolean

Availability: Read or write at run time only

The **drawFontItalic** property of the **Window** class, together with the **drawFontBold**, **drawFontStrikethru**, **drawFontUnderline**, **drawFontName**, and **drawFontSize** properties, determines the font used for graphics text drawing methods.

The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

Note The **drawFonts** available in JADE vary according to your system configuration, display devices, and printing devices. Font-related properties can be set only to values for which actual **drawFonts** exist.

The font that is used is constructed only when a draw text method call is made, so the order of setting the property values does not matter as it does for controls.

drawFontName

Type: String

Availability: Read or write at run time only

The **drawFontName** property of the **Window** class, together with the **drawFontBold**, **drawFontItalic**, **drawFontStrikethru**, **drawFontUnderline**, and **drawFontSize** properties, determines the font used for graphics text drawing methods.

The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

Note The **drawFonts** available in JADE vary according to your system configuration, display devices, and printing devices. Font-related properties can be set only to values for which actual **drawFonts** exist.

The font that is used is constructed only when a draw text method call is made, so the order of setting the property values does not matter as it does for controls.

drawFontSize

Type: Real

Availability: Read or write at run time only

The **drawFontSize** property of the **Window** class, together with the **drawFontBold**, **drawFontItalic**, **drawFontStrikethru**, **drawFontUnderline**, and **drawFontName** properties, determines the font size used for graphics text drawing methods.

The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

Note The **drawFonts** available in JADE vary according to your system configuration, display devices, and printing devices. Font-related properties can be set only to values for which actual **drawFonts** exist.

The font that is used is constructed only when a draw text method call is made, so the order of setting the property values does not matter as it does for controls.

drawFontStrikethru

Type: Boolean

Availability: Read or write at run time only

The **drawFontStrikethru** property of the **Window** class, together with the **drawFontItalic**, **drawFontBold**, **drawFontUnderline**, **drawFontName**, and **drawFontSize** properties, determines the font used for graphics text drawing methods. The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

Note The **drawFonts** available in JADE vary according to your system configuration, display devices, and printing devices. Font-related properties can be set only to values for which actual **drawFonts** exist.

The font that is used is constructed only when a draw text method call is made, so the order of setting the property values does not matter as it does for controls.

drawFontUnderline

Type: Boolean

Availability: Read or write at run time only

The **drawFontUnderline** property of the **Window** class, together with the **drawFontItalic**, **drawFontStrikethru**, **drawFontBold**, **drawFontName**, and **drawFontSize** properties, determines the font used for graphics text drawing methods.

The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

Note The **drawFonts** available in JADE vary according to your system configuration, display devices, and printing devices. Font-related properties can be set only to values for which actual **drawFonts** exist.

The font that is used is constructed only when a draw text method call is made, so the order of setting the property values does not matter as it does for controls.

drawMode

Type: Integer

Availability: Read or write at run time only

The **drawMode** property of the **Window** class, together with the **drawFontItalic**, **drawFontStrikethru**, **drawFontBold**, **drawFontName**, and **drawFontSize**, and **drawFontUnderline** properties, determines the appearance of output from graphics methods on a form or control.

Use the **drawMode** property to produce visual effects when drawing with the graphics methods. Windows compares each pixel in the draw pattern to the corresponding pixel in the existing background, and then it applies bit-wise operations. For example, setting **DrawMode_Xor** (7) uses the Xor operator to combine a draw pattern pixel with a background pixel.

The exact effect of the **drawMode** property setting depends on the way the color of a line drawn at run time combines with colors already on the screen. Settings **1**, **6**, **7**, **11**, **13**, and **16** yield perhaps the most-predictable results.

Note If you draw an object twice when the value is set to **DrawMode_Xor** (7), an object that is drawn twice has the original image restored. However, if the color black (**0**) is used for drawing, no image is drawn as the Xor operation results in the same color.

The settings of the **drawMode** property are listed in the following table.

Window Class Constant	Value	Description
DrawMode_Black	1	Black Pen.
DrawMode_Copy	13	Color specified by the **foreColor** property Copy Pen (the default value).
DrawMode_Invert	6	Inverse of the display color.
DrawMode_MaskPen	9	Combination of the colors common to both the pen and the display.
DrawMode_MaskNotPen	3	Combination of the colors common to the background color and the inverse of the pen.

Window Class Constant	Value	Description
DrawMode_MaskPenNot	5	Combination of the colors common to both the pen and the inverse of the display.
DrawMode_MergeNotPen	12	Combination of the display color and the inverse of the pen color.
DrawMode_MergePen	15	Combination of the pen color and the display color.
DrawMode_MergePenNot	14	Combination of the pen color and the inverse of the display color.
DrawMode_Nop	11	No operation, output remains unchanged. In effect, this setting turns drawing off.
DrawMode_NotCopyPen	4	Inverse of setting 13 (Copy Pen).
DrawMode_NotMaskPen	8	Inverse of setting 9 (Mask Pen).
DrawMode_NotMergePen	2	Inverse of setting 15 (Merge Pen).
DrawMode_NotXorPen	10	Inverse of setting 7 (Xor Pen).
DrawMode_White	16	White Pen.
DrawMode_Xor	7	Combination of the colors in the pen and in the display color, but not in both.

drawStyle

Type: Integer

Availability: Read or write at run time only

The **drawStyle** property of the **Window** class contains the line style for output from graphics methods on a form or control.

The settings of the **drawStyle** property are listed in the following table.

Window Class Constant	Value	Description
DrawStyle_Dash	1	Dash
DrawStyle_DashDot	3	Dash-dot
DrawStyle_DashDotDot	4	Dash-dot-dot
DrawStyle_Dot	2	Dot
DrawStyle_InsideSolid	6	Inside solid
DrawStyle_Solid	0	Solid (the default)
DrawStyle_Transparent	5	Transparent

drawTextAlign

Type: Integer

Availability: Read or write at run time only

The **drawTextAlign** property of the **Window** class contains the alignment used when outputting text on a form or control using the **drawTextAt** and **drawTextIn** graphics methods.

The settings of the **drawTextAlign** property are listed in the following table.

Window Class Constant	Value	Description
DrawTextAlign_Center	2	For the **drawTextAt** method, the text is positioned so that it is centered horizontally over the specified position. For the **drawTextIn** method, the text is centered within the specified rectangle.
DrawTextAlign_Left	0	Text is drawn starting at the specified positioned requested. This is the default value.
DrawTextAlign_Right	1	For the **drawTextAt** method, the text is positioned so that it ends at the specified position. For the **drawTextIn** method, the text is positioned so that it ends at the right hand edge of the requested rectangle.

drawTextCharRotation

Type: Integer

Availability: Read or write at run time only

The **drawTextCharRotation** property of the **Window** class specifies the angle, in degrees, between the base line and the *x*-axis of the form or control of each character. For example, a value of **90** draws the characters so that they are positioned on their side with their base parallel with the right hand edge of the form or control.

The default value is **0** degrees.

This property, in conjunction with the **drawTextRotation** property, allows the output of non-horizontal left to right text.

Use this property only with the **drawTextAt** method, as the rotated text could be rotated outside the rectangle defined by the **drawTextIn** method.

The **drawTextCharRotation** property, together with the **drawFontBold**, **drawFontStrikethru**, **drawFontItalic**, **drawFontName**, **drawFontSize**, **drawFontUnderline**, and **drawTextRotation** properties, determines the font used for graphics text drawing methods.

The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

drawTextRotation

Type: Integer

Availability: Read or write at run time only

The **drawTextRotation** property of the **Window** class specifies the angle, in degrees, between the base line of the text output and the *x*-axis of the form or control. For example, a value of **270** draws text upright down the form or control. The default value is **0** degrees.

This property, in conjunction with the **drawTextCharRotation** property, allows the output of non-horizontal left to right text.

Use this property only with the **drawTextAt** method, as the rotated text could be rotated outside the rectangle defined by the **drawTextIn** method.

The **drawTextRotation** property, together with the **drawFontBold**, **drawFontStrikethru**, **drawFontItalic**, **drawFontName**, **drawFontSize**, **drawFontUnderline**, and **drawTextCharRotation** properties, determines the font used for graphics text drawing methods.

The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

drawWidth

Type: Integer

Availability: Read or write at run time only

The **drawWidth** property of the **Window** class contains the line width for output from graphics methods on a form or control.

Set the **drawWidth** property to a value in the range **1** through **32,767**. This value represents the width of the line in pixels. The default value is **1** pixel wide.

Increase the value of the **drawWidth** property to increase the width of the line.

drawWindow

Type: Integer

Availability: Read or write at run time only

By default, any drawing on a control or window is done within the client area of the form or control. The client area is that part of the area inside the borders, scroll bars, and so on. The coordinates for the drawing are relative to the left, top, width, and height of this client area.

Change the value of the **drawWindow** property of the **Window** class to **1**, so that the entire surface of the form or control can be drawn on. The coordinates for the drawing are then relative to the left, top, width, and height of the entire window.

The default setting of zero (**0**) indicates a client area window.

A setting of **1** indicates that drawing is to cover the entire form or control window. Drawing on the entire surface, however, has the following limitations.

- The **clipControls** property has no effect on the drawing process. The drawing covers any other window at the drawn positions.
- The drawing coordinates are not affected by the scroll bar positions of the window.
- Scrolling the window still scrolls any drawing present on that window, as the scrolling process causes only a repaint of any uncovered area.
- Non-client areas (for example, borders or scroll bars) are drawn by a separate window event, which is not available to the JADE developer. Redrawing the non-client area is not necessarily accompanied by a normal **paint** event.

Graphics Methods

The graphical methods defined in the **Window** class are summarized in the following table.

Method	Description
beginBatchDrawing	Starts a batch drawing operation
clearGraphics	Causes a repaint of the form or control and its children, discarding any graphical images previously drawn
drawArc	Draws an arc
drawChord	Draws a chord (an arc with the end points joined and the interior filled)
drawDeskTopRectangle	Creates a transparent desktop window onto which the rectangle is drawn
drawDeskTopRectangleEx	Creates a transparent desktop window onto which the rectangle is drawn, using the inner border style and color
drawEllipse	Draws a filled ellipse (a circle is a special ellipse)
drawFilledPolygon	Draws a filled polygon and closes the figure by using the start and end **Real** point values
drawFilledRectangle	Draws a filled rectangle
drawFloodFill	Fills an area on the object with a color
drawGrid	Draws a grid on the window
drawLine	Draws a line
drawPictureAt	Draws a picture starting at a specified location using the drawing mode
drawPictureIn	Draws a picture stretched or compressed to exactly fit the rectangle
drawPie	Draws a pie (an arc with the end points drawn to the middle point of the bounding rectangle and color-filled)
drawPoint	Draws a single point
drawPolygon	Draws the border of a polygon without automatically closing the figure
drawRectangle	Draws the border of a rectangle
drawRoundRectangle	Draws a rectangle with rounded corners
drawSize	Returns the current number of entries in the drawing history
drawSolidRectangle	Draws a rectangle filled with the same color as the border
drawTextAt	Draws text at a position
drawTextHeight	Returns the height that would be required to draw a text string
drawTextIn	Draws text bounded by a rectangle
drawTextWidth	Returns the width that would be required to draw a text string
drawUndo	Undoes drawing actions back to a specified point
endBatchDrawing	Ends a batch drawing operation
getPoint	Gets the color of a point

Note An exception is raised if a **Window**, **Form**, or **Control** graphic method is invoked from a server method.

beginBatchDrawing

Signature `beginBatchDrawing();`

The **beginBatchDrawing** method of the **Window** class marks the start of a batch mode operation.

During batch mode, drawing requests are collected into a drawing script (as normally occurs when the **autoRedraw** property is set to **true**), but are not drawn. No evaluation of whether previous script entries are redundant is performed (for example, does a filled rectangle totally cover a previously drawn line?).

Use of batch drawing mode can reduce *flicker* associated with consecutive drawing operations and improve performance by using the same window drawing environment for the entire script.

An exception is raised if the **autoRedraw** property for the window is not set to **true** when the **beginBatchDrawing** method is called.

clearGraphics

Signature `clearGraphics();`

The **clearGraphics** method of the **Window** class causes the form or control to be redrawn and discards any graphics drawn on the object. If the **autoRedraw** property is set to **true**, any history for that control or form is discarded.

All drawing properties retain their current values after this call.

drawArc

Signature

```
drawArc(x1:     Real;
        y1:     Real;
        x2:     Real;
        y2:     Real;
        startX: Real;
        startY: Real;
        endX:   Real;
        endY:   Real;
        color:  Integer);
```

The **drawArc** method of the **Window** class draws an elliptical arc on a form or control using a pen the width of the **drawWidth** property value, using the style of the **drawStyle** property and the mode of the **drawMode** property.

If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The **drawArc** method parameters are listed in the following table.

Parameter	Description
x1, y1, x2, y2	Rectangle bounding the ellipse of which the arc is a part
startX, startY	Logical *x* and *y* (horizontal and vertical) coordinates of the point that defines the starting point of the arc
endX, endY	Logical *x* and *y* coordinates of the point that defines the end point of the arc
color	Color of the pen used

The parameter position units are **Real** primitive type values, in the units of the **scaleMode** property of the form or control.

The **startX**, **startY**, **endX**, and **endY** parameter points do not need to lie exactly on the arc.

The starting point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse.

The end point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified end point intersects the ellipse.

The arc is drawn in a counterclockwise direction. As an arc is not a closed figure, it is not filled.

The width and height of a rectangle must each be in the range 2 units through 32,767 units.

drawChord

Signature

```
drawChord(x1:     Real;
          y1:     Real;
          x2:     Real;
          y2:     Real;
          startX: Real;
          startY: Real;
          endX:   Real;
          endY:   Real;
          color:  Integer);
```

The **drawChord** method of the **Window** class draws a segment of an ellipse arc on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property.

If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The segment of the ellipse is bounded by an elliptical arc and the straight line that joins the end points of the arc. A line is drawn through the end points of the arc, and the figure is filled using the color and style of the **drawFillColor** and **drawFillStyle** properties of the object.

The **drawChord** method parameters are listed in the following table.

Parameter	Description
x1, y1, x2, y2	Rectangle bounding the ellipse of which the arc is a part
startX, startY	Logical *x* and *y* (horizontal and vertical) coordinates of the point that defines the starting point of the arc
endX, endY	Logical *x* and *y* coordinates of the point that defines the end point of the arc
color	Color of the pen used

The position units of the parameters are **Real** primitive type values in the units of the **scaleMode** property of the form or control. The **startX**, **startY**, **endX**, and **endY** parameter points do not need to lie exactly on the arc.

The starting point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse. The end point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified end point intersects the ellipse. The arc is drawn in a counterclockwise direction.

The width and height of a rectangle must each be in the range 2 units through 32,767 units.

drawDeskTopRectangle

Signature

```
drawDeskTopRectangle(borderStyle: Integer;
                     x1:          Integer;
                     y1:          Integer;
                     x2:          Integer;
                     y2:          Integer;
                     borderColor: Integer;
                     borderWidth: Integer);
```

The **drawDeskTopRectangle** method of the **Window** class creates a transparent desktop window onto which the rectangle is drawn. Any white pixels are treated as being transparent.

This window is used for any subsequent calls to the **drawDeskTopRectangle** method made on the associated window and is repositioned and resized each time.

Each call to the method erases any previous drawing and draws the new rectangle pattern.

Note The drawing does not use the Xor operator to combine a draw pattern pixel with a background pixel on the desktop.

The window is destroyed if the effective rectangle passed in the call is the same as the previously drawn rectangle or the window on which the call was made is destroyed. This means that existing logic that redraws the previous rectangle drawn (double xor) will work correctly. However, the drawing will flash as the window is repeatedly created and destroyed.

To improve the user experience, change the logic to remove the second draw call to erase the rectangle, and redraw it only when the window is no longer required.

The window is also destroyed if a call is made with **-1** as the first parameter (**borderStyle**).

Note Any window can have an associated drawing, thus allowing multiple rectangle patterns to be visible at once. Each drawing is independent of the other.

Use the **drawDeskTopRectangleEx** method if you want additional functionality; that is, you want the:

- Inner part of the rectangle drawn using the value of the **innerStyle** parameter and the color specified in the **innerColor** parameter unless the value of the **innerStyle** parameter is **-1**.
- Drawing window retained (that is, it is not destroyed) if the same rectangle is drawn a second time.
- Drawing window destroyed only if both the **borderStyle** and **innerStyle** parameters have a value of **-1** or the associated window is destroyed.

The parameters for the **drawDeskTopRectangle** method are listed in the following table.

Parameter	Description
borderStyle	-1 (destroy the drawing window)
	0 (hatch style 45 degrees left to right)
	1 (cross-hatch)
	2 (45 degree cross-hatch)
	3 (hatch style 45 degrees right to left)

Parameter	Description
	4 (horizontal)
	5 (vertical)
	6 (halftone)
	7 (solid)
	Any other value is treated as no drawing required (note that this method does *not* draw the inside of the rectangle and the rectangle border is not drawn if the value of the **borderWidth** parameter is less than or equal to zero)
x1, y1	Left and top corner positions of rectangle, respectively, in pixels relative to the client area of the window
x2, y2	Right and bottom corner positions of rectangle, respectively, in pixels relative to the client area of the window
borderColor	Color with which to draw the rectangle
borderWidth	Width of the inner area of the rectangle drawn with the pattern specified in the **borderStyle** parameter

If the width of the border area is greater than or equal to the width or height of the rectangle being drawn, the entire rectangle is filled with the specified pattern.

The methods in the following examples (in which **inDragOver: Boolean**, **lastMouseX: Integer**, and **lastMouseY: Integer** are form properties) show the use of this method to draw a dragging rectangle when the user drags the mouse over the **ListBox** control.

```
listBox1_dragOver(listBox: ListBox;
                  win:     Window;
                  x, y:    Real;
                  state:   Integer) updating;
vars
    w : Integer;
    h : Integer;
begin
    w := (listBox.clientWidth/2).Integer;
    h := (listBox.clientHeight/2).Integer;
    // draw in new position
    inDragOver := true;
    lastMouseX := x.Integer;
    lastMouseY := y.Integer;
    listBox.drawDeskTopRectangle(0, lastMouseX - w, lastMouseY - h,
                                 lastMouseX + w, lastMouseY + h, Red, 4);
end;

listBox1_dragDrop(listBox: ListBox; win: Window; x, y: Real) updating;
begin
    if inDragOver then
        listBox1.drawDeskTopRectangle(-1, lastMouseX - w, lastMouseY - h,
                                      lastMouseX + w, lastMouseY + h, Red, 4);
    endif;
end;
```

Applies to Version: 2016.0.02 (Service Pack 1) and higher

drawDeskTopRectangleEx

Signature

```
drawDeskTopRectangleEx(borderStyle: Integer;
                       x1:          Integer;
                       y1:          Integer;
                       x2:          Integer;
                       y2:          Integer;
                       borderColor: Integer;
                       borderWidth: Integer;
                       innerStyle:  Integer;
                       innerColor:  Integer);
```

The **drawDeskTopRectangleEx** method of the Window class creates a transparent desktop window onto which the rectangle is drawn. Any white pixels are treated as being transparent.

This window is used for any subsequent calls to the **drawDeskTopRectangleEx** method made on the associated window and is repositioned and resized each time.

Each call to the method erases any previous drawing and draws the new rectangle pattern.

Notes The drawing does not use the Xor operator to combine a draw pattern pixel with a background pixel on the desktop.

Any window can have an associated drawing, thus allowing multiple rectangle patterns to be visible at once. Each drawing is independent of the other.

The inner part of the rectangle is drawn using the value of the **innerStyle** parameter and the color specified in the **innerColor** parameter unless the value of the **innerStyle** parameter is **-1**.

The drawing window is not destroyed if the same rectangle is drawn a second time. The drawing window is destroyed only if the value of both the **borderStyle** and **innerStyle** parameters is **-1** or the associated window is destroyed.

The parameters for the **drawDeskTopRectangleEx** method are listed in the following table.

Parameter	Description
borderStyle	-1 (destroy the drawing window)
	0 (hatch style 45 degrees left to right)
	1 (cross-hatch)
	2 (45 degree cross-hatch)
	3 (hatch style 45 degrees right to left)
	4 (horizontal)
	5 (vertical)
	6 (halftone)
	7 (solid)
	Any other value is treated as no drawing required (note that this method does *not* draw the inside of the rectangle and the rectangle border is not drawn if the value of the **borderWidth** parameter is less than or equal to zero)

Parameter	Description
x1, y1	Left and top corner positions of rectangle, respectively, in pixels relative to the client area of the window
x2, y2	Right and bottom corner positions of rectangle, respectively, in pixels relative to the client area of the window
borderColor	Color with which to draw the rectangle
borderWidth	Width of the inner area of the rectangle drawn with the pattern specified in the **borderStyle** parameter
innerStyle	Style with which to draw the inner part of the rectangle (with the same pattern options as those for the **borderStyle** parameter listed earlier in this table)
innerColor	Color with to fill the inner part of the rectangle unless the value of the **innerStyle** parameter is **-1**

If the width of the border area is greater than or equal to the width or height of the rectangle being drawn, the entire rectangle is filled with the specified pattern.

The methods in the following examples (in which **inDragOver: Boolean**, **lastMouseX: Integer**, and **lastMouseY: Integer** are form properties) show the use of this method to draw a dragging rectangle when the user drags the mouse over the **ListBox** control.

```
listBox1_dragOver(listBox: ListBox;
                  win:     Window;
                  x, y:    Real;
                  state:   Integer) updating;
vars
    w : Integer;
    h : Integer;
begin
    w := (listBox.clientWidth/2).Integer;
    h := (listBox.clientHeight/2).Integer;
    // draw in new position
    inDragOver := true;
    lastMouseX := x.Integer;
    lastMouseY := y.Integer;
    listBox.drawDeskTopRectangleEx(0, lastMouseX - w, lastMouseY - h,
                               lastMouseX + w, lastMouseY + h, Red, 4, 0, Red);
end;

listBox1_dragDrop(listBox: ListBox; win: Window; x, y: Real) updating;
begin
    if inDragOver then
        listBox1.drawDeskTopRectangleEx(-1, lastMouseX - w, lastMouseY - h,
                                        lastMouseX + w, lastMouseY + h, Red, 4,
                                        -1, Red);
    endif;
end;
```

drawEllipse

Signature
```
drawEllipse(x1:    Real;
            y1:    Real;
            x2:    Real;
            y2:    Real;
            color: Integer);
```

The **drawEllipse** method of the **Window** class draws an ellipse on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The figure is filled using the color and style of the **drawFillColor** and **drawFillStyle** properties of the object. The **drawEllipse** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top points of the rectangle bounding the ellipse
x2, y2	Right and bottom points of the rectangle bounding the ellipse
color	Color of the pen used

The position units of the parameters are **Real** primitive type values in the units of the **scaleMode** property of the form or control.

If the width or the height of the bounding rectangle is zero (**0**), the function does not draw the ellipse.

The figure drawn by this method extends up to but does not include the right and bottom coordinates. This means that the height of the figure is **y2** through **y1**, and the width is **x2** through **x1**.

The width and the height of a rectangle must be in the range 2 units through 32,767 units. To draw an unfilled ellipse, set the **drawFillStyle** property to **DrawFillStyle_Transparent** (**1**).

The code fragment in the following example shows the use of the **drawEllipse** method.

```
    table.drawEllipse(20, 20, 50, 60, Red);
```

drawFilledPolygon

Signature
```
drawFilledPolygon(points: RealArray;
                  color:  Integer);
```

The **drawFilledPolygon** method of the **Window** class draws a polygon on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property, and automatically links the start points (*x1, y1*) to the end points (*xn, yn*, where the *n* value represents a matching set of left and right end points). If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The figure is filled using the color and style of the **drawFillColor** and **drawFillStyle** properties of the object. The **drawFilledPolygon** method parameters are listed in the following table.

Parameter	Description
points	Pairs of left and right (*x, y*) points, respectively, of the polygon
color	Color of the pen used

The position units of the points in the **RealArray** are values in the units of the **scaleMode** property of the form or control.

Note that there must be an even number of entries (*x, y* points) in the array and there must be at least two sets of points, as shown in the following example.

```
create realArray transient;
realArray.add(x1);
realArray.add(y1);
realArray.add(x2);
realArray.add(y2);
drawFilledPolygon(realArray, Black);
delete realArray;
```

To draw an unfilled polygon, use the **drawPolygon** method or set the **drawFillStyle** property to **DrawFillStyle_Transparent** (1). Alternatively, you can use the **drawPolygon** method to draw a many-sided irregular shape (that is, without automatically linking the start points (*x1, y1*) to the end points (*xn, yn*, where the *n* value represents a matching set of left and right end points).

drawFilledRectangle

Signature
```
drawFilledRectangle(x1:    Real;
                    y1:    Real;
                    x2:    Real;
                    y2:    Real;
                    color: Integer);
```

The **drawFilledRectangle** method of the **Window** class draws a rectangle on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The figure is filled using the color and style of the **drawFillColor** and **drawFillStyle** properties of the object.

The **drawFilledRectangle** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top points of the rectangle
x2, y2	Right and bottom points of the rectangle
color	Color of the pen used

The position units of the parameters are **Real** primitive type values in the units of the **scaleMode** property of the form or control. If the width or the height of the rectangle is zero (**0**), the function does not draw the rectangle.

The figure drawn by this method extends up to but does not include the right and bottom coordinates. This means that the height of the figure is **y2** through **y1**, and the width is **x2** through **x1**. The width and the height of a rectangle must be in the range 2 units through 32,767 units.

When you use the **drawFilledRectangle** or **drawSolidRectangle** method to draw a solid rectangle, previous figures that were drawn in the history but are covered and all property settings that have been made redundant are removed. In addition, the **drawLine** method removes any previous lines with the same co-ordinates where the same **width** and **drawStyle** property values apply. The automatic process does not occur if you use flood-fill or after logic calls to the **drawSize** method.

To draw an unfilled rectangle, use the **drawRectangle** method or set the **drawFillStyle** property to **DrawFillStyle_Transparent** (1).

The code fragment in the following example shows the use of the **drawFilledRectangle** method.

```
    frame.drawFilledRectangle(30,100,70,200, Red);
```

drawFloodFill

Signature

```
drawFloodFill(x1:    Real;
              y1:    Real;
              type:  Integer;
              color: Integer);
```

The **drawFloodFill** method of the **Window** class fills an area of the screen surface by using the brush defined by the **drawFillStyle** and **drawFillColor** properties of the object. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The type of flood fill specified determines which part of the screen is filled.

Note Only devices (which does not include printers) that support raster-display operations support the flood-fill operation. To print a shape that is flood-filled, use the **Control** class **createPicture** method to capture an image and then print the bitmap that is created.

The **drawFloodFill** method parameters are listed in the following table.

Parameter	Description
x1, y1	Starting points of the flood fill.
type	Type of flood fill. A value of zero (**0**) specifies that flood fill occurs until a border of "color" is reached. A value of **1** specifies that flood fill continues outward in all directions as long as the "color" is encountered, which is useful for filling areas that have multicolored boundaries.
color	Color of the pen used.

drawGrid

Signature

```
drawGrid(style:  Integer;
         width:  Real;
         height: Real;
         color:  Integer);
```

The **drawGrid** method of the **Window** class draws a grid on the window, using the **Window** class constants listed in the following table.

Window Class Constant	Value	Description
DrawGrid_Crosses	1	Small crosses drawn at the grid line intersection
DrawGrid_Dots	2	Dots drawn at the grid line intersections
DrawGrid_Lines	0	Horizontal and vertical grid lines

The **drawGrid** method parameters are listed in the following table.

Parameter	Description
style	**DrawGrid_Lines** (0), **DrawGrid_Crosses** (1), or **DrawGrid_Dots** (2)

Parameter	Description
width	Increment in units specified by the **scaleMode** property between each vertical grid line
height	Increment in units specified by the **scaleMode** property between each horizontal grid line
color	Color of the pen used to draw the grid

Grid lines for the left and top edges of the window are not drawn. The grid lines are drawn by using the **Window**::**drawWidth**, **Window**::**drawStyle**, and **Window**::**drawMode** properties. For the line style (that is, **DrawGrid_Lines**) when **drawWidth= 1**, **drawWindow= 0** (client area), and **scaleMode = 0** (that is, **ScaleMode_Pixels**), the result is the same as if you were to write the code in the following method.

```
vars
    x : Integer;
    y : Integer;
begin
    foreach x in width to clientWidth - 1 step width do
        window.drawLine(x, 0, x, clientHeight, color);
    endforeach;
    foreach y in height to clientHeight - 1 step height do
        window.drawLine(0, y, clientWidth, y, color);
    endforeach;
end;
```

drawLine

Signature
```
drawLine(x1:    Real;
         y1:    Real;
         x2:    Real;
         y2:    Real;
         color: Integer);
```

The **drawLine** method of the **Window** class draws a line on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property.

The line drawn by this method extends up to but does not include the end point. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**, as shown in the method in the following example.

```
redrawGraph() updating;
vars
    counter : Real;
    s       : String;
begin
    self.width  := self.parent.IGXFrame.clientWidth * 4;
    self.height := self.parent.IGXFrame.clientHeight;
    self.top  := 0;
    self.left := 0;
    pixelInc  := self.clientWidth div 4 div (self.increment + 1);
    self.autoRedraw := true;
    self.clearGraphics;
    self.drawWidth := 3;
    self.drawLine(0, 0, self.clientWidth, 0, 0);
    counter := 1;
    while counter <= increment * 4 do
        self.drawLine(counter * pixelInc, 0, counter * pixelInc, 5, 0);
```

```
            counter := counter + 1;
        endwhile;
        counter := 1;
        foreach s in labelArray do
            if counter.Integer div 2 * 2 = counter.Integer then
                drawFontSize := 7;
                drawTextAt(s, (counter * pixelInc) - 14, 6, 0);
            endif;
            counter := counter + 1;
        endforeach;
    end;
```

The **drawLine** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top start points of the line, respectively
x2, y2	Right and bottom end points of the line, respectively
color	Color of the pen used

The position units of the parameters are **Real** primitive type values in the units of the **scaleMode** property of the form or control.

drawPictureAt

Signature
```
drawPictureAt(pict: Binary;
              x:    Real;
              y:    Real);
```

The **drawPictureAt** method of the **Window** class draws a picture on a form or control starting at a specific location, using the drawing mode defined by the mode of the **drawMode** property.

If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The picture is drawn actual size. If there is insufficient room to fit the entire picture, it is truncated.

When running applications in JADE thin client mode, the picture that is drawn is cached on the presentation client.

The **drawPictureAt** method parameters are listed in the following table.

Parameter	Description
pict	Picture file image
x	Left starting position
y	Top starting position

The following example shows the use of the **drawPictureAt** method.

```
    form1.drawPictureAt(pic1.picture, 20, 30);
```

drawPictureIn

Signature
```
drawPictureIn(pict: Binary;
              x:    Real;
              y:    Real;
              x2:   Real;
              y2:   Real);
```

The **drawPictureIn** method of the **Window** class draws a picture stretched or compressed to exactly fit the rectangle bounded by the **x**, **y** and **x2**, **y2** parameter values on a form or control. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The drawing mode defined by the mode of the **drawMode** property is used.

When running applications in JADE thin client mode, the picture that is drawn is cached on the presentation client.

The **drawPictureIn** method parameters are listed in the following table.

Parameter	Description
pict	Picture image to be drawn
x, y	Left and top corner of the drawing rectangle
x2, y2	Bottom and right corner of the drawing rectangle

The code fragment in the following example shows the use of the **drawPictureIn** method.

```
form1.drawPictureIn(pic1.picture, 20, 20, 200, 300);
```

drawPie

Signature
```
drawPie(x1:     Real;
        y1:     Real;
        x2:     Real;
        y2:     Real;
        startX: Real;
        startY: Real;
        endX:   Real;
        endY:   Real;
        color:  Integer);
```

The **drawPie** method of the **Window** class draws a pie-shaped wedge on a form or control, using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property. The wedge is an elliptical arc whose center and two end points are joined by lines. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The figure is filled using the color and style of the **drawFillColor** and **drawFillStyle** properties of the object.

The **drawPie** method parameters are listed in the following table.

Parameter	Description
x1, y1, x2, y2	Rectangle bounding the ellipse of which the pie is a part
startX, startY	Logical *x* and *y* (horizontal and vertical) coordinates of the point that defines the starting point of the arc

Parameter	Description
endX, endY	Logical *x* and *y* coordinates of the point that defines the end point of the arc
color	Color of the pen used

The position units of the parameters are **Real** primitive type values in the units of the **scaleMode** property of the form or control.

The **startX**, **startY**, **endX**, and **endY** parameter points do not need to lie exactly on the arc.

The starting point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse. The end point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified end point intersects the ellipse.

The figure drawn by this function extends up to but does not include the right and bottom coordinates, so that the height of the figure is **y2** through **y1** and the width is **x2** through **x1**. The arc is drawn in a counterclockwise direction.

The width and height of a rectangle must each be in the range 2 units through 32,767 units.

drawPoint

Signature

```
drawPoint(x1:    Real;
          y1:    Real;
          color: Integer);
```

The **drawPoint** method of the **Window** class draws a point on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The point is drawn as a line of length 1. The position units of the parameters are **Real** primitive type values in the units of the **scaleMode** property of the form or control. The **drawPoint** method parameters are listed in the following table.

Parameter	Description
x1, y1	Horizontal and vertical position of the point to be drawn
color	Color of the pen used

drawPolygon

Signature

```
drawPolygon(points: RealArray;
            color:  Integer);
```

The **drawPolygon** method of the **Window** class draws the border of a polygon on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property. The inside of the polygon is not filled. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The **drawPolygon** method parameters are listed in the following table.

Parameter	Description
points	Pairs of left and right (*x, y*) points, respectively, of the polygon
color	Color of the pen used

The figure drawn by this polygon is equivalent to using the **drawFilledPolygon** method with the **drawFillStyle** property set to **DrawFillStyle_Transparent** (1), except that the start points (*x1, y1*) and end points (*xn, yn*, where the *n* value represents a matching set of left and right end points) are not automatically linked. You can therefore use the **drawPolygon** method to draw a many-sided irregular shape.

Note that there must be an even number of entries (*x, y* points) in the array and there must be at least two sets of points, as shown in the following example.

```
create realArray transient;
realArray.add(x1);
realArray.add(y1);
realArray.add(x2);
realArray.add(y2);
realArray.add(x3);
realArray.add(y3);
drawPolygon(realArray, Black);
delete realArray;
```

drawRectangle

Signature

```
drawRectangle(x1:    Real;
              y1:    Real;
              x2:    Real;
              y2:    Real;
              color: Integer);
```

The **drawRectangle** method of the **Window** class draws the border of a rectangle on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property.

If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The inside of the rectangle is not filled. If the width or the height of the rectangle is zero (**0**), the function does not draw the rectangle.

The figure drawn by this function extends up to and includes the right and bottom coordinates.

The figure drawn by this rectangle is equivalent to using the **drawFilledRectangle** method with the **drawFillStyle** property set to **DrawFillStyle_Transparent** (1).

The **drawRectangle** method parameters are listed in the following table.

Parameter	Description
x1, y1	Horizontal and vertical left and top points of the rectangle
x2, y2	Horizontal and vertical right and bottom points of the rectangle
color	Color of the pen used

The code fragment in the following example shows the use of the **drawRectangle** method.

```
frame.drawRectangle(30, 100, 70, 200, Red);
```

drawRoundRectangle

Signature
```
drawRoundRectangle(x1:         Real;
                   y1:         Real;
                   x2:         Real;
                   y2:         Real;
                   xRoundSize: Real;
                   yRoundSize: Real;
                   color:      Integer);
```

The **drawRoundRectangle** method of the **Window** class draws a rectangle with rounded corners on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The figure is filled using the **drawFillColor** and **drawFillStyle** properties of the object. If the width or the height of the rectangle is zero (**0**), the function does not draw the rectangle.

The figure drawn by this function extends up to but does not include the right and bottom coordinates, meaning that the height of the figure is **y2** through **y1** and the width is **x2** through **x1**.

The width and the height of a rectangle must be in the range 2 units through 32,767 units.

The **drawRoundRectangle** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top points of the rectangle
x2, y2	Right and bottom points of the rectangle
xRoundSize	Width of ellipse for rounded corners
yRoundSize	Height of ellipse for rounded corners
color	Color of the pen used

drawSize

Signature
```
drawSize(): Integer;
```

The **drawSize** method of the **Window** class returns the number of entries in the drawing history. If the value of the **autoRedraw** property is **false**, the **drawSize** method returns zero (**0**). There are entries in the drawing history only if drawing has been performed on the form or control since the last **clearGraphics** or **drawUndo** method call.

The **drawSize** method provides the current size of the drawing history. Use this method to record a rollback point for the image to be used at a later point.

drawSolidRectangle

Signature
```
drawSolidRectangle(x1:    Real;
                   y1:    Real;
                   x2:    Real;
                   y2:    Real;
                   color: Integer);
```

The **drawSolidRectangle** method of the **Window** class draws a rectangle on a form or control using a colored pen the width of the **drawWidth** property, the style of the **drawStyle** property, and the mode of the **drawMode** property.

If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The figure is solidly filled using the same color as the border. The **drawFillColor** and **drawFillStyle** properties are ignored.

The **drawSolidRectangle** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top points of the rectangle
x2, y2	Right and bottom points of the rectangle
color	Color of the pen used

The position units are **Real** primitive type values, in the units of the **scaleMode** property of the form or control.

If the width or the height of the rectangle is zero (**0**), the function does not draw the rectangle.

The figure drawn by this function extends up to but does not include the right and bottom coordinates, meaning that the height of the figure is **y2** through **y1** and the width is **x2** through **x1**.

The figure drawn by this rectangle is equivalent to using the **drawFilledRectangle** method with the **drawFillColor** property set to the appropriate color and the **drawFillStyle** property set to **DrawFillStyle_Solid** (0).

When you use the **drawFilledRectangle** or **drawSolidRectangle** method to draw a solid rectangle, previous figures that were drawn in the history but are covered and all property settings that have been made redundant are removed. In addition, the **drawLine** method removes any previous lines with the same co-ordinates where the same **width** and **drawStyle** property values apply. The automatic process does not occur if you use flood-fill or after logic calls to the **drawSize** method.

drawTextAt

Signature

```
drawTextAt(text:  String;
           x1:    Real;
           y1:    Real;
           color: Integer);
```

The **drawTextAt** method of the **Window** class draws a text string on a form or control using the current values of the **drawFont** and **drawTextAlign** properties. If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

Note The **drawTextAt** property is not affected by the mode defined in the **drawMode** property.

The **drawTextAt** method parameters are listed in the following table.

Parameter	Description
text	Text string that is to be drawn
x1, y1	Horizontal and vertical positions for the text
color	Color of the text

The way in which the text is drawn is determined by the value of the **drawTextAlign** property, as listed in the following table.

Window Class Constant	Value	Description
DrawTextAlign_Left	0	Left-aligned
DrawTextAlign_Right	1	Right-aligned
DrawTextAlign_Center	2	Center-aligned (centered)

The text is drawn in a single line, unless it has embedded carriage return characters within it. Each embedded carriage return character forces a new line for the remaining text.

The text always starts at the vertical point specified by the **y1** parameter.

drawTextHeight

Signature `drawTextHeight(text: String): Real;`

The **drawTextHeight** method of the **Window** class returns the height of the graphical text output that would result from drawing the passed text string with the **drawTextAt** method. The **text** parameter is the text string.

The returned value is a **Real** primitive type, in the units of the **scaleMode** property of the form or control.

The height of the text is calculated based on a single line of text, unless it has embedded carriage return characters within it forcing new lines.

drawTextIn

Signature

```
drawTextIn(text:  String;
           x1:    Real;
           y1:    Real;
           x2:    Real;
           y2:    Real;
           color: Integer);
```

The **drawTextIn** method of the **Window** class draws a text string on a form or control within a bounded rectangle, using the current values of the **drawFont** and **drawTextAlign** properties.

Note The **drawTextIn** property is not affected by the mode defined in the **drawMode** property.

The **drawTextIn** method parameters are listed in the following table.

Parameter	Description
text	Text string that is to be drawn
x1, y1	Left and top points of the bounding rectangle
x2, y2	Right and bottom points of the bounding rectangle
color	Color of the text

The text is drawn into the bounding rectangle with word wrap.

The parameter position units are **Real** primitive type values, in the units of the **scaleMode** property of the form or control. The way in which the text is drawn is determined by the value of the **drawTextAlign** property, as listed in the following table.

Window Class Constant	Value	Description
DrawTextAlign_Left	0	Left-aligned
DrawTextAlign_Right	1	Right-aligned
DrawTextAlign_Center	2	Center-aligned (centered)

Any embedded carriage return character within the text forces a new line for the remaining text. The text always starts at the vertical point specified by the **y1** parameter.

If this method is not called from a **paint** event, set the **autoRedraw** property to **true**.

The code fragment in the following example shows the use of the **drawTextIn** method.

```
cntrl.drawTextIn(cntrl.name, 0, 30, 100, 70, labelColor);
```

drawTextWidth

Signature `drawTextWidth(text: String): Real;`

The **drawTextWidth** method of the **Window** class returns the width of the graphical text output that would result from drawing the passed text string with the **drawTextAt** method. The **text** parameter is the text string.

The returned value is a **Real** primitive type in the units of the **scaleMode** property of the form or control. The width of the text is calculated based on a single line of text, unless it has embedded carriage return characters within it forcing new lines.

The method in the following example shows the use of the **drawTextWidth** method.

```
centerText(graphWin: Window; text: String;
           pos: Integer; vertical: Boolean): Integer;
begin
    if vertical then
        return(pos + (graphWin.drawTextHeight(text).Integer/2)).Integer;
    else
        return(pos - (graphWin.drawTextWidth(text).Integer/2)).Integer;
    endif;
end;
```

drawUndo

Signature `drawUndo(toPosn: Integer);`

The **drawUndo** method of the **Window** class rolls the graphics history back to a previous point in the image construction and removes the history past the indicated point. A repaint is then caused to draw the image as it was at that point. The **toPosn** parameter is the last graphics command to be retained.

If the setting of the **autoRedraw** property is **false** or if there is no history, this method has no effect. After the command has completed, the image is painted at the indicated history point. The graphical attributes are also reset to the values as at that point.

The **drawUndo**(**0**)**;** method does the same as **clearGraphics** method, except that the undo restores the drawing attributes back to what they were at the start of the history recording process, while the **clearGraphics** method retains the current values.

The size of the draw history can be determined at any point, by using the **drawSize** method.

endBatchDrawing

Signature `endBatchDrawing();`

The **endBatchDrawing** method of the **Window** class marks the end of a batch mode operation and requests that the window redraw itself by replaying the drawing script.

If the window is repainted while still in batch mode, batch mode is cancelled and another paint request issued. This is necessary if the whole window has not been redrawn.

During batch mode, drawing requests are collected into the drawing script (as normally occurs when the **autoRedraw** property is set to **true**), but are not drawn. No evaluation of whether previous script entries are redundant is performed (for example, does a filled rectangle totally cover a previously drawn line?).

Use of batch drawing mode can reduce *flicker* associated with consecutive drawing operations and improve performance by using the same window drawing environment for the entire script.

getPoint

Signature
```
getPoint(x1: Real;
         y1: Real): Integer;
```

The **getPoint** method of the **Window** class returns the color of the pixel at the specified coordinates on the screen. The point must be physically visible and within the referenced form or control. If it is not, the **getPoint** method does nothing. The **x1** and **y1** parameters are the coordinates of the point.

Only devices that support raster operations can use this function.

The code fragment in the following example shows the use of the **getPoint** method.

```
if getPoint (1,2) <> Blue then
    picturePreview.drawFilledRectangle(1, 2, 30, 5, Blue);
    picturePreview.drawFilledRectangle(picturePreview.scaleWidth-12,
              2, picturePreview.scaleWidth, 5, Blue);
    picturePreview.drawLine(1, 7, picturePreview.scaleWidth, 7, Red);
endif;
```