Encyclopaedia of Primitive Types

VERSION 7.1.08

jade™

Contents

Contents iv

Contents v

Contents vii

Contents x

Before You Begin

The *JADE Encyclopaedia of Primitive Types* 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 Primitive Types* is expected to be developers of JADE application software products.

What's Included in this Encyclopaedia

The *JADE Encyclopaedia of Primitive Types* has one chapter and one appendix.

Chapter 1	Gives a reference to primitive types and the methods that they provide
Appendix A	Gives a reference to global constants

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 Developer's Reference	Developing or maintaining JADE applications
JADE Encyclopaedia of Classes	System classes (Volumes 1 and 2), Window classes (Volume 3)
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 Report Writer User's Guide	Using the JADE Report Writer to develop and run reports
JADE Synchronized Database Service (SDS) Administration Guide	Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)
JADE Thin Client Guide	Administering JADE thin client environments

Conventions

The *JADE Encyclopaedia of Primitive Types* 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 "isKindOf" 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 CAPITALS	Keyboard 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.

In this document, the term Microsoft *Windows* refers to Windows 10, Windows 8, Windows 7, Windows Server 2012, Windows Server 2008, Windows Vista, or Windows Mobile. When there are differences between the versions of Microsoft Windows, the specific version of Microsoft Windows is stated.

With the exception of the **jade.exe** program, when referring to program executables in this document, the **.exe** file suffix is omitted; for example, **jadclient** refers to **jadclient.exe**. Similarly, the **.dll** (Dynamic Link Library) suffix is omitted. For example, **jomos** refers to **jomos.dll**.

Chapter 1 Primitive Types

This chapter covers the following topics.

Overview

The type of a method or property determines the range of values that the method or property can take and its interface (or protocol). A type can be a primitive type, a class, or a JADE interface.

The primitive types are summarized in the following table.

Primitive Type	Description
Any	Represents any object reference or primitive value
Binary	Represents binary data
Boolean	Contains Boolean value **true** or **false**
Byte	A single byte unsigned value (8 bits)
Character	Any single ANSI or Unicode character
Date	Julian day number
Decimal	Number with specific decimal format
Integer	Signed 32-bit integer (whole number)
Integer64	Signed 64-bit integer (whole number)
MemoryAddress	Represents a memory address
Point	Represents *x* and *y* coordinates of a point
Real	Floating point number
String	Sequence of characters
StringUtf8	String encoded in the UTF-8 format
Time	Time since midnight (in milliseconds)
TimeStamp	Date and time that includes combined date and time values
TimeStampInterval	Represents the difference between two timestamp values

Primitive types have a defined **null** value, which can be tested for by using the **null** language identifier; for example:

```
if d = null then
```

You can associate methods with primitive types, but you cannot associate properties with primitive types.

Properties that are defined as primitive types represent a value. They do not represent a reference to an object.

With the exception of the **Any** primitive type, which can represent any object reference or primitive value, the value of the property is stored in the parent object record when you define a property that is a primitive type. (A property that is an object contains a reference to the object.)

Notes A temporary value is created if the return value of a primitive type method is passed to an updating primitive method. On completion of the updating method, this temporary value is discarded.

You cannot specify the **clientExecution** and **serverExecution** method options on primitive type methods. Methods defined on primitive types are always executed in the node of the calling method.

Any Type

A variable of type **Any** can contain an object reference or any primitive value.

Note The **Any** primitive type can be used only for local variables, parameters, and return types. You cannot define a property of type **Any**.

To determine the type of the value associated with a variable of type Any

- Use the **isKindOf** method.

The **Any** primitive type is useful when a:

- Method can return either an object reference or primitive value
- Parameter in a method can be either an object reference or a primitive value
- Variable can receive either an object reference or a primitive value

The following example shows the use of the **Any** primitive type.

```
userNotification(eventType: Integer; userInfo: Any);
vars
    file : DbFile;
begin
    if eventType = 19 then
        file := userInfo.DbFile;
        ...
    endif;
end;
```

For details about the methods defined in the **Any** primitive type, see "Any Methods", in the following subsection.

For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Any Methods

The methods defined in the **Any** primitive type are summarized in the following table.

Method	Returns …
asString	A string representing the value of a primitive type or the object id of an object reference
display	The string **"Any"**
getName	A string representing the value of a primitive type or the name of the receiver class if it is an object reference
isKindOf	**true** if the type of the receiver is of the type specified by the **type** parameter

asString

Signature `asString(): String;`

The **asString** method of the **Any** primitive type returns:

- A string representing the value of the primitive type, if the receiver is a primitive type
- A string containing the object identifier (oid) of the object reference, if the receiver is an object reference
- A **null** string, if the receiver is **null**

display

Signature `display(): String;`

The **display** method of the **Any** primitive type returns the string **"Any"**.

getName

Signature `getName(): String;`

The **getName** method of the **Any** primitive type returns:

- A string containing the class name of the object reference, if the receiver is an object reference
- A string representing the value of a primitive type, if the receiver is a primitive type
- A **null** string, if the receiver is **null**

isKindOf

Signature `isKindOf(type: Type): Boolean;`

The **isKindOf** method of the **Any** primitive type returns the **Boolean** value of **true** if the type of the **Any** variable is of the type specified by the **type** parameter. If the variable is a different type to that specified by the **type** parameter, the **isKindOf** method returns **false**.

The code fragment in the following example shows the use of the **isKindOf** method.

```
if not any.isKindOf(Object) then
    if any.isKindOf(Any) then
        return "not a valid reference";
    else
        return any.String;
    endif;
endif;
```

For example, **any.isKindOf(Integer)** returns **true** if the any variable contains an **Integer**, and **any.isKindOf (Customer)** returns **true** if the any variable contains a reference to an instance of the **Customer** class or one of its subclasses.

The following example shows the use of the **isKindOf** method.

```
vars
    date : Date;
begin
    date := fault.openDate;
```

```
        if fault.isKindOf(GenuineFault) then
            opFault.value := true;
        elseif ... then
           ...
        endif;
    end;
```

Binary Type

Use the **Binary** primitive type to define **Binary** variables and attributes.

When you specify a length less than or equal to **540** for a **Binary** attribute, it is embedded. Space is allocated within instances of the class to store a binary value with a length less than or equal to the specified length.

When you specify a length greater than **540** or you select the **Maximum Length** check box (which corresponds to 2,147,483,647 bytes) for a **Binary** attribute, it is not embedded. It is stored in a separate variable-length object, a Binary Large Object (blob), which can store a binary value with a length less than or equal to the specified length. The amount of storage required for a blob is determined by the binary value.

Binary variables can be bounded or unbounded, as shown in the following code fragment.

```
vars
   bin1 : Binary[100]; // Bounded - bin1 can store a binary value with a
                       // length less than or equal to 100 bytes
   bin2 : Binary;      // Unbounded - bin2 can store a binary value with a
                       // length less than or equal to 2,147,483,647 bytes
```

To specify a substring **bin[*m*:*n*]** of a **Binary** value **bin**, use two integers separated by a colon (**:**) character. The first integer is the start position and the second integer (following the colon (**:**) character) is the length of the binary substring or **end**, to indicate the end of the binary string. The first byte is at position **1**.

A variable of type **Byte** can be used to reference a single byte in a binary value, in effect treating the binary value as an array of bytes, as shown in the following code fragment.

```
vars
   bin : Binary;
   byte : Byte;
begin
   bin := app.loadPicture("C:\Jade\bin\jade.bmp");
   byte := bin[716];       // 716th byte of the binary data in bin
```

For details about the constants and methods defined in the **Binary** primitive type, see "Binary Constants" and "Binary Methods", in the following subsections. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Binary Constants

The **Binary** primitive type provides the constants listed in the following table, for use with the **compressToBinary** methods in the **Binary**, **String**, and **StringUtf8** primitive types.

Constant	Integer Value	Description
Compression_ZLib	1402	String and binary compression to binary using ZLIB level 5 (256*5 + 122)
Compression_ZLibFast	378	String and binary compression to binary using ZLIB level 1 (256*1 + 122)
Compression_ZLibSmall	2426	String and binary compression to binary using ZLIB level 9 (256*9 + 122)

Binary Methods

The methods defined in the **Binary** primitive type are summarized in the following table.

Method	Returns …
ansiToString	The string equivalent of the binary interpreted as ANSI characters
ansiToUnicode	The Unicode string equivalent of the binary interpreted as ANSI characters
asDecimal	The **Decimal** representation of the receiver
asGuidString	A visual representation of the Globally Unique Identifier (GUID) binary receiver as a string of printable characters
base64Encode	An ASCII string consisting of lines with fewer than 76 characters resulting from encoding the receiver in Base64
base64EncodeNoCrLf	An ASCII string resulting from encoding the receiver in Base64 without carriage-return and line-feed characters
bufferAddress	The value of the pointer to the internal buffer as an integer
bufferMemoryAddress	The value of the pointer to the internal buffer as a memory address
compressToBinary	A compressed binary representation of the receiver
convertPicture	A copy of the receiver converted to the requested picture type
convertToFile	A copy of the receiver converted to the requested picture type in a file
display	A string containing a hexadecimal dump of the receiver
fromANSIToString	A String containing the receiver converted using the code page for the specified locale
fromANSIToStringUtf8	A UTF8 String containing the receiver converted using the code page for the specified locale
length	The actual length of a binary variable
maxLength	The declared maximum length of a binary variable
pictureSize	The type of image contained in the binary and the width and height of the image
pictureType	The type of picture image
posBinary	The position of a specified binary string in the receiver
posByte	The position of a specified byte in the receiver
uncompressToBinary	A binary value representing the receiver after it has been uncompressed
uncompressToString	An string value representing the receiver after it has been uncompressed
uncompressToStringUtf8	A UTF8 string value representing the receiver after it has been uncompressed
unicodeToAnsi	The ANSI string equivalent of the binary interpreted as Unicode characters
unicodeToString	The string equivalent of the binary interpreted as Unicode characters
unpackCString	A string extracted from the binary, starting at a specified position within the binary and terminated by the next occurring null character
uuidAsString	A string formatted as a Universally Unique Identifier (UUID)

ansiToString

Signature `ansiToString(): String;`

The **ansiToString** method of the **Binary** primitive type interprets the binary as ANSI characters and returns a copy converted to a string.

When invoked from an ANSI application, an ANSI string is returned. When invoked from a Unicode application, the binary is converted from ANSI to Unicode, and a Unicode string is returned.

The code fragment in the following example shows the use of the **ansiToString** method.

```
str := bin.ansiToString;
```

Note When converting from ANSI to Unicode, conversion stops at the first **null** character. If the binary contains embedded nulls, the string returned from the ANSI to Unicode conversion therefore represents only that part of the binary up to the first **null** character.

ansiToUnicode

Signature `ansiToUnicode(): String;`

The **ansiToUnicode** method of the **Binary** primitive type interprets the binary as ANSI characters and returns a copy converted to a Unicode string. This method can be invoked only from a Unicode application.

If this method is invoked from an ANSI application, the following exception is raised.

```
1000    Invalid parameter type
```

Note Conversion of the binary stops at the first **null** character. If the binary contains embedded nulls, the string returned from the conversion therefore represents only that part of the binary up to the first **null** character.

asDecimal

Signature `asDecimal(): Decimal;`

The **asDecimal** method of the **Binary** primitive type returns the decimal value for a **Binary** value that was obtained by a call to the **asBinary** method of the **Decimal** primitive type.

The following example shows the use of the **asDecimal** method.

```
vars
    bin : Binary;
    dec : Decimal;
begin
    dec := 123.456.Decimal;
    bin := dec.asBinary;
    write bin.asDecimal;    // Outputs 123.456
end;
```

Use the **asDecimal** method in preference to type casting; for example:

```
bin := dec.asBinary;   // This is preferable to "bin := dec.Binary;"
```

asGuidString

Signature `asGuidString(): String;`

The **asGuidString** method of the **Binary** primitive type returns a visual representation of the Globally Unique Identifier (GUID) binary receiver as a string of printable characters, in the following format.

```
"{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}"
```

Binary class identifiers are used in ActiveX control and automation libraries, for example, and they take less space than a visual string representation. This method raises an exception if the receiver is not a valid GUID; that is, it is not a binary of length 16.

See also the **String** primitive type **asGuid** method.

base64Encode

Signature `base64Encode(): String;`

The **base64Encode** method of the **Binary** primitive type returns an ASCII string resulting from the encoding of the receiver using the Base64 encoding technique defined in RFC 1521.

Base64 encoding enables 8-bit data to be converted, so that it can be transmitted over a protocol that supports 7-bit characters only. Base64 encoding also provides enhanced privacy when the source data is standard ASCII text, as the message is no longer in clear text when it is transmitted.

The output string is represented in lines not exceeding 76 characters that are terminated with carriage return and line feed (**Cr** and **Lf**) characters.

Use the **base64Decode** method on the **String** primitive type to decode a Base64-encoded string.

The following example shows the use of the **base64Encode** method.

```
vars
    bin: Binary;
    file: File;
begin
    create file;
    file.fileName := "d:\temp\harry.jpg";
    file.kind := File.Kind_Binary;
    file.open;
    bin := file.readBinary(file.fileLength);
    write 'original length = ' & bin.length.String;
    write 'base64Encode length = ' & bin.base64Encode().length.String;
    write 'base64EncodeNoCrLf length = ' &
                        bin.base64EncodeNoCrLf().length.String;
    write 'base64Decode length = ' &
                        bin.base64Encode().base64Decode().length.String;
    write 'base64Decode length (from NoCrLf) = ' &
                        bin.base64EncodeNoCrLf().base64Decode.length.String;
    file.close;
epilog
    delete file;
end;
```

Note The length of an encoded string is about a third longer, even if the string is encoded with carriage-return and line-feed (**Cr** and **Lf**) characters.

base64EncodeNoCrLf

Signature `base64EncodeNoCrLf(): String;`

The **base64EncodeNoCrLf** method of the **Binary** primitive type returns an ASCII string resulting from the encoding of the receiver using the Base64 encoding technique defined in RFC 1521.

Base64 encoding enables 8-bit data to be converted, so that it can be transmitted over a protocol that supports 7-bit characters only. Base64 encoding also provides enhanced privacy when the source data is standard ASCII text, as the message is no longer in clear text when it is transmitted.

Unlike the **base64Encode** method, the output is not broken up into lines; that is, it does not contain carriage-return and line-feed (**Cr** and **Lf**) characters.

Use the **base64Decode** method on the **String** primitive type to decode a Base64-encoded string.

The following example shows the use of the **base64EncodeCrLf** method.

```
vars
    bin: Binary;
    file: File;
begin
    create file;
    file.fileName := "d:\temp\harry.jpg";
    file.kind := File.Kind_Binary;
    file.open;
    bin := file.readBinary(file.fileLength);
    write 'original length = ' & bin.length.String;
    write 'base64Encode length = ' & bin.base64Encode().length.String;
    write 'base64EncodeNoCrLf length = ' &
                            bin.base64EncodeNoCrLf().length.String;
    write 'base64Decode length = ' &
                            bin.base64Encode().base64Decode().length.String;
    write 'base64Decode length (from NoCrLf) = ' &
                            bin.base64EncodeNoCrLf().base64Decode.length.String;
    file.close;
epilog
    delete file;
end;
```

Note The length of an encoded string is about a third longer, even if the string is encoded with carriage-return and line-feed (**Cr** and **Lf**) characters.

bufferAddress

Signature `bufferAddress(): Integer;`

The **bufferAddress** method of the **Binary** primitive type returns an integer containing the value of the pointer to the internal buffer that contains the binary. This value may be required when a JADE **Binary** primitive type value is being mapped to a structured record type for a call to an external function.

Call the **bufferAddress** method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

Caution Do not use this method to pass the address of a binary to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a **jade.exe** (thin client) fault in the called function.

The method in the following example shows the use of the **bufferAddress** method to initialize the Windows **SECURITY_DESCRIPTOR** and **SECURITY_ATTRIBUTES** structures.

```
constants
    // Current security descriptor revision value
    SECURITY_DESCRIPTOR_REVISION = 1;
vars
    result              : Boolean;
    securityDescriptor  : Binary[20];
    securityAttributes  : Binary[9];
begin
    ...   // Call the Windows API to initialize the security descriptor
    result := call initializeSecurityDescriptor(securityDescriptor,
              SECURITY_DESCRIPTOR_REVISION);
    // Return Windows error if unable to initialize security descriptor
    if not result then
        return call getLastError;
    endif;
    // The first field (DWORD) in the security attributes structure is the
    // size (in bytes) of the structure
    securityAttributes[1:4] := securityAttributes.length.Binary;
    // The second field (LPVOID) points to the security descriptor
    // Set the value to the actual address of the buffer
    securityAttributes[5:4] := securityDescriptor.bufferAddress.Binary;
    ...
end;
```

bufferMemoryAddress

Signature `bufferMemoryAddress(): MemoryAddress;`

The **bufferMemoryAddress** method of the **Binary** primitive type returns a memory address containing the value of the pointer to the internal buffer that contains the binary. This value may be required when a JADE **Binary** primitive type value is being mapped to a structured record type for a call to an external function.

Call the **bufferMemoryAddress** method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

Caution Do not use this method to pass the address of a binary to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a **jade.exe** (thin client) fault in the called function.

The method in the following example shows the use of the **bufferMemoryAddress** method to initialize the Windows **SECURITY_DESCRIPTOR** and **SECURITY_ATTRIBUTES** structures.

```
constants
    // Current security descriptor revision value
    SECURITY_DESCRIPTOR_REVISION = 1;
vars
    result              : Boolean;
    securityDescriptor : Binary[20];
    securityAttributes : Binary[9];
begin
    ...   // Call the Windows API to initialize the security descriptor
    result := call initializeSecurityDescriptor(securityDescriptor,
              SECURITY_DESCRIPTOR_REVISION);
    // Return Windows error if unable to initialize security descriptor
    if not result then
        return call getLastError;
    endif;
    // The first field (DWORD) in the security attributes structure is the
    // size (in bytes) of the structure
    securityAttributes[1:4] := securityAttributes.length.Binary;
    // The second field (LPVOID) points to the security descriptor
    // Set the value to the actual address of the buffer
    securityAttributes[5:4] :=
              securityDescriptor.bufferMemoryAddress.asBinary32;
end;
```

compressToBinary

Signature `compressToBinary(typeAndOption: Integer): Binary;`

The **compressToBinary** method of the **Binary** primitive type returns a compressed binary representation of the binary of the receiver using the **ZLIB** compression value specified by the **typeAndOption** parameter, which can be one of the **Binary** primitive type constants listed in the following table.

Constant	Integer Value	Description
Compression_ZLib	1402	String and binary compression to binary using ZLIB level 5 (256*5 + 122)
Compression_ZLibFast	378	String and binary compression to binary using ZLIB level 1 (256*1 + 122)
Compression_ZLibSmall	2426	String and binary compression to binary using ZLIB level 9 (256*9 + 122)

Note This method adds the type byte to the front of the compressed binary. This type byte is ignored when the value is used in a JADE system but if the data is to be passed to an external library, it is your responsibility to remove the type byte, if necessary.

You cannot concatenate the results of multiple **compressToBinary** method calls.

You must use the **Binary** primitive type **uncompressToBinary** method to uncompress a binary value from this binary representation.

convertPicture

Signature `convertPicture(type: Integer): Binary;`

The **convertPicture** method of the **Binary** primitive type returns a copy of a binary picture image converted to the picture type specified in the **type** parameter. The types of images that can be converted (by using the **Window** class **PictureType_Bitmap**, **PictureType_ Jpeg**, **PictureType_ Jpeg2000**, **PictureType_Png**, or **PictureType_ Tiff** constant) are as follows.

- Bitmap (.**bmp**)
- Tag Image File Format (.**tif**)
- Joint Photographic Experts Group (.**jpg**)
- JPG 2000 (.**jp2**)
- Portable Network Graphics (.**png**)

A 14015 (File does not contain an image type that can be handled) exception is raised if the receiver does not contain valid image data.

Notes Converting to a .**tif** image type file results in a tiff packbits type image. Converting to a .**jpg** image results in loss of quality in the picture, as Joint Photographic Experts Group (JPEG) uses a *lossy* compression algorithm.

As the Portable Network Graphics image uses a *lossless* compression algorithm, it provides clarity and retains definition for images, but the files may be larger than JPEG files.

An exception is raised if this method is invoked from a server method.

You cannot convert images to GIF picture files.

See also the **convertToFile** method of the **Binary** primitive type.

convertToFile

Signature
```
convertToFile(filename: String;
              type:     Integer);
```

The **convertToFile** method of the **Binary** primitive type saves a copy of a binary picture image converted to the picture type specified in the **type** parameter, in the file specified in the **filename** parameter. If the **filename** parameter is null (**""**), the common File Save dialog is invoked, requesting the file name that is to be used to store the converted image.

The types of images that can be converted are as follows.

- Bitmap (.**bmp**)
- Tag Image File Format (.**tif**)
- Joint Photographic Experts Group (.**jpg**)
- JPG 2000 (.**jp2**)
- Portable Network Graphics (.**png)**

Use the **PictureType_Bitmap**, **PictureType_ Jpeg**, **PictureType_ Jpeg2000**, **PictureType_Png**, or **PictureType_ Tiff** constant of the **Window** class to specify the picture type.

A 14015 (File does not contain an image type that can be handled) exception is raised if the receiver does not contain valid image data.

Notes Converting to a **.tif** image type file results in a tiff packbits type image. Converting to a **.jpg** image results in loss of quality in the picture, as Joint Photographic Experts Group (JPEG) uses a *lossy* compression algorithm.

As the Portable Network Graphics image uses a *lossless* compression algorithm, it provides clarity and retains definition for images, but the files may be larger than JPEG files.

An exception is raised if this method is invoked from a server method.

You cannot convert images to GIF picture files.

See also the **convertPicture** method of the **Binary** primitive type.

display

Signature `display(): String;`

The **display** method of the **Binary** primitive type returns a string containing a hexadecimal dump of the receiver.

fromANSIToString

Signature `fromANSIToString(lcid: Integer): String;`

The **fromANSIToString** method of the **Binary** primitive type converts the receiver to a string using the code page for the locale specified by the **lcid** parameter and returns the resulting string.

Some code pages (for example, the one used in the People's Republic of China locale) contain multi-byte characters as well as single-byte characters.

fromANSIToStringUtf8

Signature `fromANSIToStringUtf8(lcid: Integer): StringUtf8;`

The **fromANSIToStringUtf8** method of the **Binary** primitive type converts the receiver to a UTF8 string using the code page for the locale specified by the **lcid** parameter and returns the resulting string.

Some code pages (for example, the one used in the People's Republic of China locale) contain multi-byte characters as well as single-byte characters.

length

Signature `length(): Integer;`

The **length** method of the **Binary** primitive type returns the actual length of the value that has been assigned to an embedded Binary property; for example, if you declared a **Binary** property with length of 30 but the value stored is of length 20, the **length** method returns 20.

maxLength

Signature `maxLength(): Integer;`

The **maxLength** method of the **Binary** primitive type returns the declared maximum length of a binary variable. If the binary variable maximum length has not been declared, the value of the **Max_UnboundedLength** global constant in the **SystemLimits** category is returned.

pictureSize

Signature

```
pictureSize(width:  Integer output;
            height: Integer output): Integer;
```

The **pictureSize** method of the **Binary** primitive type returns the type of picture image of the receiver and the width and height of the image. If the binary is not a valid image, zero (**0**) is returned for the type, width, and height of the image.

Note If the image contains multiple icon or cursor definitions, the **pictureSize** method returns the size of the largest of the images.

pictureType

Signature

```
pictureType(): Integer;
```

The **pictureType** method of the **Binary** primitive type returns the type of picture image. The return values are listed in the following table.

Integer	Picture Type	Integer	Picture Type
0	Not a valid picture	5	Cursor
1	Bitmap	6	Tag Image File Format (.**tif**)
2	Not used	7	Joint Photographic Experts Group (.**jpg**)
3	Icon	8	Portable Network Graphics (.**png**)
4	Metafile	9	Graphics Interchange Format (.**gif**)

You can use the **PictureType_Bitmap**, **PictureType_Icon**, **PictureType_MetaFile**, **PictureType_Cursor**, **PictureType_Tiff**, **PictureType_ Jpeg**, **PictureType_Png**, or **PictureType_Gif** constant of the **Window** class, respectively, to specify the picture type.

An exception is raised if this method is invoked from a server method.

posBinary

Signature

```
pos(binary: Binary;
    start:  Integer): Integer;
```

The **posBinary** method of the **Binary** primitive type returns an integer containing the position in the receiver of the binary string specified in the **binary** parameter.

The **start** parameter must be greater than zero (**0**) and less than or equal to the length of the receiver.

Note This method is preferred to the deprecated **Binary** primitive type **pos** method.

posByte

Signature

```
posByte(b:     Byte;
        start: Integer): Integer;
```

The **posByte** method of the **Binary** primitive type returns an integer containing the position in the receiver of the byte specified in the **b** parameter.

The **start** parameter must be greater than zero (**0**) and less than or equal to the length of the receiver.

Note This method is preferred to the deprecated **Binary** primitive type **pos** method.

uncompressToBinary

Signature `uncompressToBinary(): Binary;`

The **uncompressToBinary** method of the **Binary** primitive type returns the uncompressed binary representation of the receiver.

This method uses the **ZLIB** compression routine specified in the **typeAndOption** parameter of the **Binary** primitive type **compressToBinary** method that was used to produce the compressed **Binary** value.

uncompressToString

Signature `uncompressToString(): String;`

The **uncompressToString** method of the **Binary** primitive type returns the uncompressed string representation of the receiver.

This method uses the **ZLIB** compression routine specified in the **typeAndOption** parameter of the **String** primitive type **compressToBinary** method that was used to produce the compressed **Binary** value.

uncompressToStringUtf8

Signature `uncompressToStringUtf8(): StringUtf8;`

The **uncompressToStringUtf8** method of the **Binary** primitive type returns the uncompressed UTF8 string representation of the receiver.

This method uses the **ZLIB** compression routine specified in the **typeAndOption** parameter of the **StringUtf8** primitive type **compressToBinary** method that was used to produce the compressed **Binary** value.

unicodeToAnsi

Signature `unicodeToAnsi(): String;`

The **unicodeToAnsi** method of the **Binary** primitive type interprets the binary as Unicode characters and returns a copy converted to an ANSI string.

This method can be invoked only from an ANSI application.

The code fragment in the following example shows the use of the **unicodeToAnsi** method.

```
if fileIsUnicode then
    str := bin.unicodeToString;
else
    str := bin.ansiToString;
endif;
if str.length >= 3 and str[1:3] = "---" then
    rc := true;
endif;
```

If this method is invoked from a Unicode application, a 1068 - Feature not available in this release exception is raised.

Note Conversion of the binary stops at the first **null** character. If the binary contains embedded nulls, the string returned from the conversion therefore represents only that part of the binary up to the first **null** character.

unicodeToString

Signature `unicodeToString(): String;`

The **unicodeToString** method of the **Binary** primitive type interprets the binary as Unicode characters and returns a copy converted to a string.

The code fragment in the following example shows the use of the **unicodeToString** method.

```
if fileIsUnicode then
    binLen := bin[start : lenLen].unicodeToString.Integer;
else
    binLen := bin[start : lenLen].ansiToString.Integer;
endif;
```

When invoked from a Unicode application, a Unicode string is returned. When invoked from an ANSI application, the copy is converted from Unicode to ANSI, and an ANSI string is returned.

Note When converting from Unicode to ANSI, conversion stops at the first **null** character. If the binary contains embedded nulls, the string returned from the Unicode to ANSI conversion therefore represents only that part of the binary up to the first **null** character.

unpackCString

Signature `unpackCString(start: Integer): String;`

The **unpackCString** method of the **Binary** primitive type returns a string extracted from the binary, starting at the position specified by the **start** parameter and including all characters up to (but not including) the first **null** character.

The code fragment in the following example shows the use of the **unpackCString** method.

```
str := msg.unpackCString(1);
```

If the **null** character is not found, the string consists of all characters from the specified **start** position up to the end of the binary.

Note Unpacking of the string stops at the first **null** character. If the C string contains embedded nulls, the returned string therefore represents only that part of the C string up to the first **null** character.

As the input is assumed to be a binary value of ANSI characters, the returned string is converted to Unicode characters when this method is used in a Unicode JADE system.

uuidAsString

Signature `uuidAsString(): String;`

The **uuidAsString** method of the **Binary** primitive type returns a string formatted as a Universally Unique Identifier (UUID) from the receiver.

To be a valid UUID when calling this method, the binary should be 16 bytes. If it is less than 16 bytes, the value will be internally padded with zero bytes on the end, to make it 16 bytes long before the conversion is performed. If it is longer than 16 bytes, exception 1091 *(Binary too long)* is raised.

The **generateUuid** method of the **Application** class is used to generate a UUID, which has the **Binary** type.

The code fragment in the following example shows the use of the **uuidAsString** method.

```
write self.uuid.uuidAsString;
```

Boolean Type

A **Boolean** primitive type value is one of the logical truth-values represented by the standard JADE identifiers **true** and **false**.

JADE provides standard operators that take Boolean values as operands, and produce a Boolean result. These operators include the logical:

- **and**
- **not** (negation)
- **or** (inclusive)

Boolean values can also be produced by applying relational operators to operands of other types. JADE provides the standard relational operators listed in the following table.

Operator	Description
=	Equal to
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

A relational operation consists of two operands separated by a relational operator. If the relation is satisfied, it has the value **true**. If the relation is not satisfied, it has the value **false**. The result of a relational operation is therefore a Boolean value.

The following example shows the use of the **Boolean** primitive type.

```
isMarried(): Boolean;
begin
    return spouse <> null;
end;
```

For details about the method defined in the **Boolean** primitive type, see "Boolean Method", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Boolean Method

The method defined in the **Boolean** primitive type is summarized in the following table.

Method	Returns …
display	A string representing the value of the receiver

display

Signature `display(): String;`

The **display** method of the **Boolean** primitive type returns a string containing **"true"** or **"false"**, depending on the value of the receiver.

Byte Type

A **Byte** primitive type value stores an unsigned integer value in the range **0** through **255**.

JADE defines a number of arithmetic operations that take **Integer** operands and return **Integer** results, as listed in the following table. A **Byte** value can be used in place of an **Integer** value as an operand because of the implicit type conversion that takes place before the expression is evaluated. The result of an arithmetic operation involving **Byte** values is an **Integer**.

Operator	Description
+	Add
-	Subtract
*	Multiply
div	Integer division (division with truncation; for example, **7 div 3 = 2**)
^	Exponentiation (for example, **i ^ 3** is i cubed)
mod	Modulus (remainder after integer division)

These are binary (or dyadic) infix operators; that is, they are used with operands on both sides of the operator (for example, **a+b**). However, the **+** operator and **-** operator are also used as unary (or monadic) prefix operators, as listed in the following table.

Unary Prefix Operator	Description
+a	Sign identify
-a	Sign inversion

Using Byte Types in Assignments

An exception is raised when you attempt to compile an assignment of a numeric value (**Decimal**, **Integer**, **Integer64**, or **Real**) to a **Byte** variable without an explicit type cast, as shown in the following example.

```
vars
    byt : Byte;
    int : Integer;
begin
    byt := 123;                 // Not allowed to assign to a literal value
    int := 123;
    byt := int;                 // Not allowed to assign to an Integer,
                                // or other numeric type
    byt := 123.Byte;            // Allowed with the explicit type caste
    byt := int.Byte;            // Allowed with the explicit type caste
end;
```

A runtime exception is raised if the value assigned to a **Byte** variable is less than **0** or greater than **255**, as shown in the following example.

```
vars
    byt : Byte;
begin
```

```
    byt = (-64).Byte;           // Exception raised at run time
end;
```

A **Byte** value can be assigned to a variable of any of the numeric types (**Integer**, **Integer64**, **Real**, **Decimal**) without an explicit type caste.

For details about the methods defined in the **Byte** primitive type, see "Byte Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Byte Methods

The methods defined in the **Byte** primitive type are summarized in the following table.

Method	Description
bitAnd	Returns a **Byte** value representing the receiver bits ANDed with the argument
bitNot	Returns a **Byte** value whose bit values are the inverse of the bit values of the receiver
bitOr	Returns a **Byte** value representing the receiver bits ORed with the argument
bitXor	Returns a **Byte** value representing the receiver bits XORed with the argument
display	Returns a string representing the value of the receiver
isEven	Returns **true** if the receiver represents an even number; otherwise **false**
isOdd	Returns **true** if the receiver represents an odd number; otherwise **false**
max	Returns the larger value of the receiver and a specified **Byte**
min	Returns the lesser value of the receiver and a specified **Byte**
numberFormat	Returns a string in the number format of the current locale
padLeadingWith	Returns the receiver as a string padded to the specified length with a leading character
parseCurrencyWithCurrentLocale	Sets the receiver to the result of parsing a string representing a currency value for the current locale
parseCurrencyWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale
parseNumberWithCurrentLocale	Sets the receiver to the result of parsing a string representing a number for the current locale
parseNumberWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale
userCurrencyFormat	Returns the receiver as a string in the specified currency format for the current locale
userCurrencyFormatAndLcid	Returns the receiver as a string in the specified currency format for the specified locale
userNumberFormat	Returns the receiver as a string in the specified number format for the current locale

Method	Description
userNumberFormatAndLcid	Returns the receiver as a string in the specified number format for the specified locale

bitAnd

Signature `bitAnd(op: Byte): Byte;`

The **bitAnd** method of the **Byte** primitive type compares each bit in the receiver with the corresponding bit specified in the **op** parameter and returns a **Byte** value representing the receiver bits ANDed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
Both bits are 1	1
One or both bits are not 1	0

The following example shows the use of the **bitAnd** method.

```
keyDown(keyCode: Integer io; shift: Integer) updating;
constants
    Shift = 1.Byte;
    Cntrl = 2.Byte;
    Alt   = 4.Byte;
vars
    byt : Byte;
begin
    byt := shift.Byte;
    if byt.bitAnd(Shift) <> 0 then write "Shift key is down"; endif;
    if byt.bitAnd(Cntrl) <> 0 then write "Control key is down"; endif;
    if byt.bitAnd(Alt)   <> 0 then write "Alt key is down"; endif;
end;
```

bitNot

Signature `bitNot(): Byte;`

The **bitNot** method of the **Byte** primitive type returns a **Byte** value whose bit values are the inverse of the bit values of the receiver. The generated return values are listed in the following table.

Bits in Receiver	Corresponding Bit in Return Value
Bit is not 1	1
Bit is 1	0

bitOr

Signature `bitOr(op: Byte): Byte;`

The **bitOr** method of the **Byte** primitive type compares each bit in the receiver with the corresponding bit specified in the **op** parameter, and returns a **Byte** value representing the receiver bits ORed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
One or both bits are 1	1
Neither bit is 1	0

The code fragment in the following example shows the use of the **bitOr** method.

```
constants
    BitFlagNone = 0.Byte;
    BitFlag1    = 1.Byte;
vars
    byt : Byte;
begin
    byt := BitFlagNone;
    // set bit flag 1
    byt := byt.bitOr(BitFlag1);
    // test that bit flag 1 is set
    if byt.bitAnd(BitFlag1) <> 0 then
        write "flag 1 is set";
    endif;
end;
```

bitXor

Signature `bitXor(op: Byte): Byte;`

The **bitXor** method of the **Byte** primitive type compares each bit in the receiver with the corresponding bit specified in the **op** parameter, and returns a **Byte** value representing the receiver bits XORed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
The bits are complementary	1
The bits are not complementary	0

display

Signature `display(): String;`

The **display** method of the **Byte** primitive type returns a string that represents the integral value of the receiver.

isEven

Signature `isEven(): Boolean;`

The **isEven** method of the **Byte** primitive type returns **true** if the receiver represents an even number; otherwise, it returns **false**.

isOdd

Signature `isOdd(): Boolean;`

The **isOdd** method of the **Byte** primitive type returns **true** if the receiver represents an odd number; otherwise, it returns **false**.

max

Signature `max(byte: Byte): Byte;`

The **max** method of the **Byte** primitive type returns the larger value of the receiver and the value of the **byte** parameter. If the value of the receiver is greater than the value of the **byte** parameter, the value of the receiver is returned. If the value of the receiver is less than or equal to the value of the **byte** parameter, the value of **byte** is returned.

min

Signature `min(byte: Byte): Byte;`

The **min** method of the **Byte** primitive type returns the lesser value of the receiver and the value of the **byte** parameter. If the value of the receiver is less than the value of the **byte** parameter, the value of the receiver is returned. If the value of the receiver is greater than or equal to the value of the **byte** parameter, the value of **byte** is returned.

numberFormat

Signature `numberFormat(): String;`

The **numberFormat** method of the **Byte** primitive type returns a string in the numeric format defined for the current locale, which specifies the thousands separator, sign character, and decimal point character; for example, **129.00**. The following example shows the use of the **numberFormat** method.

```
vars
    str : String;
    byt : Byte;
begin
    byt := 167.Byte;
    write byt;                  // Outputs 167
    str := byt.numberFormat;
    write str;                  // Outputs 167.00
end;
```

You can use the **defineNumberFormat** method of the **NumberFormat** class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

padLeadingWith

Signature
```
padLeadingWith(char: Character;
               max:  Integer): String;
```

The **padLeadingWith** method of the **Byte** primitive type returns a string of the length specified in the **max** parameter, consisting of the receiving string padded with the leading character specified in the **char** parameter.

If the string is equal to or longer than the value specified in the **max** parameter, it is not truncated but the whole string is returned.

The following example shows the use of the **padLeadingWith** method.

```
constants
    PAD_CHARACTER = 'x';
vars
    byt : Byte;
    str : String;
begin
    byt := 123.Byte;
    str := byt.padLeadingWith('w', 15) & ' 678 Sesame St.';
    write str; // Outputs wwwwwwwwwwwww123 678 Sesame St.
    str := byt.padLeadingWith('a', 2);
    write str; // Outputs 123
    str := byt.padLeadingWith(PAD_CHARACTER, 10);
    write str; // Outputs xxxxxxx123
end;
```

parseCurrencyWithCurrentLocale

Signature
```
parseCurrencyWithCurrentLocale(source:    String;
                               errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithCurrentLocale** method of the **Byte** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Byte** format of the current locale for currency sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

This is equivalent to calling the **parseCurrencyWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseCurrencyWithFmtAndLcid

Signature

```
parseCurrencyWithFmtAndLcid(source:    String;
                            fmt:       CurrencyFormat;
                            lcid:      Integer;
                            errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithFmtAndLcid** method of the **Byte** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for currency sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

The currency character sequence is optional but if it is included in the source, it must be correctly positioned.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency character sequence is optional.

If native digits are allowed and the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the **source** parameter text can include a decimal point and decimal digits, but they must all be zero (**0**) so that rounding or truncation is not required to store the value in a **Byte** variable; for example:

- **"100"**, **"100."**, **"100.0"**, **"100.00"**, and **"100.000"** are accepted as valid and equal.
- **"100.01"** and **"100.99"** are rejected, as the value cannot be stored accurately in a **Byte** primitive type.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithCurrentLocale

Signature

```
parseNumberWithCurrentLocale(source:    String;
                             errOffset: Integer output): Integer updating;
```

The **parseNumberWithCurrentLocale** method of the **Byte** primitive parses the string specified in the **source** parameter to ensure that it matches the **Byte** format of the current locale for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

This is equivalent to calling the **parseNumberWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithFmtAndLcid

Signature

```
parseNumberWithFmtAndLcid(source: String;
                          fmt:    NumberFormat;
                          lcid:   Integer;
                          errOffset: Integer output): Integer updating;
```

The **parseNumberWithFmtAndLcid** method of the **Byte** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

The sign character sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the **source** parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in a **Byte** variable; for example:

- **"100"**, **"100."**, **"100.0"**, **"100.00"**, and **"100.000"** are accepted as valid and equal.
- **"100.01"** and **"100.99"** are rejected, as the value cannot be stored accurately in a **Byte** primitive type.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userCurrencyFormat

Signature

```
userCurrencyFormat(fmt: CurrencyFormat): String;
```

The **userCurrencyFormat** method of the **Byte** primitive type returns a string containing the receiver in the currency format specified in the **fmt** parameter.

To define your currency formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user currency format name with a dollar sign (**$**); for example, **userCurrencyFormat($MyCurrency)**.

You can use the **defineCurrencyFormat** method of the **CurrencyFormat** class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userCurrencyFormatAndLcid

Signature

```
userCurrencyFormatAndLcid(fmt:  CurrencyFormat;
                          lcid: Integer): String;
```

The **userCurrencyFormatAndLcid** method of the **Byte** primitive type returns a string containing the receiver in the currency format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature

```
userNumberFormat(fmt: NumberFormat): String;
```

The **userNumberFormat** method of the **Byte** primitive type returns a string containing the receiver in the number format specified in the **fmt** parameter.

To define your numeric formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user number format name with a dollar sign (**$**); for example, **userNumberFormat($MyNumber)**.

You can use the **defineNumberFormat** method from the **NumberFormat** class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormatAndLcid

Signature

```
userNumberFormat(fmt:  NumberFormat;
                 lcid: Integer): String;
```

The **userNumberFormatAndLcid** method of the **Byte** primitive type returns a string containing the receiver in the number format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Character Type

Use the **Character** primitive type to define a variable as a single ANSI or Unicode character. The following example shows the use of the **Character** primitive type.

```
testCharacter();
vars
    charValue : Character;
begin
    charValue := "M";              // Defines the variable value
    write charValue;               // Outputs a value of M
    write charValue.isLower;       // Outputs a value of false
    write charValue.isAlpha;       // Outputs a value of true
    write charValue.toLower;       // Outputs a value of m
end;
```

For details about the methods defined in the **Character** primitive type, see "Character Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Character Methods

The methods defined in the **Character** primitive type are summarized in the following table.

Method	Returns …
compareEql	**true** if the receiver is equal to a specified character
compareGeneric	An integer showing whether the receiver is greater than, equal to, or less than a specified character
compareGeq	**true** if the receiver is greater than or equal to a specified character
compareGtr	**true** if the receiver is greater than a specified character
compareLeq	**true** if the receiver is less than or equal to a specified character
compareLss	**true** if the receiver is less than the value of a specified character
compareNeq	**true** if the receiver is not equal to a specified character
display	A string containing the receiver
isAlpha	**true** if the receiver represents a letter for the current language setting
isDelimiter	**true** if the receiver is not alphanumeric
isHex	**true** if the receiver represents a hexadecimal character
isLower	**true** if the receiver represents a lowercase letter
isNumeric	**true** if the receiver represents a numeric digit
isPrintable	**true** if the receiver is a character that can be printed
isUpper	**true** if the receiver represents an uppercase letter
makeString	A string of the specified length filled with the value of the receiver
setByteOrderLocal	A character that has the bytes ordered as required by the local node

Method	Returns …
setByteOrderRemote	A character that has the bytes ordered as required by the specified remote node
toHex	A string containing the hexadecimal value of the receiver
toLower	The lowercase equivalent of the receiving character
toUpper	The uppercase equivalent of the receiving character

compareEql

Signature

```
compareEql(rhs:          Character;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareEql** method of the **Character** primitive type returns **true** if the receiver is equal to the value of the **rhs** parameter; otherwise it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareEql(lhs, true, false, null);

recv.toLower = lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareEql** method.

```
write "A".compareEql("a", true, false, null);          // Outputs true
write "A".compareEql("a", false, false, null);         // Outputs false
```

compareGeneric

Signature

```
compareGeneric(rhs:          Character;
               bIgnoreCase:  Boolean;
               bUseLocale:   Boolean;
               locale:       Locale): Integer;
```

The **compareGeneric** method of the **Character** primitive type compares the receiver with the value of the **rhs** parameter and returns one of the following values.

Value	Returned if the receiver is …
Negative integer	Less than the right-hand side value represented by the **rhs** parameter
Zero (**0**)	Equal to the right-hand side value represented by the **rhs** parameter
Positive integer	Greater than the right-hand side value represented by the **rhs** parameter

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operators (**<**, **<=**, **=**, **>=**, **>**, **<>**), documented in Chapter 1 of the *JADE Developer's Reference*, use a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareGeneric(lhs, true, false, null);

(recv.toLower>lhs.toLower).Integer - (recv.toLower<lhs.toLower).Integer;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareGeneric** method.

```
vars
    locale : Locale;
begin
    write "a".compareGeneric("c", false, false, null);         // Outputs -1
    write "b".compareGeneric("b", false, false, null);         // Outputs 0
    write "c".compareGeneric("a", false, false, null);         // Outputs 1
```

```
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
    write "à".compareGeneric("z", false, false, null);         // Outputs 1
    write "à".compareGeneric("z", false, true, locale);        // Outputs -1
```

compareGeq

Signature

```
compareGeq(rhs:          Character;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareGeq** method of the **Character** primitive type returns **true** if the receiver is greater than or equal to the value of the **rhs** parameter; otherwise it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**>=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareGeq(lhs, true, false, null);

recv.toLower >= lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareGeq** method.

```
vars
    locale : Locale;
begin
    write "a".compareGeq("c", false, false, null);          // Outputs false
    write "b".compareGeq("b", false, false, null);          // Outputs true
    write "c".compareGeq("a", false, false, null);          // Outputs true
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
```

```
    write "à".compareGeq("z", false, false, null);          // Outputs true
    write "à".compareGeq("z", false, true, locale);         // Outputs false
```

compareGtr

Signature

```
compareGtr(rhs:          Character;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareGtr** method of the **Character** primitive type returns **true** if the receiver is greater than the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**>**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareGtr(lhs, true, false, null);

recv.toLower > lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareGtr** method.

```
vars
    locale : Locale;
begin
    write "a".compareGtr("c", false, false, null);          // Outputs false
    write "b".compareGtr("b", false, false, null);          // Outputs false
    write "c".compareGtr("a", false, false, null);          // Outputs true
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
    write "à".compareGtr("z", false, false, null);          // Outputs true
    write "à".compareGtr("z", false, true, locale);         // Outputs false
```

compareLeq

Signature

```
compareLeq(rhs:         Character;
           bIgnoreCase: Boolean;
           bUseLocale:  Boolean;
           locale:      Locale): Boolean;
```

The **compareLeq** method of the **Character** primitive type returns **true** if the receiver is less than or equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareLeq(lhs, true, false, null);

recv.toLower <= lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareLeq** method.

```
vars
    locale : Locale;
begin
    write "a".compareLeq("c", false, false, null);          // Outputs true
    write "b".compareLeq("b", false, false, null);          // Outputs true
    write "c".compareLeq("a", false, false, null);          // Outputs false
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
    write "à".compareLeq("z", false, false, null);          // Outputs false
    write "à".compareLeq("z", false, true, locale);         // Outputs true
```

compareLss

Signature

```
compareLss(rhs:         Character;
           bIgnoreCase: Boolean;
           bUseLocale:  Boolean;
           locale:      Locale): Boolean;
```

The **compareLss** method of the **Character** primitive type returns **true** if the receiver is less than the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareLss(lhs, true, false, null);

recv.toLower < lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareLss** method.

```
vars
    locale : Locale;
begin
    write "a".compareLss("c", false, false, null);          // Outputs true
    write "b".compareLss("b", false, false, null);          // Outputs false
    write "c".compareLss("a", false, false, null);          // Outputs false
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
    write "à".compareLss("z", false, false, null);          // Outputs false
    write "à".compareLss("z", false, true, locale);         // Outputs true
```

compareNeq

Signature

```
compareNeq(rhs:         Character;
           bIgnoreCase: Boolean;
           bUseLocale:  Boolean;
           locale:      Locale): Boolean;
```

The **compareNeq** method of the **Character** primitive type returns **true** if the receiver is not equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<>**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareNeq(lhs, true, false, null);

recv.toLower <> lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareNeq** method.

```
write "A".compareNeq("a", true, false, null);          // Outputs false
write "A".compareNeq("a", false, false, null);         // Outputs true
```

display

Signature

```
display(): String;
```

If the receiver is a printable character, the **display** method of the **Character** primitive type returns a string containing the receiving character if it is a printable character for the current language setting.

If the receiver is not a printable character, the **display** method returns a string containing the hexadecimal value of the receiving character, enclosed in single quotation marks (**'**) and preceded by a number sign (**#**).

The following example shows the use of the **display** method.

```
vars
    char1, char2, char3 : Character;
begin
    char1 := "q";
    char2 := #'13';
    char3 := #'41';
    write char1.display;     // Outputs q
    write char2.display;     // Outputs #'13'
    write char3.display;     // Outputs A
end;
```

isAlpha

Signature `isAlpha(): Boolean;`

The **isAlpha** method of the **Character** primitive type returns **true** if the receiver represents a letter for the current language setting; otherwise, it returns **false**.

The code fragment in the following example shows the use of the **isAlpha** method.

```
//strip leading non-alpha characters
count := 1;
alphaFound := false;
while count <= str.length do
    if str[count].Character.isAlpha then
        if newStr.length > 0 then
            newStr := newStr & str[count].toUpper;
        else
            newStr := str[count].toUpper;
        endif;
        count := count + 1;
        alphaFound := true;
        break;
    endif;
    count := count + 1;
endwhile;
if alphaFound = false and newStr.length = 0  then
    return "";
endif;
```

isDelimiter

Signature `isDelimiter(): Boolean;`

The **isDelimiter** method of the **Character** primitive type returns **true** if the receiver is not alphanumeric; otherwise, it returns **false**.

The code fragment in the following example shows the use of the **isDelimiter** method.

```
// find delimiter
count := int;
while count < self.length do
    charValue := self[count];
    if charValue.isDelimiter and charValue <> '_' then
```

```
            break;
        else
            count := count + 1;
        endif;
    endwhile;
```

isHex

Signature `isHex(): Boolean;`

The **isHex** method of the **Character** primitive type returns **true** if the receiver represents a hexadecimal character; that is, **0** through **9**, **a** through **f**, or **A** through **F**.

isLower

Signature `isLower(): Boolean;`

The **isLower** method of the **Character** primitive type returns **true** if the receiver represents a lowercase letter for the current language setting.

The code fragment in the following example shows the use of the **isLower** method.

```
if count <= str.length and str[count].Character.isLower then
    str[count] := str[count].Character.toUpper;
endif;
```

isNumeric

Signature `isNumeric(): Boolean;`

The **isNumeric** method of the **Character** primitive type returns **true** if the receiver represents a numeric digit for the current language setting.

The code fragment in the following example shows the use of the **isNumeric** method.

```
if prodTitle.text.length = 0 then
    // Build default caption from name by inserting a space before each
    // uppercase character; for example, "3NeonHuedChocolateCoveredGumDrops"
    // becomes "3 Neon Hued Chocolate Covered Gum Drops"
    str := prodName.text;
    prodTitle.text[1] := str[1];
    count := 2;
    while count <= str.length do
        if str[count].isUpper or (str[count].isNumeric and not
                      str[count-1].isNumeric) then
            prodTitle.text := prodTitle.text & " ";
        endif;
        prodTitle.text := prodTitle.text & str[i];
        count := count + 1;
    endwhile;
endif;
```

isPrintable

Signature `isPrintable(): Boolean;`

The **isPrintable** method of the **Character** primitive type returns **true** if the receiver is a character that can be printed; otherwise, it returns **false**.

Character values in the range #20 through #7E can always be printed. Whether a character greater than #7F can be printed depends on the current locale, and character set (ANSI or Unicode).

isUpper

Signature `isUpper(): Boolean;`

The **isUpper** method of the **Character** primitive type returns **true** if the receiver represents an uppercase letter for the current language setting.

The code fragment in the following example shows the use of the **isUpper** method.

```
// Check the first character is an uppercase letter
if not theName[1].Character.isUpper then
    app.beep;
    statusLine.caption := "Error - The name must begin with an uppercase
                           letter";
    stringName.setFocus;
    return;
endif;
```

makeString

Signature `makeString(length: Integer): String;`

The **makeString** method of the **Character** primitive type returns a string of the length specified in the **length** parameter filled with the value of the receiver.

If the receiver is null, the returned string is filled with spaces. If the value of the **length** parameter is less than or equal to zero (**0**), an empty string is returned.

The following example shows the use of the **makeString** method.

```
vars
    charValue : Character;
begin
    charValue := "*";
    write charValue.makeString(10);    // Outputs **********
    charValue := " ";
    write charValue.makeString(10);    // Outputs           (ten spaces)
end;
```

setByteOrderLocal

Signature `setByteOrderLocal(architecture: Integer): Character;`

The **setByteOrderLocal** method of the **Character** primitive type returns a character that has the bytes ordered as required by the local node.

The bytes of the receiver are assumed to be ordered as indicated by the **architecture** parameter. The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature `setByteOrderRemote(architecture: Integer): Character;`

The **setByteOrderRemote** method of the **Character** primitive type returns a character that has the bytes ordered as required by the remote node indicated by the **architecture** parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

toHex

Signature `toHex(): String;`

The **toHex** method of the **Character** primitive type returns a string containing the hexadecimal value of the receiving character.

The following example shows the use of the **toHex** method.

```
vars
    charValue : Character;
begin
    charValue := "q";
    write charValue.toHex;    // Outputs 71
end;
```

toLower

Signature `toLower(): Character;`

The **toLower** method of the **Character** primitive type returns the lowercase equivalent of the receiving character. Any character that is not an uppercase character is left unchanged.

The following example shows the use of the **toLower** method.

```
vars
    charValue : Character;
begin
    charValue := "A";
    write charValue.toLower;   // Outputs a
end;
```

toUpper

Signature `toUpper(): Character;`

The **toUpper** method of the **Character** primitive type returns the uppercase equivalent of the receiving character. Any character that is not a lowercase character is left unchanged.

The following example shows the use of the **toUpper** method.

```
vars
    charValue : Character;
begin
    charValue := "r";
    write charValue.toUpper;   // Outputs R
end;
```

Date Type

A **Date** variable represents a specific day since the start of the Julian period. The **Date** primitive type defines the protocol for comparing and manipulating dates.

In JADE thin client mode, local variables of type **Date** are always initialized to the date relative to the presentation client.

Dates are generally obtained from user input in a Gregorian Calendar format, and converted to the internal (Julian day) format for internal storage and computation. Strictly speaking, the valid range for Julian day numbers is **0** through **2914726**, which are the limits of the current Julian Period (7980 Julian years in length). This supports a valid date range from 24 November -4713 through 23 February 3268, Gregorian. However, JADE does not adhere to this limit, and allows day numbers to extend beyond this range.

JADE handles any calendar available within Microsoft Windows, including the conversion and display of non-Gregorian calendar dates, based on the locale and calendar set for the current user.

Notes JADE prevents the use of the day **0** as a valid date, since the Julian day **0** is considered to be **null**. The maximum Julian day number is not imposed as the maximum date permitted by JADE.

```
write (1).Date // displays 25 November -4713 (one day higher than Day zero)

write (0).Date // displays null (but internally represents 24 November -4713)
```

The variable contains the current date as a Julian day number. (For details, see "Historical Note about the Date Type", in the following subsection.)

When used as a **Dictionary** key, the valid range for a **Date** key is **(0).Date** through **(Max_Integer).Date**.

If you declare a **Date** primitive type local variable in your method, it is initialized with the current date each time the method that declares the variable is invoked. Object attributes of type **Date** are initialized to **null**.

As **Date** primitive types are ordinal values (Julian day numbers), the forms of date arithmetic expressions listed in the following table are valid.

Expression	Expression Type
date-expression + integer-expression	(date)
date-expression - integer-expression	(date)
date-expression - date-expression	(integer)

The following assignment shows the calculation of a date 134 days later than a specified date:

```
date2 := date1+134;
```

You can use the **JadeEditMask** class **getTextAsDate** and **setTextFromDate** methods to handle locale formatting for date fields.

For details about the methods defined in the **Date** primitive type, see "Date Methods", and for examples of the use of this primitive type, see "Date Primitive Type Examples", later in this section. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Historical Note about the Date Type

The first official Gregorian calendar day occurred on the date of its inception on 15 October 1582 (Gregorian). This reform was later adopted by most western cultures at different times. We can still identify particular days before 15 October 1582 (Gregorian) using dates in the Gregorian calendar, simply by projecting the Gregorian dating system back beyond the time of its implementation. A calendar obtained by extension earlier in time than its invention or implementation is called the *proleptic* version of the calendar, and we therefore obtain the Proleptic Gregorian Calendar.

The Proleptic Gregorian Calendar has a year 0, and there are no years *Before Christ* (BC). The year before 1 *Anno Domini* (AD) is 0, and the year before that is -1.

The Julian day number of the **Date** primitive type is the number of elapsed days since the start of the Julian period, as defined by Joseph Scaliger. The start of the Julian period, established by Scaliger, is 1 January 4713 *Before Common Era* Proleptic Julian (BCE). The Julian period is the universal cycle (or period) used in chronology, especially for astronomical calculations involving large time intervals.

Date Primitive Type Examples

You can create a primitive method in type **Date** to return a string containing the short date in *dd-MM-yyyy* format, as follows.

```
testShortDate1(): String;
begin
    return day.String & "-" & month.String & "-" & year.String;
end;
```

In this example, the **day**, **month**, and **year** values are methods of the **Date** primitive type. However, the following example shows a more-direct method of returning the date in a short format; that is, the **format** method enables you to format dates to meet your requirements.

```
testShortDate2(): String;
vars
    date : Date;
begin
    return date.format("dd-MM-yyyy");
end;
```

Notes The month is denoted by uppercase letters (that is, *MM*) to differentiate it from minutes (that is, *mm*).

You can use the **defineLongDateFormat** or **defineShortDateFormat** method of the **DateFormat** class if you want to create your own transient format objects and define a long or short date format that dynamically overrides the appropriate format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If an attempt is made to format an invalid date, **"*invalid*"** is returned.

The following example shows the use of the **Date** primitive type.

```
daysToXmas(): Integer;     // valid only before xmas day in a specified year
vars
    xmasday, today : Date;
begin
    xmasday.setDate(25, 12, today.year);
    return xmasday - today;
end;
```

Date Methods

The methods defined in the **Date** primitive type are summarized in the following table.

Method	Description
day	Returns the day of the month as an integer
dayName	Returns the name of the day of the week as a string
dayNameWithLcid	Returns the name of the day of the week as a string for the specified locale
dayOfWeek	Returns an integer value for the day of the week
dayOfYear	Returns an integer value for the day of the year
display	Returns the date as a string
format	Returns the date as a string in the specified format
isFormatable	Returns **true** if the date falls within the valid conversion range for the execution platform
isLeapYear	Returns **true** if the year is a leap year
isValid	Returns **true** if the date is valid
longFormat	Returns the date as a string in the long date format
month	Returns the month as an integer
monthName	Returns the name of the month
monthNameWithLcid	Returns the name of the month for the specified locale
parseForCurrentLocale	Sets the receiver to the result of parsing a string representing a date for the current locale
parseLongWithCurrentLocale	Sets the receiver to the result of parsing a string representing a date in the long date format for the current locale
parseLongWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a date for the specified long date format and the specified locale
parseLongWithPicAndLcid	Sets the receiver to the result of parsing a string representing a date for the specified long date picture and the specified locale
parseShortWithCurrentLocale	Sets the receiver to the result of parsing a string representing a date in the short date format for the current locale
parseShortWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a date for the specified short date format and the specified locale
parseShortWithPicAndLcid	Sets the receiver to the result of parsing a string representing a date for the specified short date picture and the specified locale
setByteOrderLocal	Returns a date that has the bytes ordered as required by the local node
setByteOrderRemote	Returns a date that has the bytes ordered as required by the specified remote node
setDate	Sets the receiver to a specified date
setDateYearAbsolute	Sets the receiver to a specific absolute date

Method	Description
shortDayNameWithLcid	Returns the short name of the week day for the specified locale
shortFormat	Returns the date in the short date format
shortMonthNameWithLcid	Returns the short name of the month for the specified locale
subtract	Returns the interval between the receiver and the specified date
userFormat	Returns the date as a string in the specified date format
userLongFormatAndLcid	Returns the date as a string in the specified long date format for the specified locale
userLongFormatPicAndLcid	Returns the date as a string formatted in the specified long date picture for the specified locale
userShortFormatAndLcid	Returns the date as a string in the specified short date format for the specified locale
userShortFormatPicAndLcid	Returns the date as a string formatted in the specified short date picture for the specified locale
year	Returns the year as in integer

day

Signature `day(): Integer;`

The **day** method of the **Date** primitive type returns the day of the month (represented by the date value of the receiver) as an integer; for example, **21**.

The following code fragment shows the use of the **day** method.

```
while day.day <= Calendar.DaysOfWeek do
    column := day.dayOfWeek;
    text   := day.dayName[1:Calendar.DayNameLength];
    day    := day + 1;
endwhile;
display := date.day.String & "/" & date.month.String & "/" & date.year.String[3:2];
```

dayName

Signature `dayName(): String;`

The **dayName** method is defined by the current locale of the user. The **dayName** method of the **Date** primitive type returns the name of the day of the week (represented by the date value of the receiver) as a string; for example, **Tuesday**.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

For an example of the use of the **dayName** method, see the **Date** primitive type **day** method.

dayNameWithLcid

Signature `dayNameWithLcid(lcid: Integer): String;`

The **dayNameWithLcid** method of the **Date** primitive type returns a string containing the full name of the week day from the locale specified in the **lcid** parameter for the receiver date.

If the value of the **lcid** parameter is zero (**0**), the day name is obtained from the current locale. If the date is null or invalid, an exception is raised.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

dayOfWeek

Signature `dayOfWeek(): Integer;`

The **dayOfWeek** method of the **Date** primitive type returns an integer value for the day of the week (for the date value of the receiver).

For an example of the use of the **dayOfWeek** method, see the **Date** primitive type day method.

The week day values are those listed in the following table.

Integer Value	Day
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

dayOfYear

Signature `dayOfYear(): Integer;`

The **dayOfYear** method of the **Date** primitive type returns an integer value for the day of the year (for the date value of the receiver); for example, **274**.

The following example shows the use of the **dayOfYear** method.

```
vars
    date : Date;
begin
    date := "01 January 2000".Date;
    write date.dayOfYear.String;     // Outputs 1
    date := "10 March 2000".Date;
    write date.dayOfYear.String;     // Outputs 70
end;
```

display

Signature `display(): String;`

The **display** method of the **Date** primitive type returns the receiver as a string.

format

Signature `format(picture: String): String;`

The **format** method of the **Date** primitive type returns the receiver as a string formatted in the date format specified in the **picture** parameter.

Notes If the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is set to its default value of **false**, the **format** method returns ***invalid*** for dates outside the range **1 January 1601** through **31 December 30827**.

If **EnhancedLocaleSupport** is set to **true**, the **format** method can format dates in the range **1 January 100** to **31 December 30827** correctly.

If **EnhancedLocaleSupport** is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The following examples show the use of the **format** method.

```
vars
    date : Date;
begin
    write "The date today is " & date.format("dd.MM.yyyy");
end;

if cd.lastPlayed = null then
    form.lastPlayed.caption := "Never";
else
    form.lastPlayed.caption := cd.lastPlayed.format("dd-MMM-yy");
endif;
```

The **picture** parameter is the string value of the required format. The month is denoted by uppercase letters (that is, *MM*) to differentiate it from minutes (that is, *mm*).

Use the string picture elements listed in the following table to construct date format picture strings. Separate each element with a space or separator character.

Picture	Description	Output Format
d	Day of month as digits, with no leading zero	9
dd	Day of month as digits, with a leading zero	09
ddd	Day of week as an abbreviation as specified in the locale definition, usually three letters	Mon
dddd	Day of week as the full name	Wednesday
M	Month as digits, with no leading zero	6
MM	Month as digits, with leading zero	08
MMM	Month as an abbreviation as specified in the locale definition, usually three letters	Mar
MMMM	Month as the full name	September
y	Year, represented by only the last digit if less than 10, else *yy*	6
yy	Year, represented by only the last two digits	97
yyy	Year, represented by all significant digits	1998

For example, to return a date string of **Wed Aug 04 99**, use the following picture string as your **picture** parameter for the **format** method of the **Date** primitive type.

```
ddd MMM dd yy
```

Note The locale information for each country is supplied embedded with the operating system. For example, the ***MMM*** format of an English locale returns the month as a three-letter abbreviation (for example, **Apr**) and the ***ddd*** format as a three-letter abbreviation (for example, **Mon**). The ***MMM*** format of a French locale can vary from three letters to four letters and a character (for example, **mai**, **mars**, or **janv.**) and the ***ddd*** format always returns three letters and a character (for example, **lun.**).

isFormatable

Signature `isFormatable(): Boolean;`

The **isFormatable** method of the **Date** primitive type returns **true** for dates that can be formatted correctly by using the **shortFormat** and **longFormat** methods of the **Date** primitive type; otherwise it returns **false**.

The formatting methods in JADE depend on the operating system date conversion routines, which result in the **isFormatable** method returning **true** for dates in the range 1 January 1601 through 31 December 30827.

isLeapYear

Signature `isLeapYear(): Boolean;`

The **isLeapYear** method of the **Date** primitive type returns **true** if the year of the locale and calendar that is set for the current user is a leap year.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **Date** primitive type returns **true** if the receiver is a valid date. This method returns **false** if the date is invalid.

This method also returns **false** for dates outside the valid internal representation range of 24th November -4713 through 31st December 1465072 Gregorian.

The code fragment in the following example shows the use of the **isValid** method.

```
if (day.String & "/" & month.String & "/" & year.String).Date.isValid then
    calendar.date.setDate(day, month, year);
else
    calendar.date.setDate(1, month, year);
endif;
```

longFormat

Signature `longFormat(): String;`

The **longFormat** method of the **Date** primitive type is defined by the current locale of the user.

The **longFormat** method returns the receiver as a string formatted in the long date format defined for the current locale of the user; for example, **Wednesday, 04 08, 99** or **Wed, August 4, 1999**.

Notes You can use the **defineLongDateFormat** method of the **DateFormat** class if you want to create your own transient format objects and define a long date format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

The **longFormat** method returns **"*invalid*"** for dates before 1601 or after 30827.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

month

Signature `month(): Integer;`

The **month** method of the **Date** primitive type returns the month (represented by the date value of the receiver) as an integer; for example, **08**.

The code fragment in the following example shows the use of the **month** method.

```
if date <> null then
    if date.month <> value.month or date.year <> value.year then
        monthTable.showDays(value);
    endif;
endif;
```

monthName

Signature `monthName(): String;`

The **monthName** method of the **Date** primitive type returns the name of the month (for the date value of the receiver) as a string.

The following example shows the use of the **monthName** method.

```
date(set: Boolean; value: Date io) mapping, updating;
begin
    if set and isTransient and date <> value then
        if date <> null then
            if date.month <> value.month or date.year <> value.year then
                monthTable.showDays(value);
            endif;
        endif;
        monthTable.selectDay(value);
        monthLabel.caption := value.monthName;
        yearLabel.caption := value.year.String;
        date := value;
        if changeType = ChangeType_None then
            changeType := ChangeType_Script;
        endif;
        click(self);
        changeType := ChangeType_None;
    endif;
end;
```

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

monthNameWithLcid

Signature `monthNameWithLcid(lcid: Integer): String;`

The **monthNameWithLcid** method of the **Date** primitive type returns a string containing the full name of the month from the locale specified in the **lcid** parameter for the receiver date.

If the value of the **lcid** parameter is zero (**0**), the month name is obtained from the current locale. If the date is null or invalid, an exception is raised.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseForCurrentLocale

Signature `parseForCurrentLocale(inputDateString: String): Boolean updating;`

The **parseForCurrentLocale** method of the **Date** primitive type assigns a date to the receiver based on the string specified in the **inputDateString** parameter. The string is assumed to be formatted using the conventions of your current locale. (Define the date convention for your locale by using the **Date** sheet of the Regional Settings Properties dialog, accessed from the Regional Settings icon in the Control Panel.)

This method returns **true** if the string represents a valid date; otherwise it returns **false**. This method handles strings in long or short date format, as shown in the following examples.

```
27 August 2011

Monday, 27 August 2011

2010-10-29

08/27/2010

27 Aug 99

20 noiembrie 1999      // the current locale is set to Romanian, for example
```

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale 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.

Non-English day and month names are handled; that is, locales that have short names with more than three characters or names with non-alphanumeric characters (for example, French and Thai).

parseLongWithCurrentLocale

Signature
```
parseLongWithCurrentLocale(source:    String;
                           errOffset: Integer output): Integer updating;
```

The **parseLongWithCurrentLocale** method of the **Date** primitive type parses the string specified in the **source** parameter to ensure that it matches the long date format of the current locale in terms of element (day, month, year) order and separators and if valid, assigns the value to the receiver.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid date value (the **isValid** method of the **Date** primitive type will return **false**).

This is equivalent to calling the **parseLongWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseLongWithFmtAndLcid

Signature

```
parseLongWithFmtAndLcid(source:    String;
                        fmt:       DateFormat;
                        lcid:      Integer;
                        errOffset: Integer output): Integer updating;
```

The **parseLongWithFmtAndLcid** method of the **Date** primitive type parses the string specified in the **source** parameter to ensure that it matches the long date format specified in the **fmt** parameter in terms of element (day, month, year) order and separator.

Text such as day and month names must match the appropriate values for the locale specified in the **lcid** parameter. If the source string contains a valid date, it is assigned to the receiver; otherwise the invalid date value is assigned to the receiver (the **isValid** method of the **Date** primitive type will return **false**).

If the value of the **fmt** parameter is null, the long date format of the locale specified in the **lcid** parameter is used. If the value of the **lcid** parameter is zero (**0**), the long date format of the current locale is used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid date value.

This method is the same as the **parseLongWithPicAndLcid** method except that the picture string is taken from the **DateFormat** class **longFormat** property. For more details and examples of valid date matches, see the **parseLongWithPicAndLcid** method.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseLongWithPicAndLcid

Signature

```
parseLongWithPicAndLcid(source:    String;
                        pic:       String;
                        lcid:      Integer;
                        errOffset: Integer output): Integer updating;
```

The **parseLongWithPicAndLcid** method of the **Date** primitive type parses the string specified in the **source** parameter to ensure that it matches the long date picture string specified in the **pic** parameter in terms of element (day, month, year) order and separators.

Text such as day and month names must match the appropriate values for the locale specified in the **lcid** parameter. If the source string contains a valid date, it is assigned to the receiver; otherwise the invalid date value is assigned to the receiver (the **isValid** method of the **Date** primitive type will return **false**).

If the value of the **pic** parameter is null, the long date format picture of the locale specified in the **lcid** parameter is used. If the value of the **lcid** parameter is zero (**0**), the long date format picture of the current locale is used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid date value.

The picture element is defined in the **picture** parameter of the **Date** primitive type **format** method.

The following table shows examples of valid matches between source text and format in the English (NZ) locale.

Date Value (Specified by the source Parameter)	Format (Specified by the pic Parameter)
Thursday, 1/July/2010	dddd, d/MMMM/yyyy
1/July/2010	d/MMMM/yyyy
1/7/2010	d/M/yyyy
1 July 2010, Thursday	d MMMM yyyy, dddd
1/7/2010, Thu	d/M/yyyy, ddd
1/Jul/2001	d/MMM/yyyy
1 Jul 2001	d MMM yyyy

Day and month names are matched with the locale table entries using a locale-driven case-insensitive comparison. Abbreviated names are considered to be valid matches for full name specifiers.

Note The date is assumed to be localized Gregorian. Other calendars such as Hebrew Lunar are not supported.

If the first character of the source is a digit, the characters in the long date format picture up to but not including the first day number, month, or year specifier are considered optional. The source can include a month number, where a month name specifier occurs. Leading zeros for day and month numbers are optional when day number, month, and year have intervening separators.

Provided the source text includes values for each of day number, month, and year and the source text ends with the last element, any trailing text in the format picture (for example, separators and day name) is considered optional.

If the year designator is **"yy"**, exactly two digits must be present for the year number. The two-digit value is upgraded to a four-digit value using the **CAL_ITWODIGITYEARMAX** setting, which you can set and modify in the **Regional** settings of the Windows Control Panel. Its default value is **2029**, which gives the following conversions.

- 00…29 becomes 2000…2029
- 30…99 becomes 1930…1999

All other year designators require that exactly four digits must be specified for the year number and its value must be in the range zero (**0**) through 30,000. Negative values (implying the Gregorian BC era) are not supported. If the locale is Thai, the year is assumed to include the Thai year offset (that is, 543), which is subtracted from the date assigned to the receiver.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

All characters in the picture string other than the format specifiers (**d**, **M**, **y**, and **g**) must be matched exactly in the source string. For example, if the format picture string includes some characters before the first specifier, the source must include those exact characters before the first day digit.

Note Single quote characters (**'**) can be used to enclose literal characters, including the format specifier characters. A pair of single quotes occurring in single quoted text is treated as one character. For example, to display the date as **"May '93"**, the format string is **"MMMM ''''yy"**. The first and last single quotation marks are the enclosing quotation marks. The second and third single quotation marks are the escape sequence, to allow the single quotation mark to be displayed before the century.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseShortWithCurrentLocale

Signature

```
parseShortWithCurrentLocale(source:    String;
                            errOffset: Integer output): Integer updating;
```

The **parseShortWithCurrentLocale** method of the **Date** primitive type parses the string specified in the **source** parameter to ensure that it matches the short date format of the current locale in terms of element (day, month, year) order and separators and if valid, assigns the value to the receiver.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid date value (the **isValid** method of the **Date** primitive type will return **false**).

This is equivalent to calling the **parseShortWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseShortWithFmtAndLcid

Signature

```
parseShortWithFmtAndLcid(source:    String;
                         fmt:       DateFormat;
                         lcid:      Integer;
                         errOffset: Integer output): Integer updating;
```

The **parseShortWithFmtAndLcid** method of the **Date** primitive type parses the string specified in the **source** parameter to ensure that it matches the short date format specified in the **fmt** parameter in terms of element (day, month, year) order and separators.

Text such as day and month names must match the appropriate values for the locale specified in the **lcid** parameter. If the source string contains a valid date, it is assigned to the receiver; otherwise the invalid date value is assigned to the receiver (the **isValid** method of the **Date** primitive type will return **false**).

If the value of the **fmt** parameter is null, the short date format of the locale specified in the **lcid** parameter is used. If the value of the **lcid** parameter is zero (**0**), the short date format of the current locale is used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid date value.

This method is the same as the **parseShortWithPicAndLcid** method except that the picture string is taken from the **DateFormat** class **shortFormat** property. For more details and examples of valid date matches, see the **parseShortWithPicAndLcid** method.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseShortWithPicAndLcid

Signature

```
parseShortWithPicAndLcid(source:    String;
                         pic:       String;
                         lcid:      Integer;
                         errOffset: Integer output): Integer updating;
```

The **parseShortWithPicAndLcid** method of the **Date** primitive type parses the string specified in the **source** parameter to ensure that it matches the short date picture string specified in the **pic** parameter in terms of element (day, month, year) order and separators.

Text such as day and month names must match the appropriate values for the locale specified in the **lcid** parameter. If the source string contains a valid date, it is assigned to the receiver; otherwise the invalid date value is assigned to the receiver (the **isValid** method of the **Date** primitive type will return **false**).

If the value of the **pic** parameter is null, the short date format picture of the locale specified in the **lcid** parameter is used. If the value of the **lcid** parameter is zero (**0**), the short date format picture of the current locale is used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid date value. The picture element is defined in the **picture** parameter of the **Date** primitive type **format** method.

The following are examples of valid matches between source text and format in the English (NZ) locale.

Date Value (Specified by the source Parameter)	Format (Specified by the pic Parameter)
01072001	ddMMyyyy
1/Jul/2010	d/MMM/yyyy
1/7/2010	d/M/yyyy
1 Jul 2001	d MMM yyyy
1/07/2001	d/MM/yyyy

Day and month names are matched with the locale table entries using a locale-driven case-insensitive comparison. Abbreviated names are considered to be valid matches for full name specifiers.

Note The date is assumed to be localized Gregorian. Other calendars such as Hebrew Lunar are not supported.

If the first character of the source is a digit, the characters in the short date format picture up to but not including the first day number, month, or year specifier are considered optional. The source can include a month number, where a month name specifier occurs. Leading zeros for day and month numbers are optional when day number, month, and year have intervening separators.

Provided the source text includes values for each of day number, month, and year and the source text ends with the last element, any trailing text in the format picture (for example, separators and day name) is considered optional.

If the year designator is **"yy"**, exactly two digits must be present for the year number. The two-digit value is upgraded to a four-digit value using the **CAL_ITWODIGITYEARMAX** setting, which you can set and modify in the **Regional** settings of the Windows Control Panel. Its default value is **2029**, which gives the following conversions.

- 00…29 becomes 2000…2029
- 30…99 becomes 1930…1999

All other year designators require that exactly four digits must be specified for the year number and its value must be in the range zero (**0**) through 30,000. Negative values (implying the Gregorian BC era) are not supported. If the locale is Thai, the year is assumed to include the Thai year offset (that is, 543), which is subtracted from the date assigned to the receiver.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

All characters in the picture string other than the format specifiers (**d**, **M**, **y**, and **g**) must be matched exactly in the source string. For example, if the format picture string includes some characters before the first specifier, the source must include those exact characters before the first day digit.

Note Single quote characters (') can be used to enclose literal characters, including the format specifier characters. A pair of single quotes occurring in single quoted text is treated as one character. For example, to display the date as **"May '93"**, the format string is **"MMM ""'yy"**. The first and last single quotation marks are the enclosing quotation marks. The second and third single quotation marks are the escape sequence, to allow the single quotation mark to be displayed before the century.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

setByteOrderLocal

Signature `setByteOrderLocal(architecture: Integer): Date;`

The **setByteOrderLocal** method of the **Date** primitive type returns a date that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the **architecture** parameter.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature `setByteOrderRemote(architecture: Integer): Date;`

The **setByteOrderRemote** method of the **Date** primitive type returns a date that has the bytes ordered as required by the remote node indicated by the **architecture** parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setDate

Signature
```
setDate(day:   Integer;
        month: Integer;
        year:  Integer) updating;
```

The **setDate** method of the **Date** primitive type sets the receiver to a specific date, as shown in the following example.

```
testDateSet();
vars
    date : Date;
begin
    date.setDate(31, 08, 1);
    write date;                // Outputs 31 August 2001
end;
```

The parameters are the integer values for day, month, and year.

Note When you specify a value for the **year** parameter in the range **0** through **99**, the year is assumed to be in the current century. If you want an absolute date in this range, use the **Date**::**setDateYearAbsolute** method.

setDateYearAbsolute

Signature

```
setDateYearAbsolute(day:   Integer;
                    month: Integer;
                    year:  Integer) updating;
```

The **setDateYearAbsolute** method of the **Date** primitive type sets the receiver to a specific absolute date. Use this method to specify a date in any year.

The integer values that you define in the **day**, **month**, and **year** variables are initialized when your method is invoked.

The following example shows the use of the **setDateYearAbsolute** method.

```
testAbsoluteDateSet();
vars
    date : Date;
begin
    date.setDateYearAbsolute(31, 08, 1);
    write date;                          // Outputs 31 August 0001
end;
```

See also the **Date**::**setDate** method.

shortDayNameWithLcid

Signature

```
shortDayNameWithLcid(lcid: Integer): String;
```

The **shortDayNameWithLcid** method of the **Date** primitive type returns a string containing the short name of the week day from the locale specified in the **lcid** parameter for the receiver date.

If the value of the **lcid** parameter is zero (**0**), the short day name is obtained from the current locale. If the date is null or invalid, an exception is raised.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

shortFormat

Signature

```
shortFormat(): String;
```

The **shortFormat** method of the **Date** primitive type returns the receiver as a string formatted in the short date format. The following example shows the use of the **shortFormat** method.

```
testDateShort();
vars
    date : Date;
begin
    write "The date is " & date.shortFormat;
end;
```

The output from this example depends on the short format defined for the current locale; for example, it may write **The date is 31/8/99**.

Notes You can use the **defineShortDateFormat** method of the **DateFormat** class if you want to create your own transient format objects and define a short date format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

The **shortFormat** method returns **"*invalid*"** for dates before 1601 or after 30827.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled.

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.

shortMonthNameWithLcid

Signature `shortMonthNameWithLcid(lcid: Integer): String;`

The **shortMonthNameWithLcid** method of the **Date** primitive type returns a string containing the short name of the month from the locale specified in the **lcid** parameter for the receiver date.

If the value of the **lcid** parameter is zero (**0**), the short month name is obtained from the current locale. If the date is null or invalid, an exception is raised.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

subtract

Signature `subtract(date: Date): TimeStampInterval;`

The **subtract** method of the **Date** primitive type returns the interval between the receiver and the **date** parameter as a **TimeStampInterval** value.

The following example shows the use of the **subtract** method.

```
vars
    today, tomorrow : Date;
begin
    tomorrow := today + 1;
    write tomorrow.subtract(today);  // Outputs "1:00:00:00.000"
end;
```

Caution The **TimeStampInterval** value that is returned does not take daylight saving into account.

userFormat

Signature `userFormat(fmt: DateFormat): String;`

The **userFormat** method of the **Date** primitive type returns a string containing the receiver in the supplied date format.

To define your date formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user date format name with a dollar sign (**$**); for example, **userFormat($MyDate)**.

You can use the **defineLongDateFormat** or **defineShortDateFormat** method of the **DateFormat** class if you want to create your own transient format objects and define a long or short date format that dynamically overrides the appropriate format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

In the Windows Environment, the **userFormat** method returns **"*invalid*"** for dates before 1601 or after 30827.

userLongFormatAndLcid

Signature
```
userLongFormatAndLcid(fmt:  DateFormat;
                      lcid: Integer): String;
```

The **userLongFormatAndLcid** method of the **Date** primitive type returns a string containing the receiver in the long date format specified for the **fmt** parameter using the locale specified in the **lcid** parameter.

If the value of the **fmt** parameter is null, the long date format of the locale specified in the **lcid** parameter is returned. If the value of the **lcid** parameter is zero (**0**), the long date format of the current locale is returned.

This method is the same as the **userLongFormatPicAndLcid** method except that the picture string is taken from the **DateFormat** class **longFormat** property. For more details and examples of valid date matches, see the **userLongFormatPicAndLcid** method.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userLongFormatPicAndLcid

Signature
```
userLongFormatPicAndLcid(pic:  String;
                         lcid: Integer): String;
```

The **userLongFormatPicAndLcid** method of the **Date** primitive type returns a string containing the receiver in the long date format picture specified for the **pic** parameter using the locale specified in the **lcid** parameter. For examples of **pic** values, see the **parseLongWithPicAndLcid** method.

If the value of the **pic** parameter is null, the long date format picture of the locale specified in the **lcid** parameter is returned. If the value of the **lcid** parameter is zero (**0**), the long date format picture of the current locale is returned. If the locale is Thai, the year is assumed to include the Thai year offset (that is, 543), which is added to the year included in the returned string.

Note The output text string is a localized Gregorian version of the receiver date. Other calendars such as Hebrew Lunar are not supported.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userShortFormatAndLcid

Signature

```
userShortFormatAndLcid(fmt:  DateFormat;
                       lcid: Integer): String;
```

The **userShortFormatAndLcid** method of the **Date** primitive type returns a string containing the receiver in the short date format specified for the **fmt** parameter of the locale specified in the **lcid** parameter.

If the value of the **fmt** parameter is null, the short date format of the locale specified in the **lcid** parameter is returned. If the value of the **lcid** parameter is zero (**0**), the short date format of the current locale is returned.

This method is the same as the **userShortFormatPicAndLcid** method except that the picture string is taken from the **DateFormat** class **shortFormat** property. For more details and examples of valid date matches, see the **userShortFormatPicAndLcid** method.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userShortFormatPicAndLcid

Signature

```
userShortFormatPicAndLcid(pic:  String;
                          lcid: Integer): String;
```

The **userShortFormatPicAndLcid** method of the **Date** primitive type returns a string containing the receiver in the short date format picture specified for the **pic** parameter of the locale specified in the **lcid** parameter. For examples of **pic** values, see the **parseShortWithPicAndLcid** method.

If the value of the **pic** parameter is null, the short date format picture of the locale specified in the **lcid** parameter is returned. If the value of the **lcid** parameter is zero (**0**), the short date format picture of the current locale is returned.

Notes The output text string is a localized Gregorian version of the receiver date. Other calendars such as Hebrew Lunar are not supported.

If the locale is Thai, the Thai year offset (that is, 543) is added to the year included in the returned string.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

year

Signature `year(): Integer;`

The **year** method of the **Date** primitive type returns the year (for the date value of the receiver) as an integer; for example, **1999**.

The following example shows the use of the **year** method.

```
if text <> "" then
    calendar.changeType := calendar.ChangeType_Day;
    calendar.date.setDate(text.Integer, calendar.date.month,
                          calendar.date.year);
endif;
```

Decimal Type

Use the **Decimal** primitive type to define a variable with a fixed-precision and decimal point; for example, a monetary value such as a bank balance.

Note You must specify a decimal descriptor for the integer length of the decimal variable; for example, code **d : Decimal [4];** to specify a length of *4*. You can optionally specify the number of decimal places for the decimal variable; for example, code **d : Decimal [4,3];** to specify that the variable is to be to three decimal places. (If the number of decimal places is not specified, it is assumed to be zero.)

The value of the integer length must be in the range **1** through **23**. The number of decimal places must be equal to or less than the length value of the decimal descriptor.

As the decimal value for zero (**0**) has no fixed precision or scale factor, you can assign this value to any **Decimal** value. When this value is output (for example, by a **write** instruction), the scale factor of the property or variable defines the number of decimal places that are displayed. Null decimal values are initialized with a value of zero (**0**).

When performing decimal arithmetic, only the final assignment result is rounded. For example:

```
vars
    a      : Decimal[12,4];
    b, tot : Decimal[12,2];
begin
    a := 4.9350;
    b := a;
    tot := tot + a + a;
    write b;            // outputs 4.94, being the final assignment result
    write tot;          // outputs 9.87
end;
```

Any intermediate result keeps as much precision as possible, to minimize the overall rounding loss. To force the rounding or truncation of intermediate results, use the **roundedTo** or **truncatedTo** method, respectively. You can use the **JadeEditMask** class and **TextBox** class **getTextAsDecimal** and **setTextFromDecimal** methods to handle locale formatting for numeric fields.

For details about the methods defined in the **Decimal** primitive type, see "Decimal Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Decimal Methods

The methods defined in the **Decimal** primitive type are summarized in the following table.

Method	Description
abs	Returns the absolute value of the receiver
asBinary	Returns the **Binary** representation of the receiver
asDecimal	Returns the receiver as a **Decimal** with the specified precision and decimal places
currencyFormat	Returns the receiver as a string in the currency format of the current locale
display	Returns a string containing the receiver

Method	Description
getDeclaredPrecision	Returns the declared precision (length) of a **Decimal** variable
getDeclaredScaleFactor	Returns the declared number of decimal places of a **Decimal** variable
numberFormat	Returns the receiver as a string in the numeric format of the current locale
parseCurrencyWithCurrentLocale	Sets the receiver to the result of parsing a string representing a currency value for the current locale
parseCurrencyWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale
parseNumberWithCurrentLocale	Sets the receiver to the result of parsing a string representing a number for the current locale
parseNumberWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale
rounded	Returns an integer containing the rounded value of the receiver
rounded64	Returns a 64-bit integer containing the rounded value of the receiver
roundedTo	Returns the receiver rounded to the specified number of decimal places
setByteOrderLocal	Returns a **Decimal** that has the bytes ordered as required by the local node
setByteOrderRemote	Returns a **Decimal** that has the bytes ordered as required by the specified remote node
truncated	Returns an integer containing the truncated value of the receiver
truncated64	Returns a 64-bit integer containing the truncated value of the receiver
truncatedTo	Returns the receiver truncated to the specified number of decimal places
userCurrencyFormat	Returns the receiver as a string in the specified currency format
userCurrencyFormatAndLcid	Returns the receiver as a string in the specified currency format for the current locale
userNumberFormat	Returns the receiver as a string in the specified number format
userNumberFormatAndLcid	Returns the receiver as a string in the specified number format for the specified locale

abs

Signature `abs(): Decimal;`

The **abs** method of the **Decimal** primitive type returns a decimal containing the absolute value of the receiver.

asBinary

Signature `asBinary(): Binary;`

The **asBinary** method of the **Decimal** primitive type returns the **Binary** representation of the receiver.

In situations where the compatibility with earlier releases is required, you should use this method in preference to casting the **Decimal** value to a **Binary** value (that is, **Decimal.Binary**).

Casting a **Decimal** value to a **Binary** returns the binary representation of the internal representation of the decimal. This depends on the implementation, which is subject to change.

```
dec := bin.asDecimal;    // Prefer this to "dec := bin.Decimal;"
```

To convert the resulting binary value back to a decimal value, use the **asDecimal** method of the **Binary** primitive type.

asDecimal

Signature

```
asDecimal(precision:     Integer;
          decimalPlaces: Integer): Decimal;
```

The **asDecimal** method of the **Decimal** primitive type returns the receiver as a **Decimal** with the length and scale factor specified by the values of the **precision** and **decimalPlaces** parameters, respectively.

You can use this method to convert a **Decimal** value returned by a method to a format with the specified precision and decimal places. If you reduce the number of decimal places, rounding occurs. If the receiver value overflows the specified precision, an exception is raised.

currencyFormat

Signature

```
currencyFormat(): String;
```

The **currencyFormat** method of the **Decimal** primitive type returns a string containing the receiver in the currency format defined for the current locale; for example, **$123.22** or **-123.22$**. This can include currency symbols, thousands separators, sign characters, and decimal point characters.

The following examples show the use of the **currencyFormat** method.

```
// calculate average cost of transactions
tblPortfolio.column := 2;
tblPortfolio.text := portfolio.averageCost.Decimal.currencyFormat;
testDecimal();
vars
    decimalValue : Decimal [12,4];      // Define the Decimal variable
begin
    decimalValue := 1234.56;            // Defines the variable value
    write decimalValue;                 // Outputs  1234.5600
    write decimalValue.currencyFormat;  // Outputs $1,234.56
    decimalValue := -123456;            // Defines the variable value
    write decimalValue;                 // Outputs  -123456.0000
    write decimalValue.currencyFormat;  // Outputs ($123,456.00)
end;
```

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

display

Signature

```
display(): String;
```

The **display** method of the **Decimal** primitive type returns a string containing the receiver.

getDeclaredPrecision

Signature `getDeclaredPrecision(): Integer;`

The **getDeclaredPrecision** method of the **Decimal** primitive type returns the declared length of the receiver; that is, the first value within the brackets (**[]**) defined for the **Decimal** variable.

getDeclaredScaleFactor

Signature `getDeclaredScaleFactor(): Integer;`

The **getDeclaredScaleFactor** method of the **Decimal** primitive type returns the declared scale factor, or number of decimal places, of the receiver.

This method returns the optional second value within the brackets (**[]**) defined for the **Decimal** variable. (The first value is the precision, or length, of the **Decimal** variable.)

If the scale factor was not declared, this method returns zero (**0**).

numberFormat

Signature `numberFormat(): String;`

The **numberFormat** method of the **Decimal** primitive type returns a string containing the receiver in the numeric format defined for the current locale; for example, **07456.357** or **7,456.38**. This can include thousands separators, sign characters, and decimal point characters.

The following example shows the use of the **numberFormat** method.

```
testDecimal();
vars
    decimalValue : Decimal [12,4];      // Define the Decimal variable
begin
    decimalValue := 1234.56;            // Defines the variable value
    write decimalValue;                 // Outputs  1234.5600
    write decimalValue.numberFormat;    // Outputs 1,234.56
    decimalValue := -123456;            // Defines the variable value
    write decimalValue;                 // Outputs  -123456.0000
    write decimalValue.numberFormat;    // Outputs -123,456.00
end;
```

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseCurrencyWithCurrentLocale

Signature
```
parseCurrencyWithCurrentLocale(source:    String;
                               errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithCurrentLocale** method of the **Decimal** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Decimal** format of the current locale for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

This is equivalent to calling the **parseCurrencyWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseCurrencyWithFmtAndLcid

Signature

```
parseCurrencyWithFmtAndLcid(source:    String;
                            fmt:       CurrencyFormat;
                            lcid:      Integer;
                            errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithFmtAndLcid** method of the **Decimal** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for currency character sequence, currency position, sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

The currency character sequence is optional but if it is included in the source, it must be correctly positioned as defined by the **NumberFormat** class **negativeFormat** property and the **CurrencyFormat** class **positiveFormat** property.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency character sequence is optional.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The decimal descriptor of the receiver adds restrictions to the permitted value in the **source** parameter string; for example, a descriptor of **[8,2]** allows the value to have up to eight significant digits, with at most two significant digits following the decimal point and six preceding it. Leading zeros before the decimal point and trailing zeros after the decimal point are ignored. The following values are valid.

- "123456.78"
- "1"
- "00000001.2300"
- ".01"

All significant digits in the **source** parameter string must be able to be stored in the receiver so that they are all shown if the receiver is converted back to a string.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithCurrentLocale

Signature

```
parseNumberWithCurrentLocale(source:    String;
                             errOffset: Integer output): Integer updating;
```

The **parseNumberWithCurrentLocale** method of the **Decimal** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Decimal** format of the current locale for sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

This is equivalent to calling the **parseNumberWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithFmtAndLcid

Signature

```
parseNumberWithFmtAndLcid(source:    String;
                          fmt:       NumberFormat;
                          lcid:      Integer;
                          errOffset: Integer output): Integer updating;
```

The **parseNumberWithFmtAndLcid** method of the **Decimal** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid value.

The negative sign sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional. There is no positive sign sequence.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The decimal descriptor of the receiver adds restrictions to the permitted value in the **source** parameter string; for example, a descriptor of **[8,2]** allows the value to have up to eight significant digits, with at most two significant digits following the decimal point and six preceding it. Leading zeros before the decimal point and trailing zeros after the decimal point are ignored.

The following values are valid.

- "123456.78"
- "1"
- "00000001.2300"
- ".01"

All significant digits in the **source** parameter string must be able to be stored in the receiver so that they are all shown if the receiver is converted back to a string.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

rounded

Signature `rounded(): Integer;`

The **rounded** method of the **Decimal** primitive type returns an integer containing the rounded value of the receiver.

The following code fragment shows the use of the **rounded** method.

```
write (6.4).Decimal.rounded;   // outputs 6 [0,1,2,3,4 are rounded down]
write (6.5).Decimal.rounded;   // outputs 7 [5,6,7,8,9 are rounded up]

write (-6.4).Decimal.rounded;  // outputs -6 [0,1,2,3,4 are rounded up]
write (-6.5).Decimal.rounded;  // outputs -7 [5,6,7,8,9 are rounded down]
```

rounded64

Signature `rounded64(): Integer64;`

The **rounded64** method of the **Decimal** primitive type returns a signed 64-bit integer containing the rounded value of the receiver.

The following code fragment shows the use of the **rounded64** method.

```
write (6.4).Decimal.rounded64;   // outputs 6 [0,1,2,3,4 are rounded down]
write (6.5).Decimal.rounded64;   // outputs 7 [5,6,7,8,9 are rounded up]

write (-6.4).Decimal.rounded64;  // outputs -6 [0,1,2,3,4 are rounded up]
write (-6.5).Decimal.rounded64;  // outputs -7 [5,6,7,8,9 are rounded down]
```

roundedTo

Signature `roundedTo(decimalPlaces: Integer): Decimal;`

The **roundedTo** method of the **Decimal** primitive type returns the receiver rounded to the number of decimal places specified in the **decimalPlaces** parameter.

The following code fragment shows the use of the **roundedTo** method.

```
write (3.64).Decimal.roundedTo(1);   // outputs 3.6 [0,1,2,3,4 are rounded down]
write (3.65).Decimal.roundedTo(1);   // outputs 3.7 [5,6,7,8,9 are rounded up]

write (-3.64).Decimal.roundedTo(1);  // outputs -3.6 [0,1,2,3,4 are rounded up]
write (-3.65).Decimal.roundedTo(1);  // outputs -3.7 [5,6,7,8,9 are rounded down]
```

setByteOrderLocal

Signature `setByteOrderLocal(architecture: Integer): Decimal;`

The **setByteOrderLocal** method of the **Decimal** primitive type returns a decimal that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the **architecture** parameter.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature `setByteOrderRemote(architecture: Integer): Decimal;`

The **setByteOrderRemote** method of the **Decimal** primitive type returns a decimal that has the bytes ordered as required by the remote node indicated by the **architecture** parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

truncated

Signature `truncated(): Integer;`

The **truncated** method of the **Decimal** primitive type returns an integer containing the truncated value of the receiver.

Caution As the **truncated** method returns an integer value, an integer overflow situation occurs when the returned integer value is greater than the value of the global constant **Max_Integer**, which is the limit for the **Integer** type. To safeguard against this when truncating a large decimal value, use the **Decimal** type **truncatedTo** method with a parameter of **0** decimal places. Alternatively use the **Decimal** type **truncated64** method.

The following example shows the use of the **truncated** method.

```
testDecimal();
vars
    decimalValue : Decimal [12,4];
begin
    decimalValue := 340.5678;                // Defines the variable value
    decimalValue := (decimalValue / 20).truncated;
    write decimalValue;                      // Outputs 17.0000
end;
```

truncated64

Signature `truncated64(): Integer64;`

The **truncated64** method of the **Decimal** primitive type returns a signed 64-bit integer containing the truncated value of the receiver.

The following example shows the use of the **truncated** method.

```
testDecimal();
vars
    decimalValue : Decimal [12,4];
begin
    decimalValue := 340.5678;                // Defines the variable value
    decimalValue := (decimalValue / 20).truncated64;
```

```
    write decimalValue;                     // Outputs 17.0000
end;
```

truncatedTo

Signature `truncatedTo(decimalPlaces: Integer): Decimal;`

The **truncatedTo** method of the **Decimal** primitive type returns the receiver truncated to the number of decimal places specified in the **decimalPlaces** parameter.

The following example shows the use of the **truncatedTo** method.

```
testDecimal();
vars
    decimalValue : Decimal [12,4];
begin
    decimalValue := 340.56789;              // Defines the variable value
    decimalValue := (decimalValue/20).truncatedTo(2);
    write decimalValue;                     // Outputs 17.0200
end;
```

userCurrencyFormat

Signature `userCurrencyFormat(fmt: CurrencyFormat): String;`

The **userCurrencyFormat** method of the **Decimal** primitive type returns a string containing the receiver in the currency format specified in the **fmt** parameter.

To define your currency formats, use the Schema menu **Format** command from the Schema Browser.

The code fragment in the following example shows the use of the **userCurrencyFormat** method.

```
lblBank.caption  := app.crntInvestor.cash.userCurrencyFormat($DollarsCents);
lblWorth.caption := (totalWorth).userCurrencyFormat($DollarsCents);
```

Notes When you use a format in a JADE method, prefix your user currency format name with a dollar sign (**$**); for example, **userCurrencyFormat($MyCurrency)**.

You can use the **defineCurrencyFormat** method of the **CurrencyFormat** class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userCurrencyFormatAndLcid

Signature
```
userCurrencyFormatAndLcid(fmt:  CurrencyFormat;
                          lcid: Integer): String;
```

The **userCurrencyFormatAndLcid** method of the **Decimal** primitive type returns a string containing the receiver in the currency format and locale specified in the **fmt** parameter and **lcid** parameter, respectively. If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature
```
userNumberFormat(fmt: NumberFormat): String;
```

The **userNumberFormat** method of the **Decimal** primitive type returns a string containing the receiver in the number format specified in the **fmt** parameter. To define your numeric formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user number format name with a dollar sign (**$**); for example, **userNumberFormat($MyNumber)**.

You can use the **defineNumberFormat** method of the **NumberFormat** class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The code fragment in the following example shows the use of the **userNumberFormat** method.

```
if totalWorth = 0 then
    tblPortfolio.text := "0.00";
else
    tblPortfolio.text := (portfolio.myCompany.currentPrice *
                          portfolio.shares.Decimal * Percent
                          / totalWorth).Decimal.userNumberFormat($Percent);
endif;
```

userNumberFormatAndLcid

Signature
```
userNumberFormatAndLcid(fmt:  NumberFormat;
                        lcid: Integer): String;
```

The **userNumberFormatAndLcid** method of the **Decimal** primitive type returns a string containing the receiver in the number format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Integer Type

The **Integer** primitive type represents the set of positive and negative whole numbers in the range -2,147,483,648 through 2,147,483,647. Any value of the **Integer** primitive type is therefore a whole number.

JADE defines a number of arithmetic operations that take integer operators and return integer results, as listed in the following table.

Operand	Description
+	Add
-	Subtract
*	Multiply
div	Integer division (division with truncation; for example, **7 div 3 = 2**)
^	Exponentiation (for example, **i ^ 3** is i cubed)
mod	Modulus (remainder after integer division)

These are binary (or dyadic) infix operators; that is, they are used with two operands written on each side of the operator (for example, **a+b**). However, the **+** operator and **-** operator can also be used as unary (or monadic) prefix operators, as listed in the following table

Unary Prefix Operator	Description
+a	Sign identify
-a	Sign inversion

The div operator (integer division) performs division with truncation; for example, **7 div 3 = 2**.

An integer variable can contain any *whole* number in the range -2,147,483,648 through 2,147,483,647.

The following table lists valid operations for the **Integer** primitive type.

Expression	Expression Type
integer-expression + date-expression	(date)
integer-expression + time-expression	(time)

You can use the **JadeEditMask** class and **TextBox** class **getTextAsInteger** and **setTextFromInteger** methods to handle locale formatting for numeric fields.

For details about the methods defined in the **Integer** primitive type, see "Integer Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Integer Methods

The methods defined in the **Integer** primitive type are summarized in the following table.

Method	Description
abs	Returns the absolute value of the receiver
bitAnd	Returns an integer representing the receiver bits ANDed with the argument
bitNot	Returns an integer whose bit values are the inverse of the bit values of the receiver
bitOr	Returns an integer representing the receiver bits ORed with the argument
bitXor	Returns an integer representing the receiver bits XORed with the argument
display	Returns the receiver as a string
isEven	Returns **true** if the receiver represents an even number; otherwise **false**
isOdd	Returns **true** if the receiver represents an odd number; otherwise **false**
max	Returns the larger value of the receiver and a specified **Integer**
min	Returns the lesser value of the receiver and a specified **Integer**
numberFormat	Returns a string in the number format of the current locale
padLeadingWith	Returns a copy of the receiver padded to the specified length with a leading character
parseCurrencyWithCurrentLocale	Sets the receiver to the result of parsing a string representing a currency value for the current locale
parseCurrencyWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale
parseNumberWithCurrentLocale	Sets the receiver to the result of parsing a string representing a number for the current locale
parseNumberWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale
setByteOrderLocal	Returns an integer that has the bytes ordered as required by the local node
setByteOrderRemote	Returns an integer that has the bytes ordered as required by a specified remote node
userCurrencyFormat	Returns the receiver as a string in the specified currency format
userCurrencyFormatAndLcid	Returns the receiver as a string in the specified currency format for the specified locale
userNumberFormat	Returns the receiver as a string in the specified number format
userNumberFormatAndLcid	Returns the receiver as a string in the specified number format for the specified locale

abs

Signature `abs(): Integer;`

The **abs** method of the **Integer** primitive type returns an integer containing the absolute value of the receiver.

bitAnd

Signature `bitAnd(op: Integer): Integer;`

The **bitAnd** method of the **Integer** primitive type compares each bit in the receiver with the corresponding bit in the **op** parameter, and returns an integer representing the receiver bits ANDed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
Both bits are 1	1
One or both bits are not 1	0

The following example shows the use of the **bitAnd** method.

```
vars
    platform     : Integer;
    version      : String;
    architecture : Integer;
begin
    platform := node.getOSPlatform(version, architecture);
    if platform.bitAnd(Node.OSWindows) <> 0 then
        // operating system is Windows family (10, 8, 7, 2008, or
        // Vista)
        if platform = Node.OSWindowsHome then
            // version is an older version of Windows (unsupported)
            return 'Windows (unsupported) ' & version;
        endif;
        if platform = Node.OSWindowsEnterprise then
            // version is Windows 10, Windows 8, Windows 7, Windows
            // Server 2012, Windows Server 2008, or Windows Vista
            return 'Windows ' & version;
        endif;
        if platform = Node.OSWindowsMobile then
            // version is Windows CE
            return 'Windows CE ' & version;
        endif;
    endif;
    return '* Unknown platform: ' & platform.String & ' version: ' &
           version;
end;
```

bitNot

Signature `bitNot(): Integer;`

The **bitNot** method of the **Integer** primitive type returns an integer whose bit values are the inverse of the bit values of the receiver.

The generated return values are listed in the following table.

Bits in Receiver	Corresponding Bit in Return Value
Bit is not 1	1
Bit is 1	0

bitOr

Signature `bitOr(op: Integer): Integer;`

The **bitOr** method of the **Integer** primitive type compares each bit in the receiver with the corresponding bit in the **op** parameter, and returns an integer representing the receiver bits ORed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
One or both bits are 1	1
Neither bit is 1	0

The code fragment in the following example shows the use of the **bitOr** method.

```
constants
    BitFlagNone = #00;
    BitFlag1    = #01;
vars
    int : Integer;
begin
    int := BitFlagNone;
    // set bit flag 1
    int := int.bitOr(BitFlag1);
    // test that bit flag 1 is set
    if int.bitAnd(BitFlag1) <> 0 then
        write "flag 1 is set";
    endif;
end;
```

bitXor

Signature `bitXor(op: Integer): Integer;`

The **bitXor** method of the **Integer** primitive type compares each bit in the receiver with the corresponding bit specified in the **op** parameter, and returns an integer representing the receiver bits XORed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
The bits are complementary	1
The bits are not complementary	0

display

Signature `display(): String;`

The **display** method of the **Integer** primitive type returns a string containing the receiver.

isEven

Signature `isEven(): Boolean;`

The **isEven** method of the **Integer** primitive type returns **true** if the receiver represents an even number or it returns **false** if it does not.

The following example shows the use of the **isEven** method.

```
setColor(prod: Product) updating;
begin
    if prod.isNew 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;
```

isOdd

Signature `isOdd(): Boolean;`

The **isOdd** method of the **Integer** primitive type returns **true** if the receiver represents an odd number; otherwise, it returns **false**.

The code fragment in the following example shows the use of the **isOdd** method.

```
if sides.isOdd then
    selectedRectangle.left := (maxWidth - defaultWidth - defaultSpacing) / 2 + 10;
    selectedRectangle.top  := (sides - 1) * (defaultHeight + defaultSpacing)
                              / 2 + 10;
else
    selectedRectangle.left := maxWidth / 2 + 10;
    selectedRectangle.top := maxHeight / 2 - defaultHeight - defaultSpacing + 10;
endif;
```

max

Signature `max(int: Integer): Integer;`

The **max** method of the **Integer** primitive type returns the larger value of the receiver and the **int** parameter.

If the value of the receiver is greater than the value of the **int** parameter, the value of the receiver is returned. If the value of the receiver is less than or equal to the value of the **int** parameter, the value of **int** is returned.

min

Signature `min(int: Integer): Integer;`

The **min** method of the **Integer** primitive type returns the lesser value of the receiver and the **int** parameter.

If the value of the receiver is less than the value of the **int** parameter, the value of the receiver is returned. If the value of the receiver is greater than or equal to the value of the **int** parameter, the value of **int** is returned.

numberFormat

Signature `numberFormat(): String;`

The **numberFormat** method of the **Integer** primitive type returns a string in the number format defined for the current locale; for example, **-7456.000** or **7,456**. This can include thousands separators, sign characters, and decimal point characters.

The following example shows the use of the **numberFormat** method.

```
testInteger();
vars
    str : String;
    int : Integer;
begin
    int := -01234567890;
    write int;                  // Outputs -1234567890
    str := int.numberFormat;
    write str;                  // Outputs -1,234,567,890.00
end;
```

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

padLeadingWith

Signature
```
padLeadingWith(char: Character;
               max:  Integer): String;
```

The **padLeadingWith** method of the **Integer** primitive type returns a string of the length specified in the **max** parameter, consisting of the receiving string padded with the leading character specified in the **char** parameter. If the string is equal to or longer than the value specified in the **max** parameter, it is not truncated but the whole string is returned.

The following example shows the use of the **padLeadingWith** method.

```
constants
    PAD_CHARACTER = 'x';
vars
    int : Integer;
    str : String;
begin
    int := -012345;
    str := int.padLeadingWith('w', 15) & ' 678 Sesame St.';
```

```
        write str;     // Outputs wwwwwwwww-12345 678 Sesame St.
        str := int.padLeadingWith('a', 2);
        write str;     // Outputs -12345
        str := int.padLeadingWith(PAD_CHARACTER, 10);
        write str;     // Outputs xxxx-12345
    end;
```

parseCurrencyWithCurrentLocale

Signature

```
parseCurrencyWithCurrentLocale(source:    String;
                               errOffset: Integer output)
                                        : Integer updating;
```

The **parseCurrencyWithCurrentLocale** method of the **Integer** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Integer** format of the current locale for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

This is equivalent to calling the **parseCurrencyWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseCurrencyWithFmtAndLcid

Signature

```
parseCurrencyWithFmtAndLcid(source:    String;
                            fmt:       CurrencyFormat;
                            lcid:      Integer;
                            errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithFmtAndLcid** method of the **Integer** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid value.

The currency character sequence is optional but if it is included in the source, it must be correctly positioned as defined by the **NumberFormat** class **negativeFormat** property and the **CurrencyFormat** class **positiveFormat** property.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency character sequence is optional.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the **source** parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in the **Integer** variable; for example:

- **"100"**, **"100."**, **"100.0"**, **"100.00"**, and **"100.000"** are accepted as valid and equal.
- **"100.01"** and **"100.99"** are rejected, as the value cannot be stored accurately in an **Integer** primitive type.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithCurrentLocale

Signature

```
parseNumberWithCurrentLocale(source:    String;
                             errOffset: Integer output): Integer updating;
```

The **parseNumberWithCurrentLocale** method of the **Integer** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Integer** format of the current locale for sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid value.

This is equivalent to calling the **parseNumberWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithFmtAndLcid

Signature

```
parseNumberWithFmtAndLcid(source:    String;
                          fmt:       NumberFormat;
                          lcid:      Integer;
                          errOffset: Integer output): Integer updating;
```

The **parseNumberWithFmtAndLcid** method of the **Integer** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid value.

The negative sign character sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional. There is no positive sign sequence.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the **source** parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in the **Integer** variable; for example:

- **"100"**, **"100."**, **"100.0"**, **"100.00"**, and **"100.000"** are accepted as valid and equal.
- **"100.01"** and **"100.99"** are rejected, as the value cannot be stored accurately in an **Integer** primitive type.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

setByteOrderLocal

Signature `setByteOrderLocal(architecture: Integer): Integer;`

The **setByteOrderLocal** method of the **Integer** primitive type returns an integer that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the **architecture** parameter.

The **architecture** parameter indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)
No constant; that is, zero (0)	Reorders the bytes from network byte order (a standard for passing binary integers across a TCP/IP connection).

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature `setByteOrderRemote(architecture: Integer): Integer;`

The **setByteOrderRemote** method of the **Integer** primitive type returns an integer that has the bytes ordered as required by the remote node indicated by the **architecture** parameter. The bytes of the receiver are assumed to be ordered as required by the local node.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)
No constant; that is, zero (**0**)	Reorders the bytes from network byte order (a standard for passing binary integers across a TCP/IP connection)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

userCurrencyFormat

Signature `userCurrencyFormat(fmt: CurrencyFormat): String;`

The **userCurrencyFormat** method of the **Integer** primitive type returns a string containing the receiver in the supplied currency format.

To define your currency formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user currency format name with a dollar sign (**$**); for example, **userCurrencyFormat($MyCurrency)**.

You can use the **defineCurrencyFormat** method of the **CurrencyFormat** class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userCurrencyFormatAndLcid

Signature
```
userCurrencyFormatAndLcid(fmt:  CurrencyFormat;
                          lcid: Integer): String;
```

The **userCurrencyFormatAndLcid** method of the **Integer** primitive type returns a string containing the receiver in the currency format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature
```
userNumberFormat(fmt: NumberFormat): String;
```

The **userNumberFormat** method of the **Integer** primitive type returns a string containing the receiver in the supplied number format.

To define your numeric formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user number format name with a dollar sign (**$**); for example, **userNumberFormat($MyNumber)**.

You can use the **defineNumberFormat** method from the **NumberFormat** class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormatAndLcid

Signature
```
userNumberFormatAndLcid(fmt:  NumberFormat;
                        lcid: Integer): String;
```

The **userNumberFormatAndLcid** method of the **Integer** primitive type returns a string containing the receiver in the number format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Integer64 Type

The **Integer64** primitive type represents the set of positive and negative whole numbers in the range -9,223,372,036,854,775,807 through 9,223,372,036,854,775,807. Any value of the **Integer64** primitive type is therefore a whole number.

JADE defines a number of arithmetic operations that take **Integer64** operators and return **Integer64** results, as listed in the following table.

Operator	Description
+	Add
-	Subtract
*	Multiply
div	Integer division (division with truncation; for example, **7 div 3 = 2**)
^	Exponentiation (for example, **i ^ 3** is i cubed)
mod	Modulus (remainder after integer division)

These are binary (or dyadic) infix operators; that is, they are used with operands on both sides of the operator (for example, **a+b**). However, the **+** operator and **-** operator can also be used as unary (or monadic) prefix operators, as listed in the following table.

Unary Prefix Operator	Description
+a	Sign identify
-a	Sign inversion

The div operator (integer division) performs division with truncation; for example, **7 div 3 = 2**.

For details about the methods defined in the **Integer64** primitive type, see "Integer64 Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Integer64 Methods

The methods defined in the **Integer64** primitive type are summarized in the following table.

Method	Description
abs	Returns the absolute value of the receiver
bitAnd	Returns an integer representing the receiver bits ANDed with the argument
bitNot	Returns an integer whose bit values are the inverse of the bit values of the receiver
bitOr	Returns an integer representing the receiver bits ORed with the argument
bitXor	Returns an integer representing the receiver bits XORed with the argument

Method	Description
display	Returns the receiver as a string
isEven	Returns **true** if the receiver represents an even number; otherwise **false**
isOdd	Returns **true** if the receiver represents an odd number; otherwise **false**
max	Returns the larger value of the receiver and a specified **Integer64**
min	Returns the lesser value of the receiver and a specified **Integer64**
numberFormat	Returns a string in the number format of the current locale
padLeadingWith	Returns a copy of the receiver padded to the specified length with a leading character
parseCurrencyWithCurrentLocale	Sets the receiver to the result of parsing a string representing a currency value for the current locale
parseCurrencyWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale
parseNumberWithCurrentLocale	Sets the receiver to the result of parsing a string representing a number for the current locale
parseNumberWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale
setByteOrderLocal	Returns an integer that has the bytes ordered as required by the local node
setByteOrderRemote	Returns an integer that has the bytes ordered as required by a specified remote node
userCurrencyFormat	Returns the receiver as a string in the specified currency format
userCurrencyFormatAndLcid	Returns the receiver as a string in the specified currency format for the specified locale
userNumberFormat	Returns the receiver as a string in the specified number format
userNumberFormatAndLcid	Returns the receiver as a string in the specified number format for the specified locale

abs

Signature `abs(): Integer64;`

The **abs** method of the **Integer64** primitive type returns an integer containing the absolute value of the receiver.

bitAnd

Signature `bitAnd(op: Integer64): Integer64;`

The **bitAnd** method of the **Integer64** primitive type compares each bit in the receiver with the corresponding bit in the **op** parameter, and returns an integer representing the receiver bits ANDed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
Both bits are 1	1
One or both bits are not 1	0

bitNot

Signature `bitNot(): Integer64;`

The **bitNot** method of the **Integer64** primitive type returns an integer whose bit values are the inverse of the bit values of the receiver.

The generated return values are listed in the following table.

Bits in Receiver	Corresponding Bit in Return Value
Bit is not 1	1
Bit is 1	0

bitOr

Signature `bitOr(op: Integer64): Integer64;`

The **bitOr** method of the **Integer64** primitive type compares each bit in the receiver with the corresponding bit in the **op** parameter, and returns an integer representing the receiver bits ORed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
One or both bits are 1	1
Neither bit is 1	0

The code fragment in the following example shows the use of the **bitOr** method.

```
constants
    BitFlagNone = #00;
    BitFlag1    = #01;
vars
    int : Integer64;
begin
    int := BitFlagNone;
    // set bit flag 1
    int := int.bitOr(BitFlag1);
    // test that bit flag 1 is set
    if int.bitAnd(BitFlag1) <> 0 then
        write "flag 1 is set";
    endif;
end;
```

bitXor

Signature `bitXor(op: Integer64): Integer64;`

The **bitXor** method of the **Integer64** primitive type compares each bit in the receiver with the corresponding bit specified in the **op** parameter, and returns an integer representing the receiver bits XORed with the argument.

The generated return values are listed in the following table.

Bits in Receiver and Operand	Corresponding Bit in Return Value
The bits are complementary	1
The bits are not complementary	0

display

Signature `display(): String;`

The **display** method of the **Integer64** primitive type returns a string containing the receiver.

isEven

Signature `isEven(): Boolean;`

The **isEven** method of the **Integer64** primitive type returns **true** if the receiver represents an even number or it returns **false** if it does not.

isOdd

Signature `isOdd(): Boolean;`

The **isOdd** method of the **Integer64** primitive type returns **true** if the receiver represents an odd number; otherwise, it returns **false**.

max

Signature `max(int: Integer64): Integer64;`

The **max** method of the **Integer64** primitive type returns the larger value of the receiver and the **int** parameter. If the value of the receiver is greater than the value of the **int** parameter, the value of the receiver is returned. If the value of the receiver is less than or equal to the value of the **int** parameter, the value of **int** is returned.

min

Signature `min(int: Integer64): Integer64;`

The **min** method of the **Integer64** primitive type returns the lesser value of the receiver and the **int** parameter. If the value of the receiver is less than the value of the **int** parameter, the value of the receiver is returned. If the value of the receiver is greater than or equal to the value of the **int** parameter, the value of **int** is returned.

numberFormat

Signature `numberFormat(): String;`

The **numberFormat** method of the **Integer64** primitive type returns a string in the number format defined for the current locale; for example, **-7456.000** or **7,456**. This can include thousands separators, sign characters, and decimal point characters.

The following example shows the use of the **numberFormat** method.

```
testInteger();
vars
    str : String;
    int : Integer64;
begin
    int := -01234567890;
    write int;                  // Outputs -1234567890
    str := int.numberFormat;
    write str;                  // Outputs -1,234,567,890.00
end;
```

You can use the **defineNumberFormat** method of the **NumberFormat** class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

padLeadingWith

Signature
```
padLeadingWith(char: Character;
               max:  Integer64): String;
```

The **padLeadingWith** method of the **Integer64** primitive type returns a string of the length specified in the **max** parameter, consisting of the receiving string padded with the leading character specified in the **char** parameter. If the string is equal to or longer than the value specified in the **max** parameter, it is not truncated but the whole string is returned.

The following example shows the use of the **padLeadingWith** method.

```
constants
    PAD_CHARACTER = 'x';
vars
    int : Integer64;
    str : String;
begin
    int := -012345;
    str := int.padLeadingWith('w', 15) & ' 678 Sesame St.';
    write str;     // Outputs wwwwwwwww-12345 678 Sesame St.
    str :int.padLeadingWith('a', 2);
    write str;     // Outputs -12345
    str := int.padLeadingWith(PAD_CHARACTER, 10);
```

```
    write str;     // Outputs xxxx-12345
end;
```

parseCurrencyWithCurrentLocale

Signature

```
parseCurrencyWithCurrentLocale(source:    String;
                               errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithCurrentLocale** method of the **Integer64** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Integer64** format of the current locale for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

This is equivalent to calling the **parseCurrencyWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseCurrencyWithFmtAndLcid

Signature

```
parseCurrencyWithFmtAndLcid(source:    String;
                            fmt:       CurrencyFormat;
                            lcid:      Integer;
                            errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithFmtAndLcid** method of the **Integer64** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid value.

The currency sequence is optional but if it is included in the source, it must be correctly positioned as defined by the **NumberFormat** class **negativeFormat** property and the **CurrencyFormat** class **positiveFormat** property.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency character sequence is optional.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the **source** parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in the **Integer64** variable; for example:

- **"100"**, **"100."**, **"100.0"**, **"100.00"**, and **"100.000"** are accepted as valid and equal.
- **"100.01"** and **"100.99"** are rejected, as the value cannot be stored accurately in an **Integer64** primitive type.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithCurrentLocale

Signature

```
parseNumberWithCurrentLocale(source:    String;
                             errOffset: Integer output): Integer updating;
```

The **parseNumberWithCurrentLocale** method of the **Integer64** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Integer64** format of the current locale for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter.

This is equivalent to calling the **parseNumberWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithFmtAndLcid

Signature

```
parseNumberWithFmtAndLcid(source:    String;
                          fmt:       NumberFormat;
                          lcid:      Integer;
                          errOffset: Integer output): Integer updating;
```

The **parseNumberWithFmtAndLcid** method of the **Integer64** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for sign character sequence, sign position, thousands separator, decimal point sequence, and character set. If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid value.

The negative sign sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional. There is no positive sign sequence.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the **source** parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in the **Integer64** variable; for example:

- **"100"**, **"100."**, **"100.0"**, **"100.00"**, and **"100.000"** are accepted as valid and equal.
- **"100.01"** and **"100.99"** are rejected, as the value cannot be stored accurately in an **Integer64** primitive type.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

setByteOrderLocal

Signature `setByteOrderLocal(architecture: Integer64): Integer64;`

The **setByteOrderLocal** method of the **Integer64** primitive type returns an integer that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the **architecture** parameter.

The **architecture** parameter indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature `setByteOrderRemote(architecture: Integer64): Integer64;`

The **setByteOrderRemote** method of the **Integer64** primitive type returns an integer that has the bytes ordered as required by the remote node indicated by the **architecture** parameter. The bytes of the receiver are assumed to be ordered as required by the local node.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)
No constant; that is, zero (**0**)	Reorders the bytes from network byte order (a standard for passing binary integers across a TCP/IP connection)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

userCurrencyFormat

Signature `userCurrencyFormat(fmt: CurrencyFormat): String;`

The **userCurrencyFormat** method of the **Integer64** primitive type returns a string containing the receiver in the supplied currency format. To define your currency formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user currency format name with a dollar sign (**$**); for example, **userCurrencyFormat($MyCurrency)**.

You can use the **defineCurrencyFormat** method of the **CurrencyFormat** class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userCurrencyFormatAndLcid

Signature
```
userCurrencyFormatAndLcid(fmt:  NumberFormat;
                          lcid: Integer): String;
```

The **userCurrencyFormatAndLcid** method of the **Integer64** primitive type returns a string containing the receiver in the currency format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature `userNumberFormat(fmt: NumberFormat): String;`

The **userNumberFormat** method of the **Integer64** primitive type returns a string containing the receiver in the supplied number format.

To define your numeric formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user number format name with a dollar sign (**$**); for example, **userNumberFormat($MyNumber)**.

You can use the **defineNumberFormat** method from the **NumberFormat** class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormatAndLcid

Signature
```
userNumberFormatAndLcid(fmt:  NumberFormat;
                        lcid: Integer): String;
```

The **userNumberFormatAndLcid** method of the **Integer64** primitive type returns a string containing the receiver in the number format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

MemoryAddress Type

A variable of type **MemoryAddress** is used to represent a memory address; that is, a **void*** pointer as used in C. The primary purpose for this primitive type is to interface with external C and C++ dynamic libraries being used as a parameter or a return type.

When an object with a **MemoryAddress** value is passed between nodes, it passes the value of the memory address. However, the memory address only has meaning for the node on which it was assigned a non-null value.

The following semantic rules apply to **MemoryAddress** values.

- A **MemoryAddress** variable can be assigned a **null** value but not the value zero (**0**).
- A **MemoryAddress** value can be compared to the **null** value or to another **MemoryAddress** value for equality or inequality.
- A **MemoryAddress** value can be assigned to another **MemoryAddress** variable.
- A **MemoryAddress** value cannot be changed by using an arithmetic operation.

Note Unlike other primitive types, a corresponding subclass of **Array** for **MemoryAddress** values does not exist in the **RootSchema**. If you require such an array, subclass the **Array** class in your user schema, selecting **MemoryAddress** as the membership.

For details about the methods defined in the **MemoryAddress** primitive type, see "MemoryAddress Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

MemoryAddress Methods

The methods defined in the **MemoryAddress** primitive type are summarized in the following table.

Method	Returns …
adjust	A changed memory address value
asBinary32	The value of the internal pointer as a 32-bit (4 byte) binary
asBinary64	The value of the internal pointer as a 64-bit (8 byte) binary
display	A string containing the receiver
isValid	**true** if the memory address is valid

adjust

Signature `adjust(offset: Integer64): MemoryAddress;`

The **adjust** method of the **MemoryAddress** primitive type adjusts the value of the receiver using the **offset** parameter and returns a new **MemoryAddress** value. If the **MemoryAddress** is not valid on the current node when this method is called, a 1443 (*MemoryAddress is not valid for current Node*) exception is raised.

The following code steps through a block of memory identified by **iPtr** in 1024 Byte blocks.

```
vars
    iBytesLeft: Integer;
```

```
        iBytesCopy: Integer;
        iPtr: MemoryAddress;
    begin
        ...
        while iBytesLeft > 0 do
            iBytesToCopy := iBytesLeft.min(1024);
            iPtr := iPtr.adjust(iBytesToCopy);
            iBytesLeft := iBytesLeft - iBytesToCopy;
        endwhile;
        ...
    end;
```

asBinary32

Signature `asBinary32(): Binary;`

The **asBinary32** method of the **MemoryAddress** primitive type returns the value of the internal pointer as a 32-bit (4 byte) binary value.

An exception is raised if the process is not running in a 32-bit memory address space.

asBinary64

Signature `asBinary64(): Binary;`

The **asBinary64** method of the **MemoryAddress** primitive type returns the value of the internal pointer as a 64-bit (8 byte) binary value.

An exception is raised if the process is not running in a 64-bit memory address space.

display

Signature `display(): String;`

The **display** method of the **MemoryAddress** primitive type returns a string representing the value of the receiver.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **MemoryAddress** primitive type returns **true** if the value of the receiver was assigned by a process on the current node; that is, the pointer is valid for the current operating system process.

Point Type

A variable of type **Point** is used to represent a point in two-dimensional space. A **Point** primitive type encapsulates two integer values: the *x* (horizontal) and *y* (vertical) coordinates.

You can use the **Point** primitive type to represent a position on the display or within a form or control. (When used to represent a position in a form or control, the integer values represent the units of the **scaleMode** property of the form or control.) Additionally, you can use the **Point** primitive type to represent any other two-dimensional data; for example, points on a graph.

For details about the methods defined in the **Point** primitive type, see "Point Methods", in the following subsection.

For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Point Methods

The methods defined in the **Point** primitive type are summarized in the following table.

Method	Description
display	Returns a string representing the value of the receiver
set	Sets the value of the receiver to a specified point
setX	Sets the x (horizontal) value of the receiver
setY	Sets the y (vertical) value of the receiver
x	Returns the x (horizontal) value of the receiver
y	Returns the y (vertical) value of the receiver

display

Signature `display(): String;`

The **display** method of the **Point** primitive type returns a string representing the value of the receiver.

set

Signature
```
set(x, Integer;
    y: Integer) updating;
```

The **set** method of the **Point** primitive type sets the value of the receiver to a specified point.

The **x** and **y** parameters are integer values of the horizontal (**x**) and vertical (**y**) coordinates.

setX

Signature `setX(x: Integer) updating;`

The **setX** method of the **Point** primitive type sets the *x* (horizontal) value of the receiver.

The following examples show the use of the **setX** method.

```
newPoint.setX(distance.Integer);
```

```
redrawGraph() updating;
vars
    gLine       : Object;
    objectArray : ObjectArray;
begin
    drawNew;
    foreach gLine in objectArray do
        gLine.GLines.drawNew;
        gLine.GLines.lastPoint.setX(0);
        gLine.GLines.redrawGraph;
    endforeach;
end;
```

setY

Signature `setY(y: Integer) updating;`

The **setY** method of the **Point** primitive type sets the *y* (vertical) value of the receiver.

The code fragments in the following examples show the use of the **setY** method.

```
gLine.lastPoint.setY(self.height.Integer);

newPoint.setY(yScale.heightOfYValue(dataArray.last).Integer);
```

x

Signature `x(): Integer;`

The **x** method of the **Point** primitive type returns the *x* (horizontal) value of the receiver.

The following example shows the use of the **x** method.

```
drawLine(lastPoint.x, lastPoint.y, distance + lastPoint.x,
         newPoint.y, color);
lastPoint.setX(lastPoint.x + distance.Integer);
lastPoint.setY(newPoint.y);
```

y

Signature `y(): Integer;`

The **y** method of the **Point** primitive type returns the *y* (vertical) value of the receiver.

The following example shows the use of the **y** method.

```
newPoint.setY(yScale.verticalPixels.Integer - newPoint.y);
```

Real Type

Use the **Real** primitive type to represent a floating point number. A **Real** primitive type has a set of values that is a subset of real numbers. These values can be represented in floating point notation with a fixed number of digits.

Use a **Real** primitive type to store floating point numbers; for example, a temperature.

Real numbers are useful for computations involving very large or very small numbers, or when the range of magnitudes cannot be predicted.

Notes **Real** primitive types provide fifteen digits of precision; that is, if you assign a **Real** value to a **Real** attribute or local variable and then retrieve the value in your method, only the first fifteen significant digits can be relied on for complete accuracy.

As a floating point number stores an approximation of the value that is accurate to fifteen significant digits, you should use an **Integer**, an **Integer64**, or a **Decimal** primitive type to store values where precision is required; for example, for monetary values.

Real numbers are stored internally using an eight-byte representation, providing fifteen significant digits of accuracy.

You can use the **JadeEditMask** class and **TextBox** class **getTextAsReal** and **setTextFromReal** methods to handle locale formatting for numeric fields.

For details about the constants and methods defined in the **Real** primitive type, see "Real Constants" and "Real Methods", in the following subsections. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Real Constants

The constants provided by the **Real** primitive type are listed in the following table.

Constant	Integer Value
FP_Classification_NegInfinity	2
FP_Classification_Normal	6
FP_Classification_NotANumber	1
FP_Classification_PosInfinity	3
FP_Classification_SubNormal	5
FP_Classification_Zero	4

Real Methods

The methods defined in the **Real** primitive type are summarized in the following table.

Method	Description
abs	Returns the absolute value of the receiver
arccos	Returns the arc cosine of the receiver
arcsin	Returns the arc sine of the receiver

Method	Description
arctan	Returns the arc tangent of the receiver
arcTan2	Returns the arc tangent of a point (atan2 function)
cos	Returns the cosine of the receiver
currencyFormat	Returns a string in the currency format of the current locale
display	Returns the receiver as a string
exp	Returns the exponential *e* to the power of the receiver
getFloatingPointClassification	Returns an integer indicating whether the receiver is a normal floating point value or a special value
infinity	Sets the receiver to the special *positive infinity* value
isInfinity	Returns **true** if the receiver has the special *positive infinity* value
isNaN	Returns **true** if the receiver has the special *not a number* value
log	Returns the natural logarithm of the receiver
log10	Returns the base 10 logarithm of the receiver
max	Returns the larger value of the receiver and the specified **Real**
min	Returns the lesser value of the receiver and the specified **Real**
nan	Sets the receiver to the special *not a number* value
numberFormat	Returns a string in the number format of the current locale
parseCurrencyWithCurrentLocale	Sets the receiver to the result of parsing a string representing a currency value for the current locale
parseCurrencyWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale
parseNumberWithCurrentLocale	Sets the receiver to the result of parsing a string representing a number for the current locale
parseNumberWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale
rounded	Returns an integer containing the rounded value of the receiver
rounded64	Returns a 64-bit integer containing the rounded value of the receiver
roundedTo	Returns the receiver rounded to the specified number of decimal places
roundedUp	Returns an integer containing the value of the receiver rounded up to the nearest whole number
roundedUp64	Returns a 64-bit integer containing the value of the receiver rounded up to the nearest whole number
setByteOrderLocal	Returns a **Real** that has the bytes ordered as required by the local node
setByteOrderRemote	Returns a **Real** that has the bytes ordered as required by the specified remote node
setFloatingPointClassification	Sets the receiver to a specified special value
sin	Returns the sine of the receiver

Method	Description
sqrt	Returns the square root of the receiver
tan	Returns the tangent of the receiver
truncated	Returns an integer containing the truncated value of the receiver
truncated64	Returns a 64-bit integer containing the truncated value of the receiver
truncatedTo	Returns the receiver truncated to the specified number of decimal places
userCurrencyFormat	Returns the receiver as a string in the specified currency format
userCurrencyFormatAndLcid	Returns the receiver as a string in the specified currency format for the specified locale
userNumberFormat	Returns the receiver as a string in the specified number format
userNumberFormatAndLcid	Returns the receiver as a string in the specified number format for the specified locale

abs

Signature `abs(): Real;`

The **abs** method of the **Real** primitive type returns a real containing the absolute value of the receiver.

arccos

Signature `arccos(): Real;`

The **arccos** method of the **Real** primitive type returns the arc cosine (or inverse cosine) of the receiver. An exception is raised if the receiver is invalid.

The resulting value represents an angle in degrees.

arcsin

Signature `arcsin(): Real;`

The **arcsin** method of the **Real** primitive type returns the arc sine (or inverse sine) of the receiver. An exception is raised if the receiver is invalid.

The resulting value represents an angle in degrees.

arctan

Signature `arctan(): Real;`

The **arctan** method of the **Real** primitive type returns the arc tangent (inverse tangent) of the receiver. An exception is raised if the receiver is invalid.

The resulting value represents an angle in degrees.

arcTan2

Signature `arcTan2(real: Real): Real;`

The **arcTan2** method of the **Real** primitive type returns the arc tangent (inverse tangent) of the **real** parameter divided by the receiver.

The method implements the *atan2* mathematical function relating to the angle subtended by a point in the Cartesian plane.

```
angle := self.arcTan2(real);
```

An exception is raised if the receiver is invalid. The resulting value represents an angle in degrees.

cos

Signature `cos(): Real;`

The **cos** method of the **Real** primitive type returns the cosine of the receiver. The receiver value represents an angle in degrees, and the resulting value is always in the range **-1** through **1**.

The following example shows the use of the **cos** method.

```
testReal();
vars
    realValue : Real;
begin
    realValue := 34;                    // Defines the variable value
    write realValue.cos;                // Outputs 0.829037572555042
end;
```

currencyFormat

Signature `currencyFormat(): String;`

The **currencyFormat** method of the **Real** primitive type returns a string containing the receiver in the currency format defined for the current locale; for example, **$123.22** or **-123.225**. This can include currency symbols, thousands separators, sign characters, and decimal point characters.

The following examples show the use of the **currencyFormat** method.

```
tblPrices.text := company.currentPrice.currencyFormat;

testReal();
vars
    stringValue : String;
    realValue   : Real;
begin
    realValue := -123456.987;                // Defines the variable value
    write realValue;                         // Outputs -123456.987
    stringValue := realValue.currencyFormat; // Associates string and format
    write stringValue;                       // Outputs ($123,456.99)
    realValue   := 34.5;
    write realValue.max(40.6);               // Outputs 40.6
end;
```

You can use the **defineCurrencyFormat** method of the **CurrencyFormat** class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

display

Signature `display(): String;`

The **display** method of the **Real** primitive type returns a string containing the receiver.

exp

Signature `exp(): Real;`

The **exp** method of the **Real** primitive type returns the exponential *e* to the power of the receiver.

getFloatingPointClassification

Signature `getFloatingPointClassification(): Integer;`

The **getFloatingPointClassification** method of the **Real** primitive type returns an integer that indicates the whether the receiver is a normal floating point value or some other kind of special value.

Floating point numbers can have special values, such as infinity or NaN (Not a Number). Some floating point calculations may produce infinity or NaN as the result of an operation on invalid input operands.

The values returned by the **getFloatingPointClassification** method are represented by the constants defined for the **Real** primitive type and shown in the following table.

Constant	Integer Value
FP_Classification_NegInfinity	2
FP_Classification_Normal	6
FP_Classification_NotANumber	1
FP_Classification_PosInfinity	3
FP_Classification_SubNormal	5
FP_Classification_Zero	4

infinity

Signature `infinity(): Real updating;`

The **infinity** method of the **Real** primitive type sets the receiver to the special *positive infinity* value. It does this by calling the **setFloatingPointClassification** method with the **FP_Classification_PosInfinity** constant value as the parameter. The method also returns the *positive infinity* value.

See also the **isInfinity** method.

isInfinity

Signature `isInfinity(): Boolean;`

The **isInfinity** method of the **Real** primitive type returns **true** if the value returned by the **getFloatingPointClassification** method is the constant **FP_Classification_PosInfinity**, which represents the special *positive infinity* value.

See also the **infinity** method.

isNaN

Signature `isNaN(): Boolean;`

The **isNaN** method of the **Real** primitive type returns **true** if the value returned by the **getFloatingPointClassification** method is the constant **FP_Classification_NotANumber**, which represents the special *not a number* value.

See also the **nan** method.

log

Signature `log(): Real;`

The **log** method of the **Real** primitive type returns the natural logarithm of the receiver. An exception is raised if the receiver is invalid.

log10

Signature `log10(): Real;`

The **log10** method of the **Real** primitive type returns the base 10 logarithm of the receiver. An exception is raised if the receiver is invalid.

max

Signature `max(real: Real): Real;`

The **max** method of the **Real** primitive type returns the larger value of the receiver and the **real** parameter.

If the value of the receiver is greater than the value of the **real** parameter, the value of the receiver is returned. If the value of the receiver is less than or equal to the value of the **real** parameter, the value of **real** is returned.

The following example shows the use of the **max** method.

```
testReal();
vars
    realValue : Real;
begin
    realValue := 34.5;                  // Defines the variable value
    write realValue.max(40.6);          // Outputs 40.6
end;
```

min

Signature `min(real: Real): Real;`

The **min** method of the **Real** primitive type returns the lesser value of the receiver and the **real** parameter.

If the value of the receiver is less than the value of the **real** parameter, the value of the receiver is returned. If the value of the receiver is greater than or equal to the value of the **real** parameter, the value of **real** is returned.

The following example shows the use of the **min** method.

```
testReal();
vars
    realValue : Real;
begin
    realValue := 34.5;                  // Defines the variable value
    write realValue.min(40.6);          // Outputs 34.5
end;
```

nan

Signature `nan(): Real updating;`

The **nan** method of the **Real** primitive type sets the receiver to the special *not a number* value. It does this by calling the **setFloatingPointClassification** method with the **FP_Classification_NotANumber** constant value as the parameter. The method also returns the *not a number* value.

See also the **isNaN** method.

numberFormat

Signature `numberFormat(): String;`

The **numberFormat** method of the **Real** primitive type returns a string containing the receiver in the numeric format defined for the current locale; for example, **07456.357** or **7,456.38**. This can include thousands separators, sign characters, and decimal point characters.

You can use the **defineNumberFormat** method of the **NumberFormat** class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The following example shows the use of the **numberFormat** method.

```
vars
    stringValue : String;
    realValue   : Real;
begin
    realValue := -123456.987;               // Defines the variable value
    write realValue;                        // Outputs -123456.987
    stringValue := realValue.numberFormat;  // Associates string and format
```

```
    write stringValue;                      // Outputs -123,456.99
end;
```

parseCurrencyWithCurrentLocale

Signature

```
parseCurrencyWithCurrentLocale(source:    String;
                               errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithCurrentLocale** method of the **Real** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Real** format of the current locale for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

This is equivalent to calling the **parseCurrencyWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseCurrencyWithFmtAndLcid

Signature

```
parseCurrencyWithFmtAndLcid(source:    String;
                            fmt:       CurrencyFormat;
                            lcid:      Integer;
                            errOffset: Integer output): Integer updating;
```

The **parseCurrencyWithFmtAndLcid** method of the **Real** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for currency character sequence, currency position, sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

The currency character sequence is optional but if it is included in the source, it must be correctly positioned as defined by the **NumberFormat** class **negativeFormat** property and the **CurrencyFormat** class **positiveFormat** property.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency sequence is optional.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

Only the first 15 significant digits of the value of the string are stored in the receiver. Any additional digits are rounded into the fifteenth digit and replaced with zeros. The number of decimal places before and after the decimal point is preserved. Leading zeros before the decimal point and trailing zeros after the decimal point are ignored. For example, **"12345678901234567890"** is parsed and results in **12345678901234600000** being stored in the receiver.

This method supports a maximum of 30 significant digits, whereas the fixed-point representation of a **Real** value can require up to 320 digits.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithCurrentLocale

Signature

```
parseNumberWithCurrentLocale(source:    String;
                             errOffset: Integer output): Integer updating;
```

The **parseNumberWithCurrentLocale** method of the **Real** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Real** format of the current locale for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (**0**).

This is equivalent to calling the **parseNumberWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithFmtAndLcid

Signature

```
parseNumberWithFmtAndLcid(source:    String;
                          fmt:       NumberFormat;
                          lcid:      Integer;
                          errOffset: Integer output): Integer updating;
```

The **parseNumberWithFmtAndLcid** method of the **Real** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid value.

The sign sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional.

Thousands separator sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

Only the first 15 significant digits of the value of the string are stored in the receiver. Any additional digits are rounded into the fifteenth digit and replaced with zeros.

The number of decimal places before and after the decimal point is preserved. Leading zeros before the decimal point and trailing zeros after the decimal point are ignored. For example, **"12345678901234567890"** is parsed and formatted results in **12345678901234600000** being stored in the receiver.

This method supports a maximum of 30 significant digits, whereas the fixed-point representation of a **Real** value can require up to 320 digits.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

rounded

Signature `rounded(): Integer;`

The **rounded** method of the **Real** primitive type returns an integer containing the rounded value of the receiver.

The following code fragments show the use of the **rounded** method.

```
while count > 0 do
    tbl.rowHeight[count] := (tbl.height / tbl.rows).rounded;
    count := count - 1;
endwhile;

write (6.4).Real.rounded;   // outputs 6 [0,1,2,3,4 are rounded down]
write (6.5).Real.rounded;   // outputs 7 [5,6,7,8,9 are rounded up]

write (-6.4).Real.rounded;  // outputs -6 [0,1,2,3,4 are rounded up]
write (-6.5).Real.rounded;  // outputs -7 [5,6,7,8,9 are rounded down]
```

rounded64

Signature `rounded64(): Integer64;`

The **rounded64** method of the **Real** primitive type returns a 64-bit integer containing the rounded value of the receiver.

The following examples show the use of the **rounded64** method.

```
while count > 0 do
    tbl.rowHeight[count] := (tbl.height / tbl.rows).rounded64;
```

```
    count := count - 1;
endwhile;

write (6.4).Real.rounded64;   // outputs 6 [0,1,2,3,4 are rounded down]
write (6.5).Real.rounded64;   // outputs 7 [5,6,7,8,9 are rounded up]

write (-6.4).Real.rounded64;  // outputs -6 [0,1,2,3,4 are rounded up]
write (-6.5).Real.rounded64;  // outputs -7 [5,6,7,8,9 are rounded down]
```

roundedTo

Signature `roundedTo(decimalPlaces: Integer): Real;`

The **roundedTo** method of the **Real** primitive type returns the receiver rounded to the number of decimal places specified in the **decimalPlaces** parameter.

The following example shows the use of the **roundedTo** method.

```
vars
    realValue : Real;
begin
    realValue := 340.5;                    // Defines the variable value
    realValue := (realValue / 27).roundedTo(2);
    write realValue;                       // Outputs 12.61
    realValue := 340.5;                    // Redefines the variable value
    realValue := (realValue / 27).roundedTo(5);
    write realValue;                       // Outputs 12.61111
end;
```

Note As **Real** values are implemented as floating point values, rounding may not return the expected value.

roundedUp

Signature `roundedUp(): Integer;`

The **roundedUp** method of the **Real** primitive type returns an integer containing the receiver rounded up to the nearest whole number.

The following examples show the use of the **roundedUp** method.

```
integerValue := (columnWidth[column] / realValue).roundedUp;
vars
    realValue : Real;
begin
    realValue := 340.5;                   // Defines the variable value
    realValue := (realValue / 20).roundedUp;
    write realValue;                      // Outputs 18
end;
```

roundedUp64

Signature `roundedUp64(): Integer64;`

The **roundedUp64** method of the **Real** primitive type returns a 64-bit integer containing the receiver rounded up to the nearest whole number.

The following examples show the use of the **roundedUp64** method.

```
integerValue := (columnWidth[column] / realValue).roundedUp64;
vars
    realValue : Real;
begin
    realValue := 340.5;                  // Defines the variable value
    realValue := (realValue / 20).roundedUp64;
    write realValue;                     // Outputs 18
end;
```

setByteOrderLocal

Signature `setByteOrderLocal(architecture: Integer): Real;`

The **setByteOrderLocal** method of the **Real** primitive type returns a real that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the **architecture** parameter.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature `setByteOrderRemote(architecture: Integer): Real;`

The **setByteOrderRemote** method of the **Real** primitive type returns a real that has the bytes ordered as required by the remote node indicated by the **architecture** parameter. The bytes of the receiver are assumed to be ordered as required by the local node.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setFloatingPointClassification

Signature `setFloatingPointClassification(classification: Integer) updating;`

The **setFloatingPointClassification** method of the **Real** primitive type sets the receiver to a special value specified by the **classification** parameter.

The following table lists valid values for the **classification** parameter (for which you can use a constant defined for the **Real** primitive type) and the special value that results.

classification Parameter	Resultant Special Value for Receiver
FP_Classification_NegInfinity (2)	Negative infinity.
FP_Classification_NotANumber (1)	A Not a Number value. The result of an invalid operation; for example, attempting to find the square root of a negative number.
FP_Classification_PosInfinity (3)	Positive infinity.

Note There are many possible *Not a Number* (NaN) representations. JADE returns a single NaN representation. A NaN does not compare equal to any floating-point number or NaN, even if the latter has an identical representation, as shown in the following code example.

```
vars
    real : Real;
begin
    real.setFloatingPointClassification(Real.FP_Classification_NotANumber);
    write real = real;  // outputs false
end;
```

sin

Signature `sin(): Real;`

The **sin** method of the **Real** primitive type returns the sine of the receiver. The receiver value represents an angle in degrees, and the resulting value is always in the range **-1** through **1**.

The following example shows the use of the **sin** method.

```
vars
    realValue : Real;
begin
    realValue := 340.5;                 // Defines the variable value
    write realValue.sin;                // Outputs -0.333806859233771
end;
```

sqrt

Signature `sqrt(): Real;`

The **sqrt** method of the **Real** primitive type returns the square root of the receiver.

The following example shows the use of the **sqrt** method.

```
vars
    realValue : Real;
begin
    realValue := 340.5;                 // Defines the variable value
    write realValue.sqrt;               // Outputs 18.4526420872459
end;
```

An exception is raised if the receiver is invalid.

tan

Signature `tan(): Real;`

The **tan** method of the **Real** primitive type returns the tangent of the receiver. The receiver value represents an angle in degrees, and the resulting value is always in the range **-1** through **1**.

The following example shows the use of the **tan** method.

```
vars
    realValue : Real;
begin
    realValue := 340.5;                 // Defines the variable value
    write realValue.tan;                // Outputs -0.354118572530698
end;
```

truncated

Signature `truncated(): Integer;`

The **truncated** method of the **Real** primitive type returns an integer containing the truncated value of the receiver.

The following example shows the use of the **truncated** method.

```
vars
    realValue : Real;
begin
    realValue := 340.56789;             // Defines the variable value
    write realValue.truncated;          // Outputs 340
end;
```

truncated64

Signature `truncated64(): Integer64;`

The **truncated64** method of the **Real** primitive type returns a 64-bit integer containing the truncated value of the receiver.

The following example shows the use of the **truncated64** method.

```
vars
    realValue : Real;
begin
    realValue := 340.56789;             // Defines the variable value
    write realValue.truncated64;        // Outputs 340
end;
```

truncatedTo

Signature `truncatedTo(decimalPlaces: Integer): Real;`

The **truncatedTo** method of the **Real** primitive type returns the receiver truncated to the number of decimal places specified in the **decimalPlaces** parameter. The following example shows the use of the **truncatedTo** method.

```
vars
    realValue : Real;
begin
    realValue := 340.56789;             // Defines the variable value
    write realValue.truncatedTo(3);     // Outputs 340.567
end;
```

userCurrencyFormat

Signature `userCurrencyFormat(fmt: CurrencyFormat): String;`

The **userCurrencyFormat** method of the **Real** primitive type returns a string containing the receiver in the currency format specified in the **fmt** parameter.

To define your currency formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user currency format name with a dollar sign (**$**); for example, **userCurrencyFormat($MyCurrency)**.

You can use the **defineCurrencyFormat** method of the **CurrencyFormat** class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userCurrencyFormatAndLcid

Signature
```
userCurrencyFormatAndLcid(fmt:  CurrencyFormat;
                          lcid: Integer): String;
```

The **userCurrencyFormatAndLcid** method of the **Real** primitive type returns a string containing the receiver in the currency format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature
```
userNumberFormat(fmt: NumberFormat): String;
```

The **userNumberFormat** method of the **Real** primitive type returns a string containing the receiver in the number format specified in the **fmt** parameter.

To define your numeric formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user number format name with a dollar sign (**$**); for example, **userNumberFormat($MyNumber)**.

You can use the **defineNumberFormat** method of the **NumberFormat** class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormatAndLcid

Signature
```
userNumberFormatAndLcid(fmt:  NumberFormat;
                        lcid: Integer): String;
```

The **userNumberFormatAndLcid** method of the **Real** primitive type returns a string containing the receiver in the number format and locale specified in the **fmt** parameter and **lcid** parameter, respectively.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (**0**), the settings of the current locale are used.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

String Type

Use the **String** primitive type to define **String** variables and attributes. A string contains zero or more characters. A **null** string is a string that has a zero length (**""**).

Note A string containing embedded **null** characters can be assigned to a local **String** variable and passed as a **String** parameter. Assigning a string containing embedded **null** characters to a **String** attribute may result in truncation of the string at the first **null** character.

To safely pass a string variable containing embedded **null** characters from JADE to an external method, define the string parameter in JADE as **io** usage.

When you specify a length less than **540** for a **String** attribute, it is embedded. Space is allocated within instances of the class to store a string with a length less than or equal to the specified length.

When you specify a length greater than or equal to **540** or you select the **Maximum Length** check box, which corresponds to 2,147,483,647 characters, for a **String** attribute, it is not embedded. It is stored in a separate variable-length object, a String Large Object (slob), which can store a string with a length less than or equal to the specified length. The amount of storage required for a slob is determined by the value of the string.

String variables can be bounded or unbounded, as shown in the following code fragment.

```
vars
   str1 : String[100]; // Bounded - str1 can store a string with a
                       // length less than or equal to 100 characters
   str2 : String;      // Unbounded - str2 can store a string with a length
                       // less than or equal to 2,147,483,647 characters
```

The ordering relationship of the character values in corresponding positions sets the ordering between two string values. In strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value; for example, *Zs* is greater than *Z*. **Null** strings can be equal only to other **null** strings.

To specify a substring **str[m:n]** of a string **str**, two integers separated by a colon (**:**) character are used. In substrings, the first integer is the start position, and the second integer (following the colon (**:**) character) is the length of the substring or **end**, to indicate the end of the string. The first character is defined as being at position **1**.

If the length of a substring is zero (**0**), a null string (**""**) is returned.

A variable of type **Character** can be used to reference a single character in a string, in effect treating the string as an array of characters, as shown in the following code fragment.

```
vars
   str  : String;
   char : Character;
begin
   str  := "JADE Primitive Types";
   char := str[7];        // seventh character of the string, which is 'r'
```

For details about the methods defined in the **String** primitive type, see "String Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

String Methods

The methods defined in the **String** primitive type are summarized in the following table.

Method	Description
asANSI	Returns ANSI String values as Binary values in a Unicode environment
asDate	Returns a date based on the contents from the receiving string
asGuid	Returns a binary representation based on the class identifier (clsid) of the receiving string
asObject	Returns an object reference based on the contents of the oid-like receiving string and an optional lifetime indication
asOid	Returns an object reference based on the contents of the receiving string
asStringUtf8	Returns a locale-sensitive conversion of the receiving string in UTF8 format
asUuid	Returns a binary by formatting the string as a Universally Unique Identifier (UUID)
base64Decode	Returns a binary value resulting from decoding a Base64-encoded message
bufferAddress	Returns the value of the pointer to the internal buffer as an integer
bufferMemoryAddress	Returns the value of the pointer to the internal buffer as a memory address
compareEql	Returns **true** if the receiver is equal to a specified string
compareGeneric	Returns an integer showing if the receiver is greater than, equal to, or less than a specified character
compareGeq	Returns **true** if the receiver is greater than or equal to a specified string
compareGtr	Returns **true** if the receiver is greater than a specified string
compareLeq	Returns **true** if the receiver is less than or equal to a specified string
compareLss	Returns **true** if the receiver is less than the value of a specified string
compareNeq	Returns **true** if the receiver is not equal to a specified string
compressToBinary	Returns a compressed binary representation of the receiver
display	Returns a string containing the receiver
fillString	Fills the receiving string with the specified string
firstCharToLower	Converts an uppercase first character in the receiving string to lowercase
firstCharToUpper	Converts a lowercase first character in the receiving string to uppercase
getHugeTokens	Returns an array of the tokens not greater than 2047 characters in the receiver
getNextToken	Returns the next token in the receiver
getTokens	Returns an array of the tokens not greater than 62 characters in the receiver
isByte	Returns **true** if the receiver is a string representation of a valid byte value
isDecimal	Returns **true** if the receiver is a string representation of a valid decimal value
isInteger	Returns **true** if the receiver is a string representation of a valid integer value
isInteger64	Returns **true** if the receiver is a string representation of a valid 64-bit integer value

Method	Description
isReal	Returns **true** if the receiver is a string representation of a valid real number
length	Returns the current length of a string variable or attribute
makeString	Returns a string of the specified length filled with the value of the receiver
makeXMLCData	Returns a new string of the receiver prepended with **<![CDATA[** and appended with **]]>**
maxLength	Returns the declared maximum length of a string variable
padBlanks	Returns a copy of the receiving string padded to the specified length with trailing blanks (spaces)
padLeadingZeros	Returns a copy of the receiving string padded to the specified length with leading zeros
plainTextToStringUtf8	Returns a UTF8 string with escaped character sequences replaced by UTF8 characters
pos	Returns an integer containing the position of a substring in the receiver
replaceChar	Replaces all occurrences of a character with another character
reverse	Returns a string containing the reversed characters in the receiving string
reversePos	Returns the position of the last occurrence of a substring in the receiving string
reversePosIndex	Returns the position of the last occurrence of a substring within a substring of the receiving string
scanUntil	Returns a substring of the receiving string starting from the specified index up to (but not including) the first occurrence of any of the specified characters
scanWhile	Returns a substring of the receiving string starting from the specified index up to (but not including) the first occurrence of any character other than the specified characters
toLower	Returns a copy of the receiving string with all uppercase characters converted to lowercase
toUpper	Returns a copy of the receiving string with all lowercase characters converted to uppercase
trimBlanks	Returns a copy of the receiving string with blanks (spaces) trimmed from both ends of the receiver
trimLeft	Returns a copy of the receiving string with the leading blanks (spaces) removed
trimRight	Returns a copy of the receiving string with trailing blanks (spaces) trimmed from the end of the receiver

asANSI

Signature `asANSI(lcid: Integer): Binary;`

The **asANSI** method of the **String** primitive type returns the receiving string converted to a **Binary** value using the character set of the code page for the locale specified by the **lcid** parameter. You can use this method in a Unicode environment to produce ANSI strings in a binary format.

An exception is raised on the first source character that cannot be represented in the code page of the specified locale; for example, a multi-byte Chinese character encountered when the locale is specified as New Zealand.

asDate

Signature `asDate(): Date;`

The **asDate** method of the **String** primitive type returns a date based on the contents of the receiving string. If the receiving string does not contain a valid date, **"invalid"** is returned. The data value must represent one of the following date formats.

- *dd-MMM-yy* (for example, 30-Aug-11)
- *dd/MM/yy* (for example, 30/08/11)
- *MMM dd, yy* (for example, Aug 30, 11)
- *yyyy:MM:dd* (for example, 2011:08:30)

Any non-alphanumeric character can be used as a delimiter.

JADE converts a two-digit year as follows.

- If the current year is equal to or less than 50, all dates default to the current century.
- If the current year is greater than 50, dates that have a year greater than 50 default to the current century.
- If the current year is greater than 50, dates equal to or less than 50 default to the next century.

Note You should always use four-digit years in your applications.

The following example shows the use of the **asDate** method.

```
vars
    dateValue : Date;
begin
    dateValue := "15 May 2010".asDate;   // 15 May 2010
    dateValue := "15-May-2010".asDate;   // 15 May 2010
    dateValue := "15/5/2010".asDate;     // 15 May 2010
    dateValue := "May 15, 2010".asDate;  // 15 May 2010
    dateValue := "2010:5:15".asDate;     // 15 May 2010
    dateValue := "29/2/2011".asDate;     // "*invalid*"
end;
```

asGuid

Signature `asGuid(): Binary;`

The **asGuid** method of the **String** primitive type returns the class identifier (clsid) receiver string as a Globally Unique Identifier (GUID) binary representation. Binary GUID representations of string class identifiers (used in ActiveX control and automation libraries, for example) take less space than a visual string representation. This method raises an exception if the receiver is not a GUID string in the following format.

```
"{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}"
```

See also the **Binary** primitive type **asGuidString** method.

asObject

Signature `asObject(): Object;`

The **asObject** method of the **String** primitive type returns an object reference based on the contents of the oid-like receiving string based on class numbers, followed by an optional lifetime indication.

This method is the inverse of the **Object** class **getObjectStringForObject** method.

The form of the oid-like string can be one of the following.

- *class-number.instId*
- *class-number.instId.parent-class-number*
- *class-number.instId.parent-class-number.subLevel.subId*

The optional lifetime can be **'(t)'**, to indicate a transient object, or **'(s)'**, to indicate a shared transient object. If the optional lifetime is absent, it indicates a persistent object.

The following code fragments are examples of the use of the **asObject** method.

```
// return persistent instance of class number 16401
obj := '16401.1'.asObject;
// return transient instance of class number 16401
obj := '16401.1 (t)'.asObject;
// return shared transient instance of class number 16401
obj := '16401.1 (s)'.asObject;
```

Tip The **asObject** method is useful for debugging from a Workspace method to inspect a specific oid; for example, an oid returned in an exception dialog.

For details about returning an object reference based on the contents of the receiving string, see the **String** primitive type **asOid** method.

asOid

Signature `asOid(): Object;`

The **asOid** method of the **String** primitive type returns an object reference based on the contents of the receiving string.

This method is the inverse of the **Object** class **getOidStringForObject** method.

The following example shows the use of the **asOid** method.

```
begin
    // inspect an instance of an object, in this case 2048.5
    '2048.5'.asOid.inspect;
end;
```

For details about returning an object reference based on the contents of the oid-like receiving string based on class numbers and a following optional lifetime indication, see the **String** primitive type **asObject** method.

asStringUtf8

Signature `asStringUtf8(lcid: Integer): StringUtf8;`

The **asStringUtf8** method of the **String** primitive type returns the receiving string converted to a UTF8 string value.

In an ANSI environment, the conversion uses the character set of the code page for the locale specified by the **lcid** parameter.

asUuid

Signature `asUuid(): Binary;`

The **asUuid** method of the **String** primitive type returns a binary by formatting the string as a Universally Unique Identifier (UUID).

If the string is not formatted as a valid UUID representation (that is, as returned by the **Binary** primitive type **uuidAsString** method), exception 1407 *(Invalid argument passed to method)* is raised.

The code fragment in the following example shows the use of the **asUuid** method.

```
vars
    str : String;
    bin : Binary;
begin
    str := "4dfc912a-b466-01d0-1027-000085823b00";
    bin := str.asUuid();
```

base64Decode

Signature `base64Decode(): Binary;`

The **base64Decode** method of the **String** primitive type returns a **Binary** value resulting from the decoding of a Base64-encoded message. A Base64-encoded message contains characters from the following alphabet.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

The message may also contain line-break characters (**Cr** and **Lf**) inserted by the Base64 encoding algorithm; these are ignored by the decoder, as are any characters that are not in the Base64 alphabet.

Use the **base64Encode** or **base64EncodeNoCrLf** method on the **String** primitive type to encode a binary value using Base64 encoding.

The following example shows the use of the **base64Decode** method.

```
vars
    bin: Binary;
    file: File;
begin
    create file;
    file.fileName := "d:\temp\harry.jpg";
    file.kind := File.Kind_Binary;
    file.open;
    bin := file.readBinary(file.fileLength);
    write 'original length = ' & bin.length.String;
    write 'base64Encode length = ' & bin.base64Encode().length.String;
    write 'base64EncodeNoCrLf length = ' &
                                  bin.base64EncodeNoCrLf().length.String;
    write 'base64Decode length = ' &
                         bin.base64Encode().base64Decode().length.String;
    write 'base64Decode length (from NoCrLf) = ' &
                      bin.base64EncodeNoCrLf().base64Decode.length.String;
     file.close;
```

```
epilog
    delete file;
end;
```

Note The length of an encoded string is about a third longer, even if the string is encoded with carriage-return and line-feed (**Cr** and **Lf**) characters.

bufferAddress

Signature `bufferAddress(): Integer;`

The **bufferAddress** method of the **String** primitive type returns an integer containing the value of the pointer to the internal buffer that contains the string. This value may be required when a JADE **Binary** type value is being mapped to a structured record type for a call to an external function. Call the **bufferAddress** method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

The use of the **bufferAddress** method for the **String** primitive type is similar to that for the **Binary** primitive type. For an example of using the **bufferAddress** method of the **Binary** primitive type to initialize the Windows **SECURITY_DESCRIPTOR** and **SECURITY_ATTRIBUTES** structures, see **bufferAddress**, under "Binary Type".

The code fragment in the following example shows the use of the **bufferAddress** method when copying clipboard data directly into a JADE string.

```
call copyString(str.bufferAddress, locked);
call globalUnlock(locked);
```

Caution Do not use this method to pass the address of a string to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a **jade.exe** (thin client) fault in the called function.

bufferMemoryAddress

Signature `bufferMemoryAddress(): MemoryAddress;`

The **bufferMemoryAddress** method of the **String** primitive type returns a memory address containing the value of the pointer to the internal buffer that contains the string. This value may be required when a JADE **String** type value is being mapped to a structured record type for a call to an external function.

Call the **bufferMemoryAddress** method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

The use of the **bufferMemoryAddress** method for the **String** primitive type is similar to that for the **Binary** primitive type.

For an example of using the **bufferMemoryAddress** method of the **Binary** primitive type to initialize the Windows **SECURITY_DESCRIPTOR** and **SECURITY_ATTRIBUTES** structures, see **bufferMemoryAddress**, under "Binary Type".

The code fragment in the following example shows the use of the **bufferMemoryAddress** method when copying clipboard data directly into a JADE string.

```
call copyString(str.bufferMemoryAddress, locked);
call globalUnlock(locked);
```

Caution Do not use this method to pass the address of a string to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a **jade.exe** (thin client) fault in the called function.

compareEql

Signature

```
compareEql(rhs:          String;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareEql** method of the **String** primitive type returns **true** if the receiver is equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareEql(lhs, true, false, null);

recv.toLower = lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareEql** method.

```
write "Alice".compareEql("alice", true, false, null);   // Outputs true
write "Alice".compareEql("alice", false, false, null);  // Outputs false
```

compareGeneric

Signature

```
compareGeneric(rhs:          String;
               bIgnoreCase:  Boolean;
               bUseLocale:   Boolean;
               locale:       Locale): Integer;
```

The **compareGeneric** method of the **String** primitive type compares the receiver with the value of the **rhs** parameter and returns one of the following values.

Value	Returned if the receiver is …
Negative integer	Less than the right-hand side value represented by the **rhs** parameter
Zero (**0**)	Equal to the right-hand side value represented by the **rhs** parameter
Positive integer	Greater than the right-hand side value represented by the **rhs** parameter

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operators (**<**, **<=**, **=**, **>=**, **>**, **<>**), documented in Chapter 1 of the *JADE Developer's Reference*, use a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareGeneric(lhs, true, false, null);

(recv.toLower>lhs.toLower).Integer - (recv.toLower<lhs.toLower).Integer;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareGeneric** method.

```
vars
    locale : Locale;
begin
    write "alice".compareGeneric("carol", false, false, null); // Outputs -1
    write "bob".compareGeneric("bob", false, false, null);     // Outputs 0
    write "carol".compareGeneric("alice", false, false, null); // Outputs 1
```

```
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
    write "àcute".compareGeneric("zebra", false, false, null); // Outputs 1
    write "àcute".compareGeneric("zebra", false, true, locale);// Outputs -1
```

compareGeq

Signature

```
compareGeq(rhs:          String;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareGeq** method of the **String** primitive type returns **true** if the receiver is greater than or equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**>=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareGeq(lhs, true, false, null);

recv.toLower >= lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareGeq** method.

```
vars
    locale : Locale;
begin
    write "alice".compareGeq("carol", false, false, null);  // Outputs false
    write "bob".compareGeq("bob", false, false, null);      // Outputs true
    write "carol".compareGeq("alice", false, false, null);  // Outputs true
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
```

```
    write "àcute".compareGeq("zebra", false, false, null);  // Outputs true
    write "àcute".compareGeq("zebra", false, true, locale); // Outputs false
```

compareGtr

Signature

```
compareGtr(rhs:          String;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareGtr** method of the **String** primitive type returns **true** if the receiver is greater than the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**>**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareGtr(lhs, true, false, null);

recv.toLower > lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareGtr** method.

```
vars
    locale : Locale;
begin
    write "alice".compareGtr("carol", false, false, null);  // Outputs false
    write "bob".compareGtr("bob", false, false, null);      // Outputs false
    write "carol".compareGtr("alice", false, false, null);  // Outputs true
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
    write "àcute".compareGtr("zebra", false, false, null);  // Outputs true
    write "àcute".compareGtr("zebra", false, true, locale); // Outputs false
```

compareLeq

Signature

```
compareLeq(rhs:          String;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareLeq** method of the **String** primitive type returns **true** if the receiver is less than or equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareLeq(lhs, true, false, null);

recv.toLower <= lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareLeq** method.

```
vars
    locale : Locale;
begin
    write "alice".compareLeq("carol", false, false, null);  // Outputs true
    write "bob".compareLeq("bob", false, false, null);      // Outputs true
    write "carol".compareLeq("alice", false, false, null);  // Outputs false
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
    write "àcute".compareLeq("zebra", false, false, null);  // Outputs false
    write "àcute".compareLeq("zebra", false, true, locale); // Outputs true
```

compareLss

Signature

```
compareLss(rhs:          String;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareLss** method of the **String** primitive type returns **true** if the receiver is less than the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareLss(lhs, true, false, null);

recv.toLower < lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareLss** method.

```
vars
    locale : Locale;
begin
    write "alice".compareLss("carol", false, false, null);  // Outputs true
    write "bob".compareLss("bob", false, false, null);      // Outputs false
    write "carol".compareLss("alice", false, false, null);  // Outputs false
// Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
    write "àcute".compareLss("zebra", false, false, null);  // Outputs false
    write "àcute".compareLss("zebra", false, true, locale); // Outputs true
```

compareNeq

Signature

```
compareNeq(rhs:         String;
           bIgnoreCase: Boolean;
           bUseLocale:  Boolean;
           locale:      Locale): Boolean;
```

The **compareNeq** method of the **String** primitive type returns **true** if the receiver is not equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<>**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareNeq(lhs, true, false, null);

recv.toLower <> lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareNeq** method.

```
write "Alice".compareNeq("alice", true, false, null);   // Outputs false
write "Alice".compareNeq("alice", false, false, null);  // Outputs true
```

compressToBinary

Signature `compressToBinary(typeAndOption: Integer): Binary;`

The **compressToBinary** method of the **String** primitive type returns a compressed binary representation of the string of the receiver using the **ZLIB** compression value specified in the **typeAndOption** parameter, one of the **Binary** type constants listed in the following table.

Constant	Integer Value	Description
Compression_ZLib	1402	String and binary compression to binary using ZLIB level 5 (256*5 + 122)
Compression_ZLibFast	378	String and binary compression to binary using ZLIB level 1 (256*1 + 122)
Compression_ZLibSmall	2426	String and binary compression to binary using ZLIB level 9 (256*9 + 122)

Notes This method adds the type byte to the front of the compressed binary. This type byte is ignored when the value is used in a JADE system but if the data is to be passed to an external library, it is your responsibility to remove the type byte, if necessary.

You cannot concatenate the results of multiple **compressToBinary** method calls.

You must use the **Binary** primitive type **uncompressToString** method to uncompress a binary value from this binary representation.

display

Signature `display(): String;`

The **display** method of the **String** primitive type returns a string enclosed in double quotation marks (**""**) containing the receiver.

If the length of the receiver is zero (**0**), the string **"<null>"** is returned.

fillString

Signature `fillString(string: String) updating;`

The **fillString** method of the **String** primitive type fills the receiving string with repeated copies of the string specified in the **string** parameter up to the length of the receiver.

The following example shows the use of the **fillString** method.

```
vars
    stringValue : String;
begin
    stringValue := 'hello world';
    stringValue.fillString('foo');
    write stringValue;            // Outputs 'foofoofoofo'
end;
```

firstCharToLower

Signature `firstCharToLower() updating;`

The **firstCharToLower** method of the **String** primitive type converts an uppercase first character in the receiving string to lowercase, according to the conventions of the current locale.

The following example shows the use of the **firstCharToLower** method.

```
vars
    stringValue : String;
begin
    stringValue := 'HELLO WORLD';
    stringValue.firstCharToLower;
    write stringValue;           // Outputs 'hELLO WORLD'
end;
```

firstCharToUpper

Signature `firstCharToUpper() updating;`

The **firstCharToUpper** method of the **String** primitive type converts a lowercase first character in the receiving string to uppercase, according to the conventions of the current locale.

The following example shows the use of the **firstCharToUpper** method.

```
vars
    stringValue : String;
begin
    stringValue := 'hello world';
    stringValue.firstCharToUpper;
    write stringValue;           // Outputs 'Hello world'
end;
```

getHugeTokens

Signature `getHugeTokens(): HugeStringArray;`

The **getHugeTokens** method of the **String** primitive type returns an array of the tokens in the receiver that have a length in the range **0** through **2047** characters. For details about tokens, see the **getNextToken** method.

The following example shows the use of the **getHugeTokens** method.

```
vars
    stringValue : String;
    hugeStringArray : HugeStringArray;
begin
    stringValue := 'this:is/a;string';
    hugeStringArray := stringValue.getHugeTokens;
    write hugeStringArray [1];        // Outputs this
    write hugeStringArray [2];        // Outputs is
    write hugeStringArray [3];        // Outputs a
    write hugeStringArray [4];        // Outputs string
end;
```

getNextToken

Signature `getNextToken(int: Integer io): String;`

The **getNextToken** method of the **String** primitive type returns the next token in the receiver; that is, it returns the string from the current value of the **int** parameter to the next delimiter.

Tip As the **String**::**scanUntil** method (which supersedes this **getNextToken** method) provides increased functionality and flexibility, you may want to use that method instead.

The string delimiter can be any of the following characters.

- Colon character (**:**)
- Semicolon character (**;**)
- Stroke character (*/*)
- Double quotation character (**"**)
- Single quotation character (**'**)
- Space
- Tab
- End of string

To define the position from which the next delimiter is returned, specify the starting position in the **int** parameter in a method.

The following example shows the use of the **getNextToken** method.

```
vars
   str   : String;
   token : Integer;
begin
   str   := 'this:is/a;string';
   token := 1;
   write str.getNextToken(token) & ' ' & token.String; // Outputs 'this 6'
   write str.getNextToken(token) & ' ' & token.String; // Outputs 'is 9'
   write str.getNextToken(token) & ' ' & token.String; // Outputs 'a 11'
   write str.getNextToken(token) & ' ' & token.String; // Outputs 'string 17'
end;
```

The **getNextToken** method returns **null** when the end of string is reached.

getTokens

Signature `getTokens(): StringArray;`

The **getTokens** method of the **String** primitive type returns an array of the tokens in the receiver that have a length not greater than 62 characters. For details about tokens, see the **getNextToken** method.

The following example shows the use of the **getTokens** method.

```
vars
    stringValue : String;
```

```
    stringArray : StringArray;
begin
    stringValue := 'this:is/a;string';
    stringArray := stringValue.getTokens;
    write stringArray [1];        // Outputs this
    write stringArray [2];        // Outputs is
    write stringArray [3];        // Outputs a
    write stringArray [4];        // Outputs string
end;
```

isByte

Signature `isByte(): Boolean;`

The **isByte** method of the **String** primitive type returns **true** if the receiver represents a valid byte value; that is, in the range zero (**0**) through **255**; otherwise, it returns **false**.

The following example shows the use of the **isByte** method.

```
vars
    stringValue : String;
begin
    stringValue := '+123';
    write stringValue.isByte;   // Outputs true
    stringValue := '+321';
    write stringValue.isByte;   // Outputs false
end;
```

isDecimal

Signature `isDecimal(): Boolean;`

The **isDecimal** method of the **String** primitive type returns **true** if the receiver represents a valid decimal value; otherwise, it returns **false**.

The following example shows the use of the **isDecimal** method.

```
vars
    stringValue : String;
begin
    stringValue := '+123.456';
    write stringValue.isDecimal;     // Outputs true
    stringValue := '+123,456';
    write stringValue.isDecimal;     // Outputs false
end;
```

isInteger

Signature `isInteger(): Boolean;`

The **isInteger** method of the **String** primitive type returns **true** if the receiver represents an integer value; otherwise, it returns **false**.

The following example shows the use of the **isInteger** method.

```
vars
    stringValue : String;
begin
    stringValue := '+123';
    write stringValue.isInteger;   // Outputs true
    stringValue := '+123.456';
    write stringValue.isInteger;   // Outputs false
end;
```

isInteger64

Signature `isInteger64(): Boolean;`

The **isInteger64** method of the **String** primitive type returns **true** if the receiver represents a 64-bit integer value; otherwise, it returns **false**.

The following example shows the use of the **isInteger64** method.

```
vars
    stringValue : String;
begin
    stringValue := '+123';
    write stringValue.isInteger64;   // Outputs true
    stringValue := '+123.456';
    write stringValue.isInteger64;   // Outputs false
end;
```

isReal

Signature `isReal(): Boolean;`

The **isReal** method of the **String** primitive type returns **true** if the receiver represents a valid real value; otherwise, it returns **false**.

The following example shows the use of the **isReal** method.

```
vars
    stringValue : String;
begin
    stringValue := '+123.456';
    write stringValue.isReal;   // Outputs true
    stringValue := '+123,456';
    write stringValue.isReal;   // Outputs false
end;
```

length

Signature `length(): Integer;`

The **length** method of the **String** primitive type returns the actual length of the value that has been assigned to an embedded **String** property; for example, if you declared a **String** property with length of 30 but the value stored is of length 20, the **length** method returns 20.

The following example shows the use of the **length** method.

```
vars
    stringValue : String;
begin
    stringValue := 'hello world';
    write stringValue.length;     // Outputs 11
end;
```

makeString

Signature `makeString(length: Integer): String;`

The **makeString** method of the **String** primitive type returns a string of the length specified in the **length** parameter filled with the value of the receiver. If the receiver is null (**""**), the returned string is filled with spaces.

If the value of the **length** parameter is less than or equal to zero (**0**), an empty string is returned.

The following example shows the use of the **makeString** method.

```
vars
    strValue : String;
begin
    strValue := "*";
    write strValue.makeString(10);        // Outputs **********
    strValue := "*--";
    write strValue.makeString(10);        // Outputs *--*--*--*
    strValue := null;
    write strValue.makeString(10);        // Outputs           (ten spaces)
end;
```

makeXMLCData

Signature `makeXMLCData(): String;`

The **makeXMLCData** method of the **String** primitive type returns a new string of the receiver prepended with **<![CDATA[**and appended with **]]>**. Note that the receiver is not modified in any way.

The following example is a receiver string.

```
<greeting>Hello, world!</greeting>
```

The returned string of this receiver string is as follows.

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

Call this method for any string for which you do not want the framework to interpret the XML special characters (that is, **<**, **>**, **&**, and **"**).

maxLength

Signature `maxLength(): Integer;`

The **maxLength** method of the **String** primitive type returns the declared maximum length of a string variable. If the string variable maximum length has not been declared, the value of the **Max_UnboundedLength** global constant in the **SystemLimits** category is returned.

The following example shows the use of the **maxLength** method.

```
vars
    stringValue : String[100];
begin
    stringValue := 'hello world';
    write stringValue.maxLength;   // Outputs 100
end;
```

padBlanks

Signature `padBlanks(int: Integer): String;`

The **padBlanks** method of the **String** primitive type returns a string of the length specified in the **int** parameter, consisting of the receiving string padded with appended (trailing) spaces.

If the string is longer than the integer value, it is not truncated but the whole string is returned.

The following example shows the use of the **padBlanks** method.

```
vars
    stringValue : String;
begin
    stringValue := 'Alfonso:';
    write stringValue.padBlanks(10) & '123 Sesame St.';
    // Outputs 'Alfonso:  123 Sesame St.'
end;
```

padLeadingZeros

Signature `padLeadingZeros(int: Integer): String;`

The **padLeadingZeros** method of the **String** primitive type returns a string of the length specified in the **int** parameter, consisting of the receiving string padded with leading zeros.

The following example shows the use of the **padLeadingZeros** method.

```
vars
    stringValue : String;
begin
    stringValue := '123.45';
    write stringValue.padLeadingZeros(10);   // Outputs '0000123.45'
end;
```

plainTextToStringUtf8

Signature `plainTextToStringUtf8(utf8: StringUtf8 output): Integer;`

The **plainTextToStringUtf8** method of the **String** primitive type assigns a UTF8 string to the value of the **utf8** output parameter. The plain text of the receiver is converted to UTF8 format with any escaped character sequences being replaced by the appropriate UTF8 character.

The method returns zero (**0**) if the entire string is converted successfully. If an invalid escaped character sequence is encountered, the **plainTextToStringUtf8** method returns the offset of the first character in error and the **utf8** parameter contains the result of the conversion up to the invalid character.

In the following example, the character sequence **©** is recognized as a valid character © but the character sequence **&cool;** is not recognized. The invalid character starts at position 14.

```
vars
    str : String;
    str8 : StringUtf8;
begin
    str := "&copy; Jade &cool; Software";
    write str.plainTextToStringUtf8(str8);    // 14
    write str8;                               // © Jade
end;
```

pos

Signature

```
pos(substr: String;
    start:  Integer): Integer;
```

The **pos** method of the **String** primitive type returns an integer containing the position of a substring in a string. The substring is specified by the **substr** parameter. The search for the substring begins at the position specified by the **start** parameter.

The **start** parameter must be greater than zero (**0**) and less than or equal to the length of the receiver. If the **substr** or the **start** parameter is greater than the length of the receiver, this method returns zero (**0**). This method returns zero (**0**) if the specified substring is not found.

Note The character search is case-sensitive.

The following example shows the use of the **pos** method.

```
vars
    stringValue : String;
begin
    stringValue := 'position example';
    write stringValue.pos('pos', 1);    // Outputs 1
    write stringValue.pos('pos', 10);   // Outputs 0
end;
```

replaceChar

Signature

```
replaceChar(char:     Character;
            withChar: Character): updating;
```

The **replaceChar** method of the **String** primitive type replaces all occurrences of the character specified in the **char** parameter with the character specified in the **withChar** parameter.

Note The character replacement is case-sensitive.

The following example shows the use of the **replaceChar** method.

```
vars
    stringValue : String;
begin
    stringValue := "zhis example shows character replacement";
    write stringValue;  // Outputs: zhis example shows character replacement
    stringValue.replaceChar("z", "T");
```

```
    write stringValue;  // Outputs: This example shows character replacement
end;
```

reverse

Signature `reverse(): String;`

The **reverse** method of the **String** primitive type returns a string consisting of the receiving string with the position of all characters reversed. For example, a string that contains **"abcde"** is returned as **"edcba"**. The following example shows the use of the **reverse** method.

```
vars
    stringValue : String;
begin
    stringValue := 'abcde';
    write stringValue.reverse;   // Outputs 'edcba'
end;
```

reversePos

Signature `reversePos(substr: String): Integer;`

The **reversePos** method of the **String** primitive type returns the position of the last occurrence of the substring specified in the **substr** parameter in the receiving string.

Note The character search is case-sensitive.

The following example shows the use of the **reversePos** method.

```
vars
    stringValue : String;
begin
    stringValue := "Reverse position example";
    write stringValue.reversePos('pos');      // Outputs 9
end;
```

reversePosIndex

Signature
```
reversePosIndex(substr: String;
                index:  Integer): Integer;
```

The **reversePosIndex** method of the **String** primitive type returns the position of the last occurrence of the substring specified in the **substr** parameter, in a string formed from the first character of the receiving string up to (and including) the character position specified in the **index** parameter.

Notes The character search is case-sensitive.

The value of the **index** parameter cannot exceed the length of the receiving string.

The following example shows the use of the **reversePosIndex** method.

```
vars
    stringValue : String;
    count       : Integer;
begin
```

```
        stringValue := "car->taxi->bus->train";
        count := stringValue.length;
        while count > 0 do
            count := stringValue.reversePosIndex('->', count);
            write count;
            count := count - 1;
        endwhile;
        // Outputs 15
        // Outputs 10
        // Outputs 4
        // Outputs 0
    end;
```

This method returns zero (**0**) if the specified substring is not found.

scanUntil

Signature

```
scanUntil(delimiters: String;
          index:      Integer io): String;
```

The **scanUntil** method of the **String** primitive type returns a substring of the receiving string starting from the index specified in the **index** parameter up to (but not including) the first occurrence of any of the characters specified in the **delimiters** parameter.

The index of the delimiting character is returned in the second parameter.

If a delimiting character is not found, the return value is the remainder of the receiving string (from the specified index) and an index value of zero (**0**) is returned in the second parameter.

Note The character search is case-sensitive.

The following example shows the use of the **scanUntil** method.

```
    vars
        stringValue : String;
        pos         : Integer;
    begin
        stringValue := "this:is/a;string";
        pos := 1;
        write stringValue.scanUntil(":/;", pos);  // Outputs this
        pos := pos+1;
        write stringValue.scanUntil(":/;", pos);  // Outputs is
    end;
```

See also the **String** primitive type **getNextToken** and **scanWhile** methods.

scanWhile

Signature

```
scanWhile(characters: String;
          index:      Integer io): String;
```

The **scanWhile** method of the **String** primitive type returns a substring of the receiving string starting from the index specified in the **index** parameter up to (but not including) the first occurrence of any character other than the characters specified in the **characters** parameter.

The index of the delimiting character is specified in the second parameter.

If a delimiting character is not found, the return value is the remainder of the string and an index value of zero (**0**) is returned in the second parameter, as shown in the following example.

```
vars
    i: Integer;
    s: String;
begin
    i:= 3;
    s:= '0246'.scanWhile('0123456789', i);
    write '<' & s & '> ' & i.String;       // outputs <46> 0, not <> 0
end;
```

Notes The character search is case-sensitive.

The delimiting character is any character that is *not* specified in the **characters** parameter.

The following example shows the use of the **scanWhile** method.

```
vars
    stringValue : String;
    pos         : Integer;
begin
    stringValue := "this:is/a;string";
    pos := 1;
    write stringValue.scanWhile("abcdefghijklmnopqrstuvwxyz", pos);
    // Outputs this
    pos := pos+1;
    write stringValue.scanWhile("abcdefghijklmnopqrstuvwxyz", pos);
    // Outputs is
end;
```

See also the **String** primitive type **scanUntil** method.

toLower

Signature `toLower(): String;`

The **toLower** method of the **String** primitive type returns a copy of the receiving string with all uppercase characters converted to lowercase, according to the conventions of the current locale.

The following example shows the use of the **toLower** method.

```
vars
    stringValue : String;
begin
    stringValue := "UPPERCASE TEXT CAN LOOK THREATENING";
    write stringValue.toLower;
    // Outputs uppercase text can look threatening
end;
```

toUpper

Signature `toUpper(): String;`

The **toUpper** method of the **String** primitive type returns a copy of the receiving string with all lowercase characters converted to uppercase, according to the conventions of the current locale.

The following example shows the use of the **toUpper** method.

```
vars
    stringValue : String;
begin
    stringValue := "lowercase";
    write stringValue.toUpper;   // Outputs LOWERCASE
end;
```

trimBlanks

Signature `trimBlanks(): String;`

The **trimBlanks** method of the **String** primitive type returns a copy of the receiving string with blanks (spaces) trimmed from both ends of the receiver.

The following example shows the use of the **trimBlanks** method.

```
vars
    stringValue : String;
begin
    stringValue := '     some text     '.trimBlanks;
    write stringValue;            // Outputs 'some text'
end;
```

trimLeft

Signature `trimLeft(): String;`

The **trimLeft** method of the **String** primitive type returns a copy of the receiving string with leading blanks (spaces) removed.

The following example shows the use of the **trimLeft** method.

```
vars
    stringValue : String;
begin
    stringValue := '     some text     '.trimLeft;
    write stringValue;            // Outputs 'some text     '
end;
```

trimRight

Signature `trimRight(): String;`

The **trimRight** method of the **String** primitive type returns a copy of the receiving string with trailing blanks (spaces) trimmed from the end of the receiver.

The following example shows the use of the **trimRight** method.

```
vars
    stringValue : String;
begin
    stringValue := '     some text     '.trimRight;
    write stringValue;            // Outputs '     some text'
end;
```

StringUtf8 Type

Use the **StringUtf8** primitive type to define **StringUtf8** variables and attributes; that is, strings that have been encoded in the UTF8 format. This allows all valid Unicode characters to be used even in an ANSI system. A character string contains zero or more characters. A **null** string is a string that has a zero length (**""**). You can access characters in a string as components of an array.

When you specify a length less than **540** for a **StringUtf8** attribute, it is embedded. Space is allocated within instances of the class to store a string with a length less than or equal to the specified length.

When you specify a length greater than or equal to **540** or you select the **Maximum Length** check box (which corresponds to 2,147,483,647 characters) for a **StringUtf8** attribute, it is not embedded. It is stored in a separate variable-length object, a String Large Object (slob), which can store a string with a length less than or equal to the specified length. The amount of storage required for a slob is determined by the value of the string.

StringUtf8 variables can be bounded or unbounded, as shown in the following code fragment.

```
vars
   s1 : StringUtf8[100]; // Bounded - s1 can store a string with a
                         // length less than or equal to 100 characters
   s2 : StringUtf8;      // Unbounded - s2 can store a string with a length
                         // less than or equal to 2,147,483,647 characters
```

The ordering relationship of the character values in corresponding positions sets the ordering between two string values. In strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value; for example, *Zs* is greater than *Z*. **Null** strings can be equal only to other **null** strings.

To specify a substring **str[m:n]** of a string **str**, two integers separated by a colon (**:**) character are used. The first integer indicates the start position and the second integer is the length of the substring. In place of the second integer, **end** indicates the substring extends to the end of the string. For a substring starting at the first character of the string, the first integer would be **1**.

If the length of a substring is zero (**0**), a null string (**""**) is returned.

Note You can ignore the fact that a non-ASCII character in a UTF8 string requires more than one byte of storage, as the start position and length integers are based on character positions rather than on byte positions.

A **StringUtf8** literal is enclosed in double (**""**) or single (**''**) quotation marks, and is usually preceded by an at sign (**@**), as shown in the following example.

```
stringUtf8 := @"Jade Software";
```

If all the characters are US-ASCII characters, as in the preceding example, the **@** sign is optional.

The **StringUtf8** literal can contain a non-US-ASCII character, by enclosing a value representing the character between an ampersand (**&**) character and a semicolon (**;**) character, as shown in the following examples.

```
stringUtf8 := @"Copyright &copy; Jade Software";

stringUtf8 := @"Copyright &#169; Jade Software";

stringUtf8 := @"Copyright &#xA9; Jade Software";
```

In the first example, a character entity reference as defined in the HTML 4 standard is used. In the second and third examples, the value of the Unicode code point of the character in decimal and in hexadecimal is used.

A variable of type **StringUtf8** can be used to reference a single character in a string, in effect treating the string as an array of one-character UTF8 strings, as shown in the following code fragment.

```
vars
   str1 : StringUtf8;
   str2 : StringUtf8;
begin
   str1 := @"JADE Primitive Types";
   str2 := str1[7];       // UTF8 string consisting of seventh character 'r'
```

For details about the methods defined in the **StringUtf8** primitive type, see "StringUtf8 Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

StringUtf8 Methods

The methods defined in the **StringUtf8** primitive type are summarized in the following table.

Method	Description
asANSI	Returns ANSI String values as Binary values in a Unicode environment
asDate	Returns a date based on the contents of the receiving UTF8 string
asPlainText	Returns a string with a character entity escape sequence replacing each non-US-ASCII character
asString	Returns multiple-byte or code page-sensitive values as String values in an ANSI environment
bufferMemoryAddress	Returns the value of the pointer to the internal buffer as a memory address
byteOffsetFromCharacterIndex	Returns the byte offset for a specified index of a character within the receiving UTF8 string
characterIndexFromByteOffset	Returns the index of a character for a specified byte offset within the receiving UTF8 string
compareEql	Returns **true** if the receiver is equal to a specified UTF8 string
compareGeneric	Returns an integer showing if the receiver is greater than, equal to, or less than a UTF8 string
compareGeq	Returns **true** if the receiver is greater than or equal to a specified UTF8 string
compareGtr	Returns **true** if the receiver is greater than a specified UTF8 string
compareLeq	Returns **true** if the receiver is less than or equal to a specified UTF8 string
compareLss	Returns **true** if the receiver is less than the value of a specified UTF8 string
compareNeq	Returns **true** if the receiver is not equal to a specified UTF8 string
compressToBinary	Returns a compressed binary representation of a UTF8 string
display	Returns the receiver encoded in the character set of the default code page for the current locale
firstCharToLower	Converts an uppercase first character in the receiving UTF8 string to lowercase
firstCharToUpper	Converts a lowercase first character in the receiving UTF8 string to uppercase

Method	Description
isValid	Returns **true** if the receiver represents a valid UTF8 string
length	Returns the current number of characters in the receiver
maxLength	Returns the declared maximum number of characters in the receiver
padBlanks	Returns a copy of the receiving string padded to the specified length with trailing blanks (spaces)
padLeadingZeros	Returns a copy of the receiving string padded to the specified length with leading zeros
pos	Returns an integer containing the character index of the start of a UTF8 substring in the receiver
posUsingByteOffset	Returns an integer containing the byte offset of the start of a UTF8 substring in the receiver
replaceChar	Replaces all occurrences of a character with another character
reverse	Returns a UTF8 string containing the reversed characters in the receiving UTF8 string
reversePos	Returns the position of the last occurrence of a substring in the receiving UTF8 string
reversePosIndex	Returns the position of the last occurrence of a substring within a substring of the UTF8 string
scanUntil	Returns a substring of the receiving UTF8 string starting from the specified index up to (but not including) the first occurrence of any of the specified characters
scanWhile	Returns a substring of the receiving UTF8 string starting from the specified index up to (but not including) the first occurrence of any character other than the specified characters
size	Returns the number of bytes required to store the receiver excluding the end-of-string character
substringAtByteOffset	Returns a UTF8 substring with a specified length starting with the character at a specified byte offset
toLower	Returns a copy of the receiving UTF8 string with all uppercase characters converted to lowercase
toUpper	Returns a copy of the receiving UTF8 string with all lowercase characters converted to uppercase
trimBlanks	Returns a copy of the receiving UTF8 string with blanks (spaces) trimmed from both ends of the receiver
trimLeft	Returns a copy of the receiving UTF8 string with the leading blanks (spaces) removed
trimRight	Returns a copy of the receiving UTF8 string with trailing blanks (spaces) trimmed from the end of receiver

asANSI

Signature `asANSI(lcid: Integer): Binary;`

The **asANSI** method of the **StringUtf8** primitive type returns the receiving UTF8 string converted to a **Binary** value using the character set of the code page for the locale specified by the **lcid** parameter.

You can use this method in a Unicode environment to produce ANSI strings in a binary format.

asDate

Signature `asDate(): Date;`

The **asDate** method of the **StringUtf8** primitive type returns a date based on the contents of the receiving string.

If the receiving string does not contain a valid date, **"invalid"** is returned.

The data value must represent one of the following date formats.

- *dd-MMM-yy* (for example, 30-Aug-11)
- *dd/MM/yy* (for example, 30/08/11)
- *MMM dd, yy* (for example, Aug 30, 11)
- *yyyy:MM:dd* (for example, 2011:08:30)

Any non-alphanumeric character can be used as a delimiter.

JADE converts a two-digit year as follows.

- If the current year is equal to or less than 50, all dates default to the current century.
- If the current year is greater than 50, dates that have a year greater than 50 default to the current century.
- If the current year is greater than 50, dates equal to or less than 50 default to the next century.

Note Always use four-digit years in your applications.

The following example shows the use of the **asDate** method.

```
vars
    dateValue : Date;
begin
    dateValue := @"15 May 2010".asDate;   // 15 May 2010
    dateValue := @"15-May-2010".asDate;   // 15 May 2010
    dateValue := @"15/5/2010".asDate;     // 15 May 2010
    dateValue := @"May 15, 2010".asDate;  // 15 May 2010
    dateValue := @"2010:5:15".asDate;     // 15 May 2010
    dateValue := @"29/2/2011".asDate;     // "*invalid*"
end;
```

asPlainText

Signature `asPlainText(): String;`

The **asPlainText** method of the **StringUtf8** primitive type returns a string containing the US-ASCII characters from the receiving UTF8 string with the non-US-ASCII characters replaced with a character entity escape sequence using an entity name if possible; otherwise a hexadecimal value.

ASCII control characters (excluding carriage returns, line feeds, and tabs) are converted to hexadecimal escape sequences. The ampersand and semicolon characters are converted to **&** and **;** respectively.

The following code example shows the difference between using the **asPlainText** method and converting to a **String** value.

```
vars
    str8: StringUtf8;
begin
    str8 := @"Copyright &copy;;";
    write str8.asPlainText;              // outputs "Copyright &copy;&semi;"
    write str8.String;                   // outputs "Copyright ©;"
end;
```

asString

Signature `asString(lcid: Integer): String;`

The **asString** method of the **StringUtf8** primitive type returns the receiving UTF8 string converted to a **String** value.

In an ANSI environment, the conversion uses the character set of the code page for the locale specified by the **lcid** parameter.

bufferMemoryAddress

Signature `bufferMemoryAddress(): MemoryAddress;`

The **bufferMemoryAddress** method of the **StringUtf8** primitive type returns a memory address containing the value of the pointer to the internal buffer that contains the UTF8 string. This value may be required when a JADE **StringUtf8** type value is being mapped to a structured record type for a call to an external function. Call the **bufferMemoryAddress** method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

The use of the **bufferMemoryAddress** method for the **StringUtf8** primitive type is similar to that for the **Binary** primitive type.

For an example of using the **bufferMemoryAddress** method of the **Binary** primitive type to initialize the Windows **SECURITY_DESCRIPTOR** and **SECURITY_ATTRIBUTES** structures, see **bufferMemoryAddress**, under "Binary Type".

The code fragment in the following example shows the use of the **bufferMemoryAddress** method when copying clipboard data directly into a JADE string.

```
call copyString(stringUtf8.bufferMemoryAddress, locked);
call globalUnlock(locked);
```

Caution Do not use this method to pass the address of a UTF8 string to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a **jade.exe** (thin client) fault in the called function.

byteOffsetFromCharacterIndex

Signature `byteOffsetFromCharacterIndex(index: Integer): Integer;`

The **byteOffsetFromCharacterIndex** method of the **StringUtf8** primitive type returns the byte offset for the character specified by the **index** parameter within the receiving UTF8 string.

In the following code example, the first character of the UTF8 string **str8** requires two bytes with the remaining four characters requiring one byte each. The second character therefore starts at offset three (**3**).

```
vars
    str8 : StringUtf8;
begin
    str8 := @"&copy;2007";
    write str8.byteOffsetFromCharacterIndex(1);     // writes 1
    write str8.byteOffsetFromCharacterIndex(2);     // writes 3
    write str8.byteOffsetFromCharacterIndex(3);     // writes 4
    write str8.byteOffsetFromCharacterIndex(4);     // writes 5
    write str8.byteOffsetFromCharacterIndex(5);     // writes 6
end;
```

characterIndexFromByteOffset

Signature `characterIndexFromByteOffset(offset: Integer): Integer;`

The **characterIndexFromByteOffset** method of the **StringUtf8** primitive type returns the index of the character that starts at the byte offset specified in the **byte** parameter within the receiving UTF8 string, or after that offset; that is, the method scans from the offset position forwards to find the next character. If there is no next character, an exception is raised.

In the following code example, the two characters of the string **str8** require two bytes and three bytes in the UTF8 encoding. The first character starts at offset one (**1**) and the second character at offset three (**3**).

```
vars
    str8: StringUtf8;
begin
    str8 := @"&copy;&euro;";
    write str8.characterIndexFromByteOffset(1);     // writes 1
    write str8.characterIndexFromByteOffset(2);     // writes 2
    write str8.characterIndexFromByteOffset(3);     // writes 2
    write str8.characterIndexFromByteOffset(4);     // raises 1413 exception
    write str8.characterIndexFromByteOffset(5);     // raises 1413 exception
end;
```

compareEql

Signature

```
compareEql(rhs:          StringUtf8;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareEql** method of the **StringUtf8** primitive type returns **true** if the receiver is equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareEql(lhs, true, false, null);

recv.toLower = lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

compareGeneric

Signature

```
compareGeneric(rhs:          StringUtf8;
               bIgnoreCase:  Boolean;
               bUseLocale:   Boolean;
               locale:       Locale): Integer;
```

The **compareGeneric** method of the **StringUtf8** primitive type compares the receiver with the value of the **rhs** parameter and returns one of the following values.

Value	Returned if the receiver is …
Negative integer	Less than the right-hand side value represented by the **rhs** parameter
Zero (**0**)	Equal to the right-hand side value represented by the **rhs** parameter
Positive integer	Greater than the right-hand side value represented by the **rhs** parameter

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operators (**<**, **<=**, **=**, **>=**, **>**, **<>**), documented in Chapter 1 of the *JADE Developer's Reference*, use a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

compareGeq

Signature

```
compareGeq(rhs:          StringUtf8;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareGeq** method of the **StringUtf8** primitive type returns **true** if the receiver is greater than or equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**>=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.

compareGtr

Signature

```
compareGtr(rhs:         StringUtf8;
           bIgnoreCase: Boolean;
           bUseLocale:  Boolean;
           locale:      Locale): Boolean;
```

The **compareGtr** method of the **StringUtf8** primitive type returns **true** if the receiver is greater than the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**>**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

compareLeq

Signature

```
compareLeq(rhs:         StringUtf8;
           bIgnoreCase: Boolean;
           bUseLocale:  Boolean;
           locale:      Locale): Boolean;
```

The **compareLeq** method of the **StringUtf8** primitive type returns **true** if the receiver is less than or equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<=**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

compareLss

Signature

```
compareLss(rhs:          StringUtf8;
           bIgnoreCase:  Boolean;
           bUseLocale:   Boolean;
           locale:       Locale): Boolean;
```

The **compareLss** method of the **StringUtf8** primitive type returns **true** if the receiver is less than the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

compareNeq

Signature

```
compareNeq(rhs:         StringUtf8;
           bIgnoreCase: Boolean;
           bUseLocale:  Boolean;
           locale:      Locale): Boolean;
```

The **compareNeq** method of the **StringUtf8** primitive type returns **true** if the receiver is not equal to the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note The relational binary comparison operator (**<>**), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

compressToBinary

Signature `compressToBinary(typeAndOption: Integer): Binary;`

The **compressToBinary** method of the **StringUtf8** primitive type returns a compressed binary representation of the UTF8 string of the receiver using the **ZLIB** compression routine specified in the **typeAndOption** parameter, using one of the **Binary** type constants listed in the following table.

Constant	Integer Value	Description
Compression_ZLib	1402	String and binary compression to binary using ZLIB level 5 (256*5 + 122)
Compression_ZLibFast	378	String and binary compression to binary using ZLIB level 1 (256*1 + 122)
Compression_ZLibSmall	2426	String and binary compression to binary using ZLIB level 9 (256*9 + 122)

Notes This method adds the type byte to the front of the compressed binary. This type byte is ignored when the value is used in a JADE system but if the data is to be passed to an external library, it is your responsibility to remove the type byte, if necessary.

You cannot concatenate the results of multiple **compressToBinary** method calls.

You must use the **Binary** type **uncompressToStringUtf8** method to uncompress a binary value from this binary representation.

display

Signature `display(): String;`

The **display** method of the **StringUtf8** primitive type returns a string enclosed in double quotation marks (**""**) containing the receiver encoded in the ANSI character set of the default code page for the current locale.

If the length of the receiver is zero (**0**), the string **"<null>"** is returned.

firstCharToLower

Signature `firstCharToLower() updating;`

The **firstCharToLower** method of the **StringUtf8** primitive type converts an uppercase first character in the receiving string to lowercase, according to the conventions of the current locale.

The following example shows the use of the **firstCharToLower** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := @'HELLO WORLD';
    str8.firstCharToLower;
    write str8;           // Outputs 'hELLO WORLD'
end;
```

firstCharToUpper

Signature `firstCharToUpper() updating;`

The **firstCharToUpper** method of the **StringUtf8** primitive type converts a lowercase first character in the receiving string to uppercase, according to the conventions of the current locale. The following example shows the use of the **firstCharToUpper** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := @'hello world';
    str8.firstCharToUpper;
    write str8;           // Outputs 'Hello world'
end;
```

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **StringUtf8** primitive type returns **true** if the receiver is a correctly formatted UTF8 string.

length

Signature `length(): Integer;`

The **length** method of the **StringUtf8** primitive type returns the current number of characters of the receiver. The following example shows the use of the **length** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := @"hello world";
    write str8.length;     // Outputs 11
end;
```

maxLength

Signature `maxLength(): Integer;`

The **maxLength** method of the **StringUtf8** primitive type returns the declared maximum length of the receiver. If the maximum length of a **StringUtf8** variable has not been declared, **-1** is returned.

The following example shows the use of the **maxLength** method.

```
vars
    str8 : StringUtf8[100];
begin
    str8 := @"hello world";
    write str8.maxLength;   // Outputs 100
end;
```

padBlanks

Signature `padBlanks(int: Integer): StringUtf8;`

The **padBlanks** method of the **StringUtf8** primitive type returns a string of the length specified in the **int** parameter, consisting of the receiving string padded with appended (trailing) spaces.

If the string is longer than the integer value, it is not truncated but the whole string is returned.

The following example shows the use of the **padBlanks** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := @'Alfonso:';
    write str8.padBlanks(10) & '123 Sesame St.';
    // Outputs 'Alfonso:  123 Sesame St.'
end;
```

padLeadingZeros

Signature `padLeadingZeros(int: Integer): StringUtf8;`

The **padLeadingZeros** method of the **StringUtf8** primitive type returns a string of the length specified in the **int** parameter, consisting of the receiving string padded with leading zeros.

The following example shows the use of the **padLeadingZeros** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := @'123.45';
    write str8.padLeadingZeros(10);   // Outputs '0000123.45'
end;
```

pos

Signature
```
pos(substr: StringUtf8;
    start:  Integer): Integer;
```

The **pos** method of the **StringUtf8** primitive type returns an integer containing the character index of the start of a substring within a string. The substring is specified by the **substr** parameter. The search for the substring begins at the character index specified by the **start** parameter.

The **start** parameter must be greater than zero (**0**) and less than or equal to the length of the receiver. If the **substr** or the **start** parameter is greater than the length of the receiver, this method returns zero (**0**).

This method returns zero (**0**) if the specified substring is not found.

Note The character search is case-sensitive.

The following example shows the use of the **pos** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := @'position example';
```

```
    write str8.pos('pos', 1);    // Outputs 1
    write str8.pos('pos', 10);   // Outputs 0
end;
```

posUsingByteOffset

Signature

```
posUsingByteOffset(substr: StringUtf8;
                   start:  Integer): Integer;
```

The **posUsingByteOffset** method of the **StringUtf8** primitive type returns an integer containing the byte offset of the start of a UTF8 substring within the receiver. The substring is specified by the **substr** parameter.

The search for the substring begins at the byte offset specified by the **start** parameter.

The **start** parameter must be greater than zero (**0**) and less than or equal to the number of bytes in the receiver. If the **substr** or the **start** parameter is greater than the number of bytes in the receiver, this method returns zero (**0**).

This method returns zero (**0**) if the specified substring is not found.

Note The character search is case-sensitive.

In the following code example, the two characters of the string **str8** require three bytes and two bytes in the UTF8 encoding. The first character starts at offset one (**1**) and the second character at offset four (**4**).

```
vars
    str8 : StringUtf8;
begin
    str8 := @"&euro;&copy;";
    write str8.posUsingByteOffset(@"&copy;", 3);      // Outputs 4
end;
```

replaceChar

Signature

```
replaceChar(char:     Character;
            withChar: Character): updating;
```

The **replaceChar** method of the **StringUtf8** primitive type replaces all occurrences of the character specified in the **char** parameter with the character specified in the **withChar** parameter.

Note The character replacement is case-sensitive.

The following example shows the use of the **replaceChar** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := "zhis example shows character replacement";
    write str8;  // Outputs: zhis example shows character replacement
    str8.replaceChar("z", "T");
    write str8;  // Outputs: This example shows character replacement
end;
```

reverse

Signature `reverse(): StringUtf8;`

The **reverse** method of the **StringUtf8** primitive type returns a string consisting of the receiving string with the position of all characters reversed. For example, a string that contains **"abcde"** is returned as **"edcba"**.

The following example shows the use of the **reverse** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := 'abcde';
    write str8.reverse;   // Outputs 'edcba'
end;
```

reversePos

Signature `reversePos(substr: StringUtf8): Integer;`

The **reversePos** method of the **StringUtf8** primitive type returns the position of the last occurrence of the substring specified in the **substr** parameter in the receiving string.

Note The character search is case-sensitive.

The following example shows the use of the **reversePos** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := "Reverse position example";
    write str8.reversePos('pos');      // Outputs 9
end;
```

reversePosIndex

Signature
```
reversePosIndex(substr: StringUtf8;
                index:  Integer): Integer;
```

The **reversePosIndex** method of the **StringUtf8** primitive type returns the position of the last occurrence of the substring specified in the **substr** parameter, in a string formed from the first character of the receiving string up to (and including) the character position specified in the **index** parameter.

Notes The character search is case-sensitive.

The value of the **index** parameter cannot exceed the length of the receiving string.

The following example shows the use of the **reversePosIndex** method.

```
vars
    str8  : StringUtf8;
    count : Integer;
begin
    str8 := "car->taxi->bus->train";
    count := str8.length;
    while count > 0 do
```

```
            count := str8.reversePosIndex('->', count);
            write count;
            count := count - 1;
        endwhile;
        // Outputs 15
        // Outputs 10
        // Outputs 4
        // Outputs 0
    end;
```

This method returns zero (**0**) if the specified substring is not found.

scanUntil

Signature

```
scanUntil(delimiters: StringUtf8;
          index:      Integer io): StringUtf8;
```

The **scanUntil** method of the **StringUtf8** primitive type returns a UTF8 substring of the receiving string starting from the index specified in the **index** parameter up to (but not including) the first occurrence of any of the characters specified in the **delimiters** parameter.

The index of the delimiting character is returned in the second parameter. If a delimiting character is not found, the return value is the remainder of the receiving string (from the specified index) and an index value of zero (**0**) is returned in the second parameter.

Note The character search is case-sensitive.

The following example shows the use of the **scanUntil** method.

```
    vars
        str8 : StringUtf8;
        pos  : Integer;
    begin
        str8 := @"this:is/a;string";
        pos := 1;
        write str8.scanUntil(@":/;", pos);  // Outputs this
        pos := pos + 1;
        write str8.scanUntil(@":/;", pos);  // Outputs is
    end;
```

scanWhile

Signature

```
scanWhile(characters: StringUtf8;
          index:      Integer io): StringUtf8;
```

The **scanWhile** method of the **StringUtf8** primitive type returns a UTF8 substring of the receiving string starting from the index specified in the **index** parameter up to (but not including) the first occurrence of any character other than the characters specified in the **characters** parameter.

The index of the delimiting character is specified in the second parameter. If a delimiting character is not found, the return value is the remainder of the string and an index value of zero (**0**) is returned in the second parameter, as shown in the following example.

```
    vars
        index : Integer;
        str8  : StringUtf8;
```

```
begin
    index := 3;
    str8 := @'0246'.scanWhile(@'0123456789', index);
    write '<' & str8 & '> ' & index.StringUtf8;
    // outputs <46> 0, not <> 0
end;
```

Notes The character search is case-sensitive.

The delimiting character is any character that is *not* specified in the **characters** parameter.

The following example shows the use of the **scanWhile** method.

```
vars
    str8  : StringUtf8;
    index : Integer;
begin
    str8 := @"this:is/a;string";
    index := 1;
    write str8.scanWhile(@"abcdefghijklmnopqrstuvwxyz", index);
    // Outputs this
    index := index + 1;
    write str8.scanWhile(@"abcdefghijklmnopqrstuvwxyz", index);
    // Outputs is
end;
```

size

Signature `size(): Integer;`

The **size** method of the **StringUtf8** primitive type returns the number of bytes required to store the receiver. Note that this value does not include the null character that marks the end of the string.

The following example shows the use of the **size** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := @"JADE";
    write str8.size;            // 4 bytes - one for each ASCII character
    str8 := @"&copy;";
    write str8.size;            // 2 bytes for the copyright symbol
    str8 := @"&euro;";
    write str8.size;            // 3 bytes for the euro currency symbol
end;
```

substringAtByteOffset

Signature
```
substringAtByteOffset(offset: Integer;
                      length: Integer): StringUtf8;
```

The **substringAtByteOffset** method of the **StringUtf8** primitive type returns a UTF8 substring beginning with the character that starts at the byte offset specified by the value of the **offset** parameter within the receiving UTF8 string or after that offset; that is, the method scans from the offset position forwards to find the next character.

The value of the **length** parameter determines the maximum number of characters that can be returned in the UTF8 substring.

In the following code example, the first character of the string **str8** requires three bytes for UTF8 encoding. The first character starts at byte offset one (**1**) and the second character at byte offset four (**4**).

```
vars
    str8: StringUtf8;
begin
    str8 := @"&euro;xyz";
    write str8.substringAtByteOffset(3,2);   // writes xy
end;
```

toLower

Signature `toLower(): StringUtf8;`

The **toLower** method of the **StringUtf8** primitive type returns a copy of the receiving string with all uppercase characters converted to lowercase, according to the conventions of the current locale.

The following example shows the use of the **toLower** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := "UPPERCASE TEXT CAN LOOK THREATENING";
    write str8.toLower;
    // Outputs uppercase text can look threatening
end;
```

toUpper

Signature `toUpper(): StringUtf8;`

The **toUpper** method of the **StringUtf8** primitive type returns a copy of the receiving string with all lowercase characters converted to uppercase, according to the conventions of the current locale.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The following example shows the use of the **toUpper** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := "lowercase";
    write str8.toUpper;   // Outputs LOWERCASE
end;
```

trimBlanks

Signature `trimBlanks(): StringUtf8;`

The **trimBlanks** method of the **StringUtf8** primitive type returns a copy of the receiving string with blanks (spaces) trimmed from both ends of the receiver.

The following example shows the use of the **trimBlanks** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := '     some text     ';
    write str8.trimBlanks;          // Outputs 'some text'
end;
```

trimLeft

Signature `trimLeft(): StringUtf8;`

The **trimLeft** method of the **StringUtf8** primitive type returns a copy of the receiving string with leading blanks (spaces) removed.

The following example shows the use of the **trimLeft** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := '     some text     ';
    write str8.trimLeft;            // Outputs 'some text     '
end;
```

trimRight

Signature `trimRight(): StringUtf8;`

The **trimRight** method of the **StringUtf8** primitive type returns a copy of the receiving string with trailing blanks (spaces) trimmed from the end of the receiver.

The following example shows the use of the **trimRight** method.

```
vars
    str8 : StringUtf8;
begin
    str8 := '     some text     ';
    write str8.trimRight;           // Outputs '     some text'
end;
```

Time Type

Use the **Time** primitive type to declare a variable representing the time of day since midnight to the nearest millisecond.

If you declare a **Time** primitive type variable in your method, it is initialized with the current time each time the method that declares the variable is invoked. Object properties of **Time** primitive type are initialized to **null**.

In JADE thin client mode, local variables of type **Time** are always initialized to the time relative to the presentation client.

The following example shows the use of the **Time** primitive type.

```
testTime();
vars
    time     : Time;
    h,m,s,ms : Integer;
begin
    h  := 15;
    m  := 39;
    s  := 06;
    ms := 45;
    time.setTime(h, m, s, ms);
    write time;               // Outputs 15:39:06
end;
```

The following table lists valid operations for the **Time** primitive type.

Expression	Expression Type
time-expression + integer-expression	(time)
time-expression - integer-expression	(time)
time-expression + time-expression	(time)
time-expression - time-expression	(integer)
time-expression + timestamp-expression	(timestamp)

The following example, in which 10 minutes is added to a **Time** primitive type variable, uses a millisecond integer value.

```
time := time + 600000            // 10 * 60 * 1000
```

If you add **60000** milliseconds (one minute) to a time variable and the assigned time is later than 23:59:59:999, the resulting value is 00:00:59:999 or later.

For details about the methods defined in the **Time** primitive type, see "Time Methods", in the following subsection.

For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

Time Methods

The methods defined in the **Time** primitive type are summarized in the following table.

Method	Description
currentLocaleFormat	Returns a string containing the time in the format of the current locale
display	Returns a string representing the value of the receiver
format	Returns a string containing the time in the specified format
hour	Returns the hour part of the receiver in 24-hour clock form
isValid	Returns **true** if the receiver contains a valid time
milliSecond	Returns the millisecond part of the receiver
minute	Returns the minute part of the receiver
parseWithCurrentLocale	Sets the receiver to the result of parsing a string representing a time for the current locale
parseWithFmtAndLcid	Sets the receiver to the result of parsing a string representing a time for the specified format and the specified locale
parseWithPicAndLcid	Sets the receiver to the result of parsing a string representing a time for the specified time picture and the specified locale
second	Returns the second part of the receiver
setByteOrderLocal	Returns a time that has the bytes ordered as required by the local node
setByteOrderRemote	Returns a time that has the bytes ordered as required by the specified remote node
setTime	Sets the value of the receiver to a specified time in 24-hour clock form
setTimeStrict	Sets the value of the receiver to a specified time and checks that the individual time values are within range
subtract	Returns the interval between the receiver and the specified time
userFormat	Returns a string containing the receiver in the supplied time format
userFormatAndLcid	Returns a string containing the receiver in the specified time format for the specified locale
userFormatPicAndLcid	Returns a string containing the receiver in the specified time picture for the specified locale

currentLocaleFormat

Signature `currentLocaleFormat(): String;`

The **currentLocaleFormat** method of the **Time** primitive type returns a string containing the time in the format of the current locale.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

display

Signature `display(): String;`

The **display** method of the **Time** primitive type returns a string representing the value of the receiver.

format

Signature `format(picture: String): String;`

The **format** method of the **Time** primitive type returns a string containing the time in the format specified in the **picture** parameter and current locale settings for time markers (AM/PM). For example:

```
testTimeFormat();
vars
    time : Time;
begin
    write "The time now is " & time.format("hh.m:ss tt");
end;
```

The example shown in this method writes **The time now is 08.41:08 a.m**. (if the **Time** regional setting for that user locale has the AM symbol specified as **a.m.**).

Use the string picture elements listed in the following table to construct time format picture strings. Separate each element with a space or a separator character; for example, a period (**.**) or a colon character (**:**).

Picture	Description	Output Format
h	Hours, with no leading zero (12-hour clock)	8
hh	Hours, with a leading zero (12-hour clock)	08
H	Hours, with no leading zero (24-hour clock)	13
HH	Hours, with a leading zero (24-hour clock)	08
m	Minutes, with no leading zero	6
mm	Minutes, with leading zero	06
s	Seconds, with no leading zero	47
ss	Seconds, with leading zero	07
t	One-character time marker string of the current locale	p
tt	Multiple-character time marker string of the current locale	PM

In this table, the **t** and **tt** picture elements are determined by the AM symbol or PM symbol for the current locale of the user (defined by using the **AM symbol** or **PM symbol** combo box in the **Time** sheet of the Regional Settings Properties dialog, accessed from the Regional Settings icon in the Control Panel).

Notes You can use the **defineTimeFormat** method of the **TimeFormat** class if you want to create your own transient format objects and define a time format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

hour

Signature `hour(): Integer;`

The **hour** method of the **Time** primitive type returns the hour part of the receiver in 24-hour clock form.

The following example shows the use of the **hour** method.

```
vars
    h, m, s : Integer;
begin
    // Call this method to alter the time settings
    displayedTime.setTime(displayedTime.hour + h,
                          displayedTime.minute + m,
                          displayedTime.second + s, 00);
    clockFrame.caption := displayedTime.String;
end;
```

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **Time** primitive type returns **true** if the receiver contains a valid time value.

The code fragment in the following example shows the use of the **isValid** method.

```
if not any.Time.isValid() then
    app.msgBox("New value must contain a valid Time", "No date entered",
               MsgBox_OK_Only);
    return false;
endif;
```

Use this method after a conversion instead of testing for a **null** value, as **null** indicates midnight, which is a valid time.

milliSecond

Signature `milliSecond(): Integer;`

The **milliSecond** method of the **Time** primitive type returns the millisecond part of the receiver.

The code fragment in the following example shows the use of the **milliSecond** method.

```
if eventTag = 3 then
    displayedTime.setTime(displayedTime.hour,
                          displayedTime.minute,
```

```
                                displayedTime.second,
                                displayedTime.milliSecond + 10);
        clockFrame.caption := displayedTime.minute.String[1:2].padLeadingZeros
                                (2) & ":" &
                                displayedTime.second.String[1:2].padLeadingZeros
                                (2) & ":" & displayedTime.milliSecond.String
                                [1:2].padLeadingZeros (2);
        beginTimer(10, Timer_OneShot, 4);
    elseif eventTag = 4 then
        ...
```

minute

Signature `minute(): Integer;`

The **minute** method of the **Time** primitive type returns the minute part of the receiver.

For an example of the use of the **minute** method, see the **Time** primitive type **hour** method.

parseWithCurrentLocale

Signature

```
parseWithCurrentLocale(source:    String;
                       errOffset: Integer output): Integer updating;
```

The **parseWithCurrentLocale** method of the **Time** primitive type parses the string specified in the **source** parameter to ensure that it matches the time format of the current locale in terms of element (milliseconds, seconds, minutes, hours) order and separators.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid time value (the **isValid** method of the **Time** primitive type will return **false**).

This is equivalent to calling the **parseWithFmtAndLcid** method, passing null in the **fmt** parameter and zero (**0**) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseWithFmtAndLcid

Signature

```
parseWithFmtAndLcid(source:    String;
                    fmt:       TimeFormat;
                    lcid:      Integer;
                    errOffset: Integer output): Integer updating;
```

The **parseWithFmtAndLcid** method of the **Time** primitive type parses the string specified in the **source** parameter to ensure that it matches the time format specified in the **fmt** parameter.

The time must match the appropriate values for the locale specified in the **lcid** parameter. If the source string contains a valid time, it is assigned to the receiver; otherwise the invalid time value is assigned to the receiver (the **isValid** method of the **Time** primitive type will return **false**).

If the value of the **fmt** parameter is null, the time format of the locale specified in the **lcid** parameter is used. If the value of the **lcid** parameter is zero (**0**), the time format of the current locale is used. If the value of the **fmt** parameter is *not* null, the AM/PM indicators, if specified, are used rather than the locale indicators.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid time value.

Leading zeros in the hour, minute, and second elements are optional.

This method is the same as the **parseWithPicAndLcid** method except that the picture string is taken from the **TimeFormat** class **format** property. For more details and examples of valid date matches, see the **parseWithPicAndLcid** method.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseWithPicAndLcid

Signature

```
parseWithPicAndLcid(source:    String;
                    pic:       String;
                    lcid:      Integer;
                    errOffset: Integer output): Integer updating;
```

The **parseWithPicAndLcid** method of the **Time** primitive type parses the string specified in the **source** parameter using the specified time picture and locale, validating that the source matches the time format picture to ensure that it matches the time picture string specified in the **pic** parameter.

If the source string contains a valid time, it is assigned to the receiver; otherwise the invalid time value is assigned to the receiver (the **isValid** method of the **Time** primitive type will return **false**).

If the value of the **pic** parameter is null, the time format picture of the locale specified in the **lcid** parameter is used. If the value of the **lcid** parameter is zero (**0**), the time format picture of the current locale is used.

If the value of the **source** parameter matches the format rules, the method returns zero (**0**) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid time value.

Leading zeros in the hour, minute, and second elements are optional when separators are specified.

If the marker picture is **"t"**, the source marker text must be a single character matching the first character of one of the AM/PM indicators for the locale. If the marker picture is **"tt"** (or longer), the source marker text must match exactly one of the AM/PM indicators for the locale. A locale-based case-insensitive comparison is used. If the AM indicator for the locale is **"a.m."** (for example, New Zealand), the indicator **"AM"** (for example, United States) is also accepted.

If the hour picture is **"h"** or **"hh"**, the hour value must be in the range 0 through 12. If the hour picture is **"H"** or **"HH"**, the hour value must be in the range 0 through 23. The minute and second values must be in the range 0 through 59.

Source text **"12:00a.m."** and **"0:00a.m."** with picture **"h:mmtt"** or **"hh:mmtt"** converts to time **00:00** (midnight). Source text **"12:00p.m."** with picture **"h:mmtt"** or **"hh:mmtt"** converts to time **12:00** (midday).

The format can include **".fff"** after **"s"**, to recognize a millisecond value. The decimal separator for the locale is expected between the second and millisecond values. The text can include zero (**0**) through three digits in the millisecond value.

The **"H:mm:ss.fff"** picture allows the following.

- "1:23:45.678" 1:23:45.678
- "1:23:45" 1:23:45.000
- "1:23:45.6" 1:23:45.600
- "1:23:45.600" 1:23:45.600
- "1:23:45.006" 1:23:45.006
- "1:23:45.07" 1:23:45.070

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

second

Signature `second(): Integer;`

The **second** method of the **Time** primitive type returns the second part of the receiver.

For an example of the use of the **second** method, see the **Time** primitive type **hour** method.

setByteOrderLocal

Signature `setByteOrderLocal(architecture: Integer): Time;`

The **setByteOrderLocal** method of the **Time** primitive type returns a time that has the bytes ordered as required by the local node.

The bytes of the receiver are assumed to be ordered as indicated by the **architecture** parameter.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature `setByteOrderRemote(architecture: Integer): Time;`

The **setByteOrderRemote** method of the **Time** primitive type returns a time that has the bytes ordered as required by the remote node indicated by the **architecture** parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setTime

Signature

```
setTime(hours:        Integer;
        minutes:      Integer;
        seconds:      Integer;
        milliseconds: Integer): Boolean updating;
```

The **setTime** method of the **Time** primitive type sets the value of the receiver to a specified time in 24-hour clock form using any valid combination of parameters. This method returns **true** if the specified time is valid or it returns **false** if it is invalid (for example, **24:00**).

The parameters are integer values for hours, minutes, seconds, and milliseconds.

The code fragments in the following examples show the use of the **setTime** method.

```
startTime.setTime(startH.Integer, startM.Integer, 0, 0);
endTime.setTime(endH.Integer, endM.Integer, 0, 0);

if stopWatchButton.caption = "Stop Watch" then
    endTimer(1);
    stopWatchButton.caption := "Start";
    app.doWindowEvents (1);
    clockFrame.caption := "00:00:00";
```

```
        displayedTime.setTime(0, 0, 0, 0);
    elseif ...  then
        ...
    endif;
```

You can use this method to set a valid time less than 24 hours in milliseconds. For example, the following code fragment sets the time to twelve hours in milliseconds.

```
startTime.setTime(0, 720, 0, 0);
```

setTimeStrict

Signature

```
setTimeStrict(hours:          Integer;
              minutes:        Integer;
              seconds:        Integer;
              milliseconds:   Integer): Boolean updating;
```

The **setTimeStrict** method of the **Time** primitive type checks that the hours, minutes, seconds, and milliseconds specified in the method parameters are within the range of time in the **HH:MM:SS:sss** (24-hour clock form) format.

If any of the specified parameters is outside the range of the **HH:MM:SS:sss** time format (that is, a value that is greater than 23 hours, 59 minutes, 59 seconds, or 999 milliseconds), this method returns **false** and sets the receiver to the specified "invalid" time.

Tip Use the **Time** primitive type **setTime** method to set the value of the receiver in 24-hour clock form to a specified time using any *valid* combination of parameters.

subtract

Signature

```
subtract(time: Time): TimeStampInterval;
```

The **subtract** method of the **Time** primitive type returns the interval between the receiver and the value of the **time** parameter as a **TimeStampInterval** value.

The following example shows the use of the **subtract** method.

```
vars
    now, hourFromNow : Time;
begin
    hourFromNow:= now + 60 * 60 * 1000;
    write hourFromNow.subtract(now);  // Outputs "0:01:00:00.000"
end;
```

userFormat

Signature

```
userFormat(fmt: TimeFormat): String;
```

The **userFormat** method of the **Time** primitive type returns a string containing the receiver in the specified time format.

To define your time formats, use the Schema menu **Format** command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user time format name with a dollar sign (**$**); for example, **userFormat($MyTime)**.

You can use the **defineTimeFormat** method of the **TimeFormat** class if you want to create your own transient format objects and define a time format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The code fragments in the following examples show the use of the **userFormat** method.

```
tblTime.text := p.name & " (" &
                p.startTime.userFormat($HourMin) & "-" &
                p.endTime.userFormat($HourMin) & " )";
if counter > 0 then
    igfFrame.myOutline.IGOutline.addXLabel(time.userFormat
             ($PlainTime));
endif;
```

userFormatAndLcid

Signature

```
userFormatAndLcid(fmt:  TimeFormat;
                  lcid: Integer): String;
```

The **userFormatAndLcid** method of the **Time** primitive type returns a string containing the receiver in the time format specified in the **fmt** parameter of the locale specified in the **lcid** parameter.

If the value of the **fmt** parameter is null, the time format of the locale specified in the **lcid** parameter is returned. If the value of the **lcid** parameter is zero (**0**), the time format of the current locale is returned. same as the **userFormatPicAndLcid** method except that the picture string is taken from the **TimeFormat** class **format** property. For more details and examples of valid date matches, see the **userFormatPicAndLcid** method.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userFormatPicAndLcid

Signature

```
userFormatPicAndLcid(pic:  String;
                     lcid: Integer): String;
```

The **userFormatPicAndLcid** method of the **Time** primitive type returns a string containing the receiver in the time format picture specified in the **pic** parameter of the locale specified in the **lcid** parameter.

If the value of the **pic** parameter is null, the time format picture of the locale specified in the **lcid** parameter is returned. If the value of the **lcid** parameter is zero (**0**), the time format picture of the current locale is returned.

The picture string can include **".fff"** following **"s"**, to output the millisecond part of the time; for example, **"H:mm:ss.fff"** can generate the strings **"13:07:23.543"** and **"9:53:11.000"**.

Note If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled.

The following example of the **userFormatPicAndLcid** method outputs **15:07:23.123**.

```
vars
    t : Time;
    s : String;
begin
    t.setTime(15,7,23,123);
    s := t.userFormatPicAndLcid("HH:mm:ss.fff", 0);
    write s;
end;
```

TimeStamp Type

A **TimeStamp** primitive type is used to store the variable as type timestamp; that is, the date and time. If you declare a variable of type **TimeStamp** in your method, it is initialized with the current date and time each time the method is invoked.

In JADE thin client mode, local variables of type **TimeStamp** are always initialized to the date and time relative to the presentation client.

The following example shows the use of the **TimeStamp** primitive type.

```
vars
    timeStamp : TimeStamp;
    time      : Time;
    h,m,s,ms  : Integer;
begin
    h := 15;
    m := 39;
    s := 06;
    time.setTime(h, m, s, ms);
    timeStamp.setTime(time);      // Assigns format to string
    write timeStamp;              // Outputs 11 August 2000 3:39pm
end;
```

The following table lists valid operations for the **TimeStamp** primitive type.

Expression	Expression Type
timestamp-expression + time-expression	(timestamp)
timestamp-expression - time-expression	(timestamp)
timestamp-expression + timestampinterval-expression	(timestamp)
timestamp-expression - timestampinterval-expression	(timestamp)

Caution The **TimeStamp** value that results from adding to a timestamp value or subtracting from a timestamp value does not take daylight saving into account.

For details about the methods defined in the **TimeStamp** primitive type, see "TimeStamp Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

TimeStamp Methods

The methods defined in the **TimeStamp** primitive type are summarized in the following table.

Method	Description
date	Returns the date part of the receiver
display	Returns a string representing the value of the receiver
isValid	Returns **true** if the receiver contains a valid timestamp value
literalFormat	Returns a string representing the receiver in literal format

Method	Description
localToUTCTime	Converts a timestamp in local time to UTC using the time zone in which the method executes
localToUTCTimeUsingBias	Converts a timestamp in local time to UTC using the specified bias
setByteOrderLocal	Returns a timestamp that has the bytes ordered as required by the local node
setByteOrderRemote	Returns a timestamp that has the bytes ordered as required by the specified remote node
setDate	Sets the date part of the receiver to a specified date
setTime	Sets the time part of the receiver to a specified time
time	Returns the time part of the receiver
utcToLocalTime	Converts a timestamp in UTC time to local time for the time zone in which the method executes
utcToLocalTimeUsingBias	Converts a timestamp in UTC time to local time using the specified bias

date

Signature `date(): Date;`

The **date** method of the **TimeStamp** primitive type returns a date that is the same as the date part of the receiver.

Note If you change the date returned by the **date** method using the **setDate** method of the **Date** primitive type, you are not actually changing the date part of the timestamp.

```
vars
    ts : TimeStamp;
begin
    ts.date.setDate(1,2,2007);  // does not change ts
```

To change the date part of a timestamp, use the **setDate** method of the **TimeStamp** primitive type:

```
vars
    ts : TimeStamp;
    d  : Date;
begin
    d.setDate(1,2,2007);
    ts.setDate(d);              // does change ts
```

display

Signature `display(): String;`

The **display** method of the **TimeStamp** primitive type returns a string representing the value of the receiver.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **TimeStamp** primitive type returns **true** if the receiver contains a valid timestamp value.

This method returns **false** if the date or time is not valid.

literalFormat

Signature `literalFormat(): String;`

The **literalFormat** method of the **TimeStamp** primitive type returns a string representing the receiver in literal format.

The following example shows the use of the **literalFormat** method.

```
vars
    timeStamp : TimeStamp;
    date      : Date;
    time      : Time;
begin
    date.setDate(11, 5, 2013);
    time.setTime(16, 12, 23, 0);
    timeStamp.setDate(date);
    timeStamp.setTime(time);
    write timeStamp.literalFormat;   // Outputs 2013:05:11:16:12:23
    date.setDate(11, 5, 2013);
    time.setTime(16, 12, 23, 5);
    timeStamp.setDate(date);
    timeStamp.setTime(time);
    write timeStamp.literalFormat;   // Outputs 2013:05:11:16:12:23.005
end;
```

localToUTCTime

Signature `localToUTCTime(): TimeStamp;`

The **localToUTCTime** method of the **TimeStamp** primitive type converts a timestamp in local time to Coordinated Universal Time (UTC) using the time zone of the machine in which the method executes; for example, if the method is executing in an application server for a presentation client running on another machine, the bias is taken from the time zone of the machine running the application server.

To convert between local and UTC time in a thin client application in which you want to be sensitive to the bias of the presentation client machine, use the **localToUTCTimeUsingBias** method, as shown in the following code fragment.

```
self.localToUTCTimeUsingBias(app.currentUTCBias(PresentationClient));
```

Notes Translations between UTC and local time are based on the formula **UTC = local time + bias**.

Greenwich Mean Time (GMT) has been replaced as the world standard time by Coordinated Universal Time (UTC), which is based on atomic measurements rather than the rotation of the earth. (GMT remains the standard time zone for the Prime Meridian, or zero longitude.)

See also the **TimeStamp** primitive type **localToUTCTimeUsingBias**, **utcToLocalTime**, and **utcToLocalTimeUsingBias** methods and the **Application** class **currentUTCBias** and **getUTCTime** methods.

localToUTCTimeUsingBias

Signature `localToUTCTimeUsingBias(bias: Integer): TimeStamp;`

The **localToUTCTimeUsingBias** method of the **TimeStamp** primitive type converts a timestamp in local time to UTC time using the number of minutes specified in the **bias** parameter.

Note Translations between UTC and local time are based on the formula **UTC = local time + bias**.

See also the **TimeStamp** primitive type **localToUTCTime**, **utcToLocalTimeUsingBias**, and **utcToLocalTime** methods and the **Application** class **currentUTCBias** and **getUTCTime** methods.

setByteOrderLocal

Signature `setByteOrderLocal(architecture: Integer): TimeStamp;`

The **setByteOrderLocal** method of the **TimeStamp** primitive type returns a timestamp that has the bytes ordered as required by the local node.

The bytes of the receiver are assumed to be ordered as indicated by the **architecture** parameter.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature `setByteOrderRemote(architecture: Integer): TimeStamp;`

The **setByteOrderRemote** method of the **TimeStamp** primitive type returns a timestamp that has the bytes ordered as required by the remote node indicated by the **architecture** parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the **getOSPlatform** method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment

Node Class Constant	Description
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setDate

Signature `setDate(date: Date) updating;`

The **setDate** method of the **TimeStamp** primitive type sets the date part of the receiver to a specified date.

setTime

Signature `setTime(time: Time) updating;`

The **setTime** method of the **TimeStamp** primitive type sets the time part of the receiver to a specified time.

time

Signature `time(): Time;`

The **time** method of the **TimeStamp** primitive type returns a time that is the same as the time part of the receiver.

Note If you change the time returned by the **time** method using the **setTime** method of the **Time** primitive type, you are not actually changing the time part of the timestamp.

```
vars
    ts : TimeStamp;
begin
    ts.time.setTime(09,10,11,999);  // does not change ts
```

To change the time part of a timestamp, use the **setTime** method of the **TimeStamp** primitive type:

```
vars
    ts : TimeStamp;
    t  : Time;
begin
    t.setTime(09,10,11,999);
    ts.setTime(t);                  // does change ts
```

utcToLocalTime

Signature `utcToLocalTime(): TimeStamp;`

The **utcToLocalTime** method of the **TimeStamp** primitive type converts a timestamp in UTC time to local time using the time zone of the machine in which the method executes; for example, if the method is executing in an application server for a presentation client running on another machine, the bias is taken from the time zone of the machine running the application server.

To convert between local and UTC time in a thin client application in which you want to be sensitive to the bias of the presentation client machine, use the **utcToLocalTimeUsingBias** method, as shown in the following code fragment.

```
self.utcToLocalTimeUsingBias(app.currentUTCBias(PresentationClient));
```

Note Translations between UTC and local time are based on the formula **UTC = local time + bias**.

See also the **TimeStamp** primitive type **utcToLocalTimeUsingBias**, **localToUTCTimeUsingBias**, and **localToUTCTime** methods and the **Application** class **currentUTCBias** and **getUTCTime** methods.

utcToLocalTimeUsingBias

Signature `utcToLocalTimeUsingBias(bias: Integer): TimeStamp;`

The **utcToLocalTimeUsingBias** method of the **TimeStamp** primitive type converts a timestamp in UTC time to local time using the number of minutes specified in the **bias** parameter.

Note Translations between UTC and local time are based on the formula **UTC = local time + bias**.

See also the **TimeStamp** primitive type **utcToLocalTime**, **localToUTCTimeUsingBias**, and **localToUTCTime** methods and the **Application** class **currentUTCBias** and **getUTCTime** methods.

TimeStampInterval Type

The **TimeStampInterval** primitive type is used to represent the difference between two **TimeStamp** values. The **null** value for timestamp interval is equivalent to a timestamp interval of zero duration.

The following example shows a timestamp interval established by subtracting two timestamps. This example also shows the use of methods to display the number of whole days in an interval and the remaining time in milliseconds.

```
vars
    time     : Time;
    date     : Date;
    ts1, ts2 : TimeStamp;
    interval : TimeStampInterval;
begin
    date.setDate(31,12,2007);       // New Year's Eve
    time.setTime(23,59,0,0);        // Minute before midnight
    ts1.setDate(date);
    ts1.setTime(time);
    date.setDate(1,1,2008);         // New Year's Day
    time.setTime(12,0,0,0);         // Noon
    ts2.setDate(date);
    ts2.setTime(time);
    interval := ts2 - ts1;
    write interval;                 // 0:12:01:00.000 (days:hours:mins:secs)
    write interval.getMilliseconds; // 43260000 milliseconds
end;
```

The following table lists valid operations for the **TimeStampInterval** primitive type.

Expression	Expression Type
timestamp-expression - timestamp-expression	(timestampinterval)
timestamp-expression + or - timestampinterval-expression	(timestamp)
timestampinterval-expression * or / integer-expression	(timestampinterval)
timestampinterval-expression + or - timestampinterval-expression	(timestampinterval)
timestampinterval-expression < or <= or = or >= or > or <> timestampinterval-expression	(boolean)

Caution The **TimeStampInterval** value that results from subtracting two timestamp values does not take daylight saving into account.

For details about the methods defined in the **TimeStampInterval** primitive type, see "TimeStampInterval Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

TimeStampInterval Methods

The methods defined in the **TimeStampInterval** primitive type are summarized in the following table.

Method	Description
display	Returns a string representing the value of the receiver
getMilliseconds	Returns the duration of the timestamp interval in milliseconds
isValid	Returns **true** if the receiver represents a valid timestamp interval
set	Sets the value of the receiver from a number of whole days and a number of milliseconds

display

Signature `display(): String;`

The **display** method of the **TimeStampInterval** primitive type returns a string representing the value of the receiver, in the format *days:hours:minutes:seconds.milliseconds*.

getMilliseconds

Signature `getMilliseconds(): Integer64;`

The **getMilliseconds** method of the **TimeStampInterval** primitive type returns the duration of the timestamp interval in milliseconds.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **TimeStampInterval** primitive type returns **true** if the receiver contains a valid timestamp interval value.

set

Signature
```
set(days:         Integer;
    milliseconds: Integer) updating;
```

The **set** method of the **TimeStampInterval** primitive type sets the value of a timestamp interval from the **days** and **milliseconds** parameters.

The value of the days and milliseconds parameters should both have the same sign. If one is positive and one is negative, an exception is raised.

If the value of the **milliseconds** parameter is greater than one day in milliseconds (that is, **86400000**), the **TimeStampInterval** value that is set is incremented by the number of whole days that the **milliseconds** parameter value represents.

TimeStampOffset Type

The **TimeStampOffset** primitive type is used to represent a Coordinated Universal Time (UTC) date and time value, together with an offset that indicates how much that value differs from the local time when the value was set. The value of the offset component is precise to the minute.

A **TimeStampOffset** value unambiguously identifies a single point in time.

If you declare a **TimeStampOffset** primitive type local variable in your method, it is initialized with the current date, time, and offset from UTC each time the method that declares the variable is invoked. The offset is that of the presentation client if running in thin client mode; otherwise, it is the offset of the node where the code is executing.

The following example shows the initialization of a local **TimeStampOffset** variable.

```
vars
    tso : TimeStampOffset;
    ts : TimeStamp;
begin
//  Executed in New Zealand, which is 13 hours 'ahead of Greenwich' in summer
    write ts;     // outputs 20 January 2009, 09:15:20
                  // (current date and time in New Zealand)
    write tso;    // outputs 19 January 2009, 20:15:20 +1300
                  // (current date and time in Greenwich)
end;
```

Object attributes of type **TimeStampOffset** are initialized with a **null** date, **null** time, and **null** offset; that is, **00:00:00 +0000**.

The following table lists valid operations for the **TimeStampOffset** primitive type.

Expression	Expression Type
timestampoffset-expression + time-expression	(timestampoffset)
timestampoffset-expression - time-expression	(timestampoffset)
timestampoffset-expression + timestampinterval-expression	(timestampoffset)
timestampoffset-expression - timestampinterval-expression	(timestampoffset)
timestampoffset-expression - timestampoffset-expression	(timestampinterval)
timestampoffset -expression < or <= or = or >= or > or <> timestampoffset -expression	(boolean)

For details about the methods defined in the **TimeStampInterval** primitive type, see "TimeStampOffset Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the *JADE Developer's Reference*.

TimeStampOffset Methods

The methods defined in the **TimeStampOffset** primitive type are summarized in the following table.

Method	Description
asLocalTimeStamp	Returns a timestamp representing the local time of the receiver
asUTCTimeStamp	Returns a timestamp representing the UTC time of the receiver

Method	Description
display	Returns a string representing the value of the receiver
getUTCBias	Returns the current bias in minutes
isValid	Returns **true** if the receiver represents a valid timestamp offset
setFromLocalTimeStamp	Sets the value of the receiver from the specified timestamp

asLocalTimeStamp

Signature `asLocalTimeStamp(): TimeStamp;`

The **asLocalTimeStamp** method of the **TimeStampOffset** primitive type returns a timestamp representing the local time of the receiver.

asUTCTimeStamp

Signature `asUTCTimeStamp(): TimeStamp;`

The **asUTCTimeStamp** method of the **TimeStampOffset** primitive type returns a timestamp representing the UTC time of the receiver.

display

Signature `display(): String;`

The **display** method of the **TimeStampOffset** primitive type returns a string representing the value of the receiver, in the format *date, time + offset*, where the offset is in the twenty-four hour clock format; for example, **19 January 2009, 02:56:10 +1300**.

getUTCBias

Signature `getUTCBias(): Integer;`

The **getUTCBias** method of the **TimeStampOffset** primitive type returns the current bias, which is the difference in minutes between Coordinated Universal Time (UTC) and local time (that is, **bias = UTC - 'local time'**).

As the bias is current, it includes any daylight saving adjustment in effect at the time the value is obtained.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **TimeStampOffset** primitive type returns **true** if the date, time, and offset components of the receiver contain valid values.

setFromLocalTimeStamp

Signature
```
setFromLocalTimeStamp(timeStamp: TimeStamp;
                      utcBias:   Integer) updating;
```

The **setFromLocalTimeStamp** method of the **TimeStampOffset** primitive type sets the value of the receiver from the **timeStamp** parameter and the offset specified in the **utcBias** parameter.

In the following example, a **TimeStampOffset** value is constructed from a local date and time together with an offset value obtained from the **getUTCBias** method.

```
vars
    tso_direct : TimeStampOffset;
    bias : Integer;
    tso_calculated : TimeStampOffset;
    ts : TimeStamp;
begin
    write tso_direct;
    bias := tso_direct.getUTCBias;
    tso_calculated.setFromLocalTimeStamp(ts, bias);
    write tso_calculated;
end;
```

Appendix A Global Constants Reference

Global constants provide a more-meaningful representation than simply using literal values.

JADE provides system global constants at the **Object** class level, which are grouped by the following categories.

- ApplicationStatus
- CharacterConstants
- ColorConstants
- Environment
- Exceptions
- ExecutionLocation
- JadeDbFileVolatility
- JadeDynamicObjectNames
- JadeDynamicObjectTypes
- JadeErrorCodesDatabase
- JadeErrorCodesIDE
- JadeErrorCodesRPS
- JadeErrorCodesSDS
- JadeErrorCodesWebService
- JadeLocaleIdNumbers
- JadeOdbc
- JadeProcessEvents
- JadeProfileString
- KeyCharacterCodes
- LockDurations
- LockTimeouts
- Locks
- MessageBox
- NotificationResponses
- ObjectVolatility
- PossibleTransientLeaks
- Printer
- RPSTransitionHaltCode

- SDSConnectionState
- SDSDatabaseRoles
- SDSEventTypes
- SDSReorgState
- SDSSecondaryState
- SDSStopTrackingCodes
- SDSTakeoverState
- SDSTransactionStates
- SQL
- Sounds
- SystemEvents
- SystemLimits
- TimerDurations
- UserEvents

For details about promoting class constants to global constants, see "Promoting Class Constants to Global Constants", in Chapter 4 of the *JADE Development Environment User's Guide*.

ApplicationStatus Category

The global constants for the application mouse pointer (**app.mousePointer**) status are listed in the following table.

Global Constant	Integer Value	Description
Busy	11	Hourglass mouse pointer indicates that the application is currently busy
Idle	0	Mouse pointer is in the idle state

For details about the mouse pointer (which controls the shape of the mouse pointer for all windows of the application), see the **Application** class **mousePointer** property in Chapter 1 of the *JADE Encyclopaedia of Classes*.

CharacterConstants Category

The global constants for the carriage return and tab characters are listed in the following table.

Global Constant	String or Character Value	Description
Cr	#"0D" character	Carriage return character
CrLf	#"0D 0A" string	Carriage return / line feed characters
Lf	#"0A" character	Line feed character
Tab	#"09" character	Tab character

ColorConstants Category

The color global constants are listed in the following table.

Global Constant	Integer Value
Black	0
Blue	16711680
DarkBlue	12582912
DarkGray	4210752
Gray	12632256
Green	32768
LightGreen	65280
LightYellow	8454143
Mauve	16711935
Purple	12583104
Red	255
White	16777215
Yellow	65535

Environment Category

The global constant for JADE environments that you can use in your applications, if required, is listed in the following table.

Global Constant	Primitive Type	Value
IsUnicodeSystem	Boolean	(-1).Character.Integer <> 255

Exceptions Category

The global constants for exceptions are listed in the following table.

Global Constant	Integer Value	Description
Ex_Abort_Action	1	Causes the currently executing methods to be aborted
Ex_Continue	0	Resumes execution from the next expression after the expression that caused the exception
Ex_Pass_Back	-1	Passes control back to the prior local exception handler for this type of exception, or if a local handler is not found, a global exception handler for this type of exception
Ex_Resume_Next	2	Passes control back to the method that armed the exception handler

For more details, see "Exception Class Return Values", in Chapter 1 of the *JADE Encyclopaedia of Classes*.

ExecutionLocation Category

The global constants for the location of executed methods are listed in the following table.

Global Constant	Integer Value	Method is executed…
CurrentLocation	0	In the current location
DatabaseServer	1	On the database server node
PresentationClient	2	On the presentation client (applicable to applications running in thin client mode)

JadeDbFileVolatility Category

The global constants for the volatility of database files and partitions are listed in the following table.

Global Constant	Integer Value
FileVolatility_Frozen	Volatility_Frozen + 1
FileVolatility_Stable	Volatility_Stable + 1
FileVolatility_Transparent	0
FileVolatility_Volatile	Volatility_Volatile + 1

JadeDynamicObjectNames Category

The global constants for the name of dynamic objects used by JADE **RootSchema** classes are listed in the following table.

Global Constant	String Value
JStats_ArrayName	"JStatsArray"
JStats_DictionaryName	"JStatsDictionary"
JStats_JadeBytesName	"JStatsJadeBytes"
JStats_SetName	"JStatsSet"
SDS_PrimaryName	"SDSPrimary"
SDS_SecondaryName	"SDSSecondary"
SDS_SecondaryProxyName	"SDSSecondaryProxy"
SDS_TransactionName	"SDSTransaction"

For more detail, see the **JadeDynamicObject** class **name** property in Chapter 1 of the *JADE Encyclopaedia of Classes*.

JadeDynamicObjectTypes Category

The global constants for the type of dynamic objects used by JADE **RootSchema** classes are listed in the following table.

Global Constant	Integer Value
JStats_ArrayType	101
JStats_DictionaryType	102
JStats_JadeBytesType	104
JStats_SetType	103
SDS_PrimaryType	1
SDS_SecondaryProxyType	2
SDS_SecondaryType	3
SDS_TransactionType	4

For more detail, see the **JadeDynamicObject** class **type** property in Chapter 1 of the *JADE Encyclopaedia of Classes*.

JadeErrorCodesDatabase Category

The global constants for JADE database exception error codes that you can use in your own exception handlers, if required, are listed in the following table.

Global Constant	Integer Value
JErr_DbDiskFull	3033
JErr_DbEditionOutOfDate	3049
JErr_DbFileExists	3071
JErr_DbFileNotCreated	3121
JErr_DbFileNotDefined	3120
JErr_DbFileNotFound	3036
JErr_DbFileOffline	3162
JErr_DbLockedForArchive	3079
JErr_DbLockedForReorg	3059
JErr_DbUserAbort	3051
JErr_FileInstantiated	3142
JErr_PartitionModulusRangeErr	3161

For details about exception handling, see Chapter 3 of the *JADE Developer's Reference*. See also "Exception Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*.

JadeErrorCodesIDE Category

The global constants for the JADE development environment error codes are listed in the following table.

Global Constant	Integer Value
Patch_History_Load_Dup_Patch	16007
Patch_History_Load_No_Schema	16006

JadeErrorCodesRPS Category

The global constants for the Relational Population Service (RPS) error codes that you can use in your own exception handlers, if required, are listed in the following table.

Global Constant	Integer Value
JErr_DataPumpAlreadyRunning	3265
JErr_LegalOnRpsOnly	3264
JErr_NotDataPumpApp	3266
JErr_RpsAdminHalt	3262
JErr_RpsConnectionError	3274
JErr_RpsDuplicatedKey	3258
JErr_RpsExtractRequestError	3269
JErr_RpsMultiRowAffected	3260
JErr_RpsTableNameNotFound	3273
JErr_RpsZeroRowsAffected	3259
JErr_ValidOnRpsMappingOnly	3272

JadeErrorCodesSDS Category

The global constants for the Synchronized Database Service (SDS) error codes that you can use in your own exception handlers, if required, are listed in the following table.

Global Constant	Integer Value
JErr_SdsIllegalOnPrimary	3207
JErr_SdsIllegalOnSecondary	3206
JErr_SdsIncompleteJournal	3200
JErr_SdsInvalidCommand	3205
JErr_SdsMaxSecondariesExceeded	3210
JErr_SdsNotInitialized	3201
JErr_SdsResponseTimeout	3212

Global Constant	Integer Value
JErr_SdsSecondaryNotAttached	3204
JErr_SdsSecondaryNotFound	3208
JErr_SdsTrackerBusy	3211

JadeErrorCodesWebService Category

The global constants for Web service error codes that you can use in your own exception handlers, if required, are listed in the following table.

Global Constant	Integer Value
JADEWS_CREATE_WS_APP_FAILED	11082
JADEWS_DECIMAL_OVERFLOW	11055
JADEWS_ENUM_FAULT	11053
JADEWS_INVALID_REQUEST	11056
JADEWS_INVALID_RESPONSE	11051
JADEWS_LICENCES_EXCEEDED	11004
JADEWS_NO_WEBSERVICE_CLASS	11001
JADEWS_NO_WEBSERVICE_METHOD	11002
JADEWS_RESPONSE_TIME_EXCEEDED	11005
JADEWS_SERVICE_FAULT	11052
JADEWS_SERVICE_UNAVAILABLE	11008
JADEWS_SESSION_ENDED	11006
JADEWS_SESSION_TIMED_OUT	11007
JADEWS_STRING_TOO_LONG	11054
JADEWS_VERSION_MISMATCH	11009
JADEWS_WSDL_GENERATION_FAILED	11081

For more details, see Chapter 11, "Building Web Services Applications", of the *JADE Developer's Reference*.

JadeLocaleIdNumbers Category

The global constants for commonly used locale identifiers (LCIDs) are listed in the following table. You can use these values with the **Application** class **setJadeLocale** method, which changes the formatting information to suppress the regional overrides for all locales except for the **LCID_SessionWithOverrides** global constant.

Global Constant	Integer Value
LCID_Arabic_Bahrain	15361
LCID_Arabic_Egypt	3073

Global Constant	Integer Value
LCID_Arabic_Kuwait	13313
LCID_Arabic_SaudiArabia	1025
LCID_Arabic_UAE	14337
LCID_Assamese_India	1101
LCID_Bengali_India	1093
LCID_Chinese_SimplfdSingapore	4100
LCID_Chinese_SimplifiedPRC	2052
LCID_Chinese_TraditionalMacao	5124
LCID_Chinese_TraditionalTaiwan	1028
LCID_Chinese_TraditnalHongKong	3076
LCID_Dutch_Belgium	2067
LCID_Dutch_Netherlands	1043
LCID_English_Australia	3081
LCID_English_Canada	4105
LCID_English_India	16393
LCID_English_Ireland	6153
LCID_English_Jamaica	8201
LCID_English_Malaysia	17417
LCID_English_NewZealand	5129
LCID_English_Singapore	18441
LCID_English_SouthAfrica	7177
LCID_English_UnitedKingdom	2057
LCID_English_UnitedStates	1033
LCID_French_Belgium	2060
LCID_French_Canada	3084
LCID_French_France	1036
LCID_French_Switzerland	4108
LCID_German_Austria	3079
LCID_German_Germany	1031
LCID_German_Switzerland	2055
LCID_Greek_Greece	1032
LCID_Gujarati_India	1095
LCID_Hindi_India	1081
LCID_Indonesian_Indonesia	1057

Global Constant	Integer Value
LCID_Invariant	127
LCID_Irish_Ireland	2108
LCID_Italian_Italy	1040
LCID_Japanese_Japan	1041
LCID_Kannada_India	1099
LCID_Konkani_India	1111
LCID_Korean_Korea	1042
LCID_Malay_Malaysia	1086
LCID_Malayalam_India	1100
LCID_Maori_NewZealand	1153
LCID_Marathi_India	1102
LCID_Oriya_India	1096
LCID_Polish_Poland	1045
LCID_Portuguese_Brazil	1046
LCID_Portuguese_Portugal	2070
LCID_Punjabi_India	1094
LCID_Russian_Russia	1049
LCID_Sanskrit_India	1103
LCID_SessionWithOverrides	1024
LCID_Spanish_Argentina	11274
LCID_Spanish_Chile	13322
LCID_Spanish_Mexico	2058
LCID_Spanish_Nicaragua	19466
LCID_Spanish_PuertoRico	20490
LCID_Spanish_Spain_InternatSrt	3082
LCID_Spanish_Spain_TradSort	1034
LCID_Spanish_UnitedStates	21514
LCID_Tamil_India	1097
LCID_Telugu_India	1098
LCID_Thai_Thailand	1054
LCID_Vietnamese_Vietnam	1066
LCID_Welsh_UnitedKingdom	1106

The **LCID_Invariant** global constant is an operating system-independent locale, based on English (USA). The format of the short date is **MM/dd/yyyy** and time is **HH:mm:ss**.

The **LCID_SessionWithOverrides** global constant is the Windows session locale, including user regional overrides. It is the initial locale for the application. You can pass the enhanced locale support methods that have an **lcid** parameter zero (**0**), in which case the current locale is used.

JadeOdbc Category

The global constants for Web service error codes that you can use in your own exception handlers, if required, are listed in the following table.

Global Constant	Integer Value
JErr_Attribute_Name_Conflict	8347
JErr_ColumnName_Cannot_Change	8356
JErr_Column_Not_Found	8349
JErr_Invalid_For_RpsMapping	8353
JErr_No_Jade_Type	8352
JErr_Not_Soft_Table	8351
JErr_SQL_Type_Not_Mapped	8348
JErr_Table_Name_Conflict	8345
JErr_Table_Not_Found	8346

JadeProcessEvents Category

The global constants for JADE process events for which user notifications are sent are listed in the following table.

Global Constant	Integer Value	Description
Process_Call_Stack_Info_Event	System_Base_Event + 243	Call stack information
Process_Local_Stats_Event	System_Base_Event + 240	Local request statistics
Process_Remote_Stats_Event	System_Base_Event + 241	Remote request statistics
Process_TDB_Analysis_Event	System_Base_Event + 245	Detailed analysis of a transient database file
Process_TDB_Info_Event	System_Base_Event + 244	File name and length of a transient database file
Process_Web_Stats_Event	System_Base_Event + 242	Web statistics information

JadeProfileString Category

The global constants for JADE initialization file profiles are listed in the following table.

Global Constant	Description
ProfileAllKeys	Returns all key strings, separated by spaces, in the initialization file section
ProfileAllSections	Returns all initialization file sections, separated by spaces

Global Constant	Description
ProfileRemoveKey	Removes the key string from the initialization file section
ProfileRemoveSection	Removes an entire initialization file section

For more details, see the **Application** class **getProfileString**, **getProfileStringAppServer**, **setProfileString**, and **setProfileStringAppServer** methods, the **Process** class **getProfileString** and **setProfileString** methods, and the **Node** class **getProfileString** and **setProfileString** methods in Chapter 1 of the *JADE Encyclopaedia of Classes*.

KeyCharacterCodes Category

The global constants for printable key character codes are listed in the following table.

Global Constant	Value	Global Constant	Value
J_key_0	48	J_key_1	49
J_key_2	50	J_key_3	51
J_key_4	52	J_key_5	53
J_key_6	54	J_key_7	55
J_key_8	56	J_key_9	57
J_key_A	65	J_key_Ampersand	38
J_key_Asterisk	42	J_key_AtSign	64
J_key_B	66	J_key_Back	8
J_key_BackSlash	92	J_key_Bar	124
J_key_C	67	J_key_Carat	94
J_key_Colon	58	J_key_Comma	44
J_key_Ctrl	17	J_key_CurlyLeft	123
J_key_CurlyRight	125	J_key_D	68
J_key_Delete	46	J_key_Dollar	36
J_key_DoubleQuote	34	J_key_DownArrow	40
J_key_E	69	J_key_End	35
J_key_Enter	13	J_key_Equal	61
J_key_Escape	27	J_key_Exclamation	33
J_key_F	70	J_key_F1	112
J_key_F10	121	J_key_F11	122
J_key_F12	123	J_key_F2	113
J_key_F3	114	J_key_F4	115
J_key_F5	116	J_key_F6	117
J_key_F7	118	J_key_F8	119

Global Constant	Value	Global Constant	Value
J_key_F9	120	J_key_G	71
J_key_GreaterThan	62	J_key_H	72
J_key_Hash	35	J_key_Home	36
J_key_Hyphen	45	J_key_I	73
J_key_Insert	45	J_key_J	74
J_key_K	75	J_key_L	76
J_key_LeftArrow	37	J_key_LeftBracket	91
J_key_LeftParenthesis	40	J_key_LeftQuote	96
J_key_LessThan	60	J_key_Linefeed	10
J_key_M	77	J_key_N	78
J_key_NumPadMinus	109	J_key_NumPadMultiply	106
J_key_NumPadPlus	107	J_key_O	79
J_key_P	80	J_key_PageDown	34
J_key_PageUp	33	J_key_Percent	37
J_key_Plus	43	J_key_Q	81
J_key_Question	63	J_key_R	82
J_key_Return	13	J_key_RightArrow	39
J_key_RightBracket	93	J_key_RightParenthesis	41
J_key_S	83	J_key_SemiColon	59
J_key_Shift	16	J_key_SingleQuote	39
J_key_Slash	47	J_key_Space	32
J_key_Stop	46	J_key_T	84
J_key_Tab	9	J_key_Tilde	126
J_key_U	85	J_key_UnderScore	95
J_key_UpArrow	38	J_key_V	86
J_key_W	87	J_key_X	88
J_key_Y	89	J_key_Z	90
J_key_a	97	J_key_b	98
J_key_c	99	J_key_d	100
J_key_e	101	J_key_f	102
J_key_g	103	J_key_h	104
J_key_i	105	J_key_j	106
J_key_k	107	J_key_l	108
J_key_m	109	J_key_n	110

Global Constant	Value	Global Constant	Value
J_key_o	111	J_key_p	112
J_key_q	113	J_key_r	114
J_key_s	115	J_key_t	116
J_key_u	117	J_key_v	118
J_key_w	119	J_key_x	120
J_key_y	121	J_key_z	122
J_with_Alt	4	J_with_Ctrl	2
J_with_Shift	1		

LockDurations Category

The global constants for lock durations are listed in the following table.

Global Constant	Integer Value	Description
Persistent_Duration	2	Reserved for future use (not yet implemented).
Session_Duration	1	Automatically unlocks the object at the end of the current session (that is, the current thread or process) if no manual unlocks are issued. In persistent transaction state, all unlock requests for persistent objects are ignored. Similarly, in transient transaction state, all unlock requests for shared transient objects are ignored. A session lock is therefore not released if the unlock request is made while in transaction state. To release a session lock, the unlock request must be made while not in transaction state.
Transaction_Duration	0	Automatically unlocks the object at the end of transaction time. If a manual unlock is issued, this unlocks the object only if you are not in transaction or load state.

LockTimeouts Category

The global constants for lock timeouts are listed in the following table.

Global Constant	Integer Value	Description
LockTimeout_Immediate	-1	Lock request times out immediately
LockTimeout_Infinite	Max_Integer (#7FFFFFFF)	Lock request times out only after the number of milliseconds indicated by the **Max_Integer** value is reached
LockTimeout_Server_Defined	0	Uses the server-defined default

Locks Category

The global constants for locks are listed in the following table.

Global Constant	Integer Value	Description
Exclusive_Lock	3	No other process can lock the same object.
Get_Lock	0	Not valid for lock requests. This lock type indicates a process is waiting to acquire a lock that will cause all other lock requests for the object to be queued (for example, when upgrading a lock from update to exclusive).
Reserve_Lock	2	When you place a **reserve** lock on an object, other processes attempting to acquire an **exclusive** lock or **reserve** lock on that same object wait until the **reserve** lock is relinquished, but those attempting to acquire a shared lock succeed.
Share_Lock	1	When you place a **shared** lock on an object, other processes attempting to update the object or explicitly acquire an **exclusive** lock wait until the lock is released but can acquire a **shared** lock or a **reserve** lock.
Update_Lock	4	Placing an **update** lock allows you to update the object, while still allowing other processes to acquire **shared** locks to view the most recently committed edition.

MessageBox Category

The global constants for message boxes are listed in the following table. (For more details, see the **Application** class **msgBox** method in Chapter 1 of the JADE *Encyclopaedia of Classes*.)

Global Constant	Integer Value	Description
MsgBox_Abort_Retry_Ignore	2	Displays the **Abort**, **Retry**, and **Ignore** buttons
MsgBox_App_Modal	0	User must respond to the message box before continuing work
MsgBox_Default_First	0	First button is the default
MsgBox_Default_Second	256	Second button is the default
MsgBox_Default_Third	512	Third button is the default
MsgBox_Exclamation_Mark_Icon	48	Displays the Exclamation Mark icon
MsgBox_Information_Icon	64	Displays the Information icon
MsgBox_OK_Cancel	1	Displays the **OK** and **Cancel** buttons
MsgBox_OK_Only	0	Displays only the **OK** button
MsgBox_Question_Mark_Icon	32	Displays the Question Mark icon
MsgBox_Retry_Cancel	5	Displays the **Retry** and **Cancel** buttons
MsgBox_Return_Abort	3	Returned when the **Abort** button has been selected
MsgBox_Return_Cancel	2	Returned when the **Cancel** button or the ESC key has been selected

Global Constant	Integer Value	Description
MsgBox_Return_Ignore	5	Returned when the **Ignore** button has been selected
MsgBox_Return_No	7	Returned when the **No** button has been selected
MsgBox_Return_OK	1	Returned when the **OK** button has been selected
MsgBox_Return_Retry	4	Returned when the **Retry** button has been selected
MsgBox_Return_Yes	6	Returned when the **Yes** button has been selected
MsgBox_Stop_Icon	16	Displays the Stop icon
MsgBox_System_Modal	4096	All applications are suspended until the user responds to the message box
MsgBox_Yes_No	4	Displays **Yes** and **No** buttons
MsgBox_Yes_No_Cancel	3	Displays **Yes**, **No**, and **Cancel** buttons

NotificationResponses Category

The global constants for event notification responses are listed in the following table.

Global Constant	Integer Value	Sends a notification …
Response_Cancel	1	When the target notification object receives a matching event and then cancels the notification
Response_Continuous	0	Whenever the target notification object receives a matching event
Response_Suspend	2	When the target notification object receives a matching event and suspends notification until users refresh their local copy of the target object from the database

ObjectVolatility Category

The global constants for the volatility state of persistent objects are listed in the following table.

Global Constant	Integer Value	Object is…
Volatility_Frozen	#04	Frozen (that is, it is not updated)
Volatility_Stable	#08	Stable (that is, it is updated infrequently)
Volatility_Volatile	#00	Volatile (that is, it is updated often)

For details, see "Cache Concurrency", in Chapter 6 of the *JADE Developer's Reference*.

PossibleTransientLeaks Category

The global constant that enables you to mark lines of code for exclusion from the transient leaks analysis report is listed in the following table.

Global Constant	Description
ExcludeFromTransientLeakReport	Marks the line of code for exclusion from the transient leaks analysis report when specified in a comment on the same line

For details about possible transient leak analysis, see "Locating Possible Transient Object Leaks", in Chapter 4 of the *JADE Development Environment User's Guide*.

Printer Category

The printer global constants are listed in the following table.

Global Constant	Integer Value	Description
Print_10X11	45	10 x 11 inches
Print_10X14	16	10 x 14 inches
Print_11X17	17	11 x 17 inches
Print_15X11	46	15 x 11 inches
Print_9X11	44	9 x 11 inches
Print_A2	66	A2 420 x 594 mm
Print_A3	8	A3 297 x 420 mm
Print_A3_Extra	63	A3 Extra 322 x 445 mm
Print_A3_Extra_Transverse	68	A3 Extra Transverse
Print_A3_Transverse	67	A3 Transverse 297 x 420 mm
Print_A4	9	A4 210 x 297 mm
Print_A4Small	10	A4 Small 210 x 297 mm
Print_A4_Extra	53	A4 Extra 9.27 x 12.69 inches
Print_A4_Plus	60	A4 Plus 210 x 330 mm
Print_A4_Transverse	55	A4 Transverse 210 x 297 mm
Print_A5	11	A5 148 x 210 mm
Print_A5_Extra	64	A5 Extra 174 x 235 mm
Print_A5_Transverse	61	A5 Transverse 148 x 210 mm
Print_A_Plus	57	Super A - A4 227 x 356 mm
Print_B4	12	B4 250 x 354 mm
Print_B5	13	B5 182 x 257 mm

Global Constant	Integer Value	Description
Print_B5_Extra	65	B5 (ISO) Extra 201 x 276 mm
Print_B5_Transverse	62	B5 (JIS) Transverse 182 x 257 mm
Print_B_Plus	58	Super B – A3 305 x 487 mm
Print_CSheet	24	C size sheet
Print_Cancelled	15015	Print progress dialog **Cancel** button was clicked
Print_Collate_Ignored	15030	Printing started, so change to the **collate** property ignored
Print_Copies_Ignored	15010	Printing started, so change of copies ignored
Print_Currently_Open	15013	Printer is currently open
Print_Custom_Paper	256	Customized paper size
Print_DSheet	25	D size sheet
Print_DocumentType_Invalid	15032	You changed **printer**.**documentType** to **Print_Custom_Paper** instead of calling the **printer**.**setCustomPaperSize** method
Print_Duplex_Ignored	15029	Printing started, so change to the **duplex** property ignored
Print_Duplex_Invalid	15028	Value of the **duplex** property is invalid
Print_ESheet	26	E size sheet
Print_Env_10	20	Envelope #10 4.125 x 9.5 inches
Print_Env_11	21	Envelope #11 4.5 x 10.375 inches
Print_Env_12	22	Envelope #12 4.75 x 11 inches
Print_Env_14	23	Envelope #14 5 x 11.5 inches
Print_Env_9	19	Envelope #9 3.875 x 8.875 inches
Print_Env_B4	33	Envelope B4 250 x 353 mm
Print_Env_B5	34	Envelope B5 176 x 250 mm
Print_Env_B6	35	Envelope B6 176 x 125 mm
Print_Env_C3	29	Envelope C3 324 x 458 mm
Print_Env_C4	30	Envelope C4 229 x 324 mm
Print_Env_C5	28	Envelope C5 162 x 229 mm
Print_Env_C6	31	Envelope C6 114 x 162 mm
Print_Env_C65	32	Envelope C65 114 x 229 mm
Print_Env_DL	27	Envelope DL 110 x 220 mm
Print_Env_Invite	47	Envelope Invite 220 x 220 mm
Print_Env_Italy	36	Envelope 110 x 230 mm
Print_Env_Monarch	37	Envelope Monarch 3.875 x 7.5 inches
Print_Env_Personal	38	6¾ Envelope 3.625 x 6.5 inches

Global Constant	Integer Value	Description
Print_Executive	7	Executive 7¼ x 10½ inches
Print_Failed_To_Obtain_Printer	15014	Task could not obtain printer
Print_Fanfold_Lgl_German	41	German Legal Fanfold 8½ x 13 inches
Print_Fanfold_Std_German	40	German Std Fanfold 8½ x 12 inches
Print_Fanfold_US	39	US Std Fanfold 14.875 x 11 inches
Print_Folio	14	Folio 8½ x 13 inches
Print_Frame_Too_Large	15007	Frame larger than page depth
Print_Header_Footer_Too_Large	15006	Header and footer larger than page depth
Print_In_Preview	15031	Print object in use in print preview and cannot be reused
Print_ISO_B4	42	B4 (ISO) 250 x 353 mm
Print_Invalid_Control	15001	Attempt to print control that is not a frame
Print_Invalid_Position	15024	Attempt to set a print position that is outside the valid range
Print_Japanese_PostCard	43	Japanese Postcard 100 x 148 mm
Print_Landscape	2	Horizontal page orientation
Print_Ledger	4	Ledger 17 x 11 inches
Print_Legal	5	Legal 8½ x 14 inches
Print_Legal_Extra	51	Legal Extra 9.275 x 15 inches
Print_Letter	1	Letter 8½ x 11 inches
Print_LetterSmall	2	Letter Small 8½ x 11 inches
Print_Letter_Extra	50	Letter Extra 9.275 x 12 inches
Print_Letter_Extra_Transverse	56	Letter Extra Transverse 9.275 x 12 inches
Print_Letter_Plus	59	Letter Plus 8.5 x 12.69 inches
Print_Letter_Transverse	54	Letter Transverse 8.275 x 11 inches
Print_Metafile_Playback_Error	15033	Internal error occurred when attempting to play back a print metafile
Print_NewPage_Failed	15002	New page on printer failed
Print_NoDefault_Printer	15021	Your workstation has no default printer set up
Print_Not_Available	15017	Printer does not match available printers
Print_Note	18	Note 8½ x 11 inches
Print_Orientation_Invalid	15011	Orientation must be **Portrait** (1) or **Landscape** (2)
Print_PaperSource_Invalid	15027	Value of the **paperSource** property is invalid
Print_Portrait	1	Vertical page orientation
Print_Preview_Ignored	15008	Printing started, so change of print preview ignored
Print_PrintReport_Ignored	15022	Printing started, so change of print report ignored

Global Constant	Integer Value	Description
Print_Printer_Ignored	15023	You attempted to change the printer in use after printing began or before any printing has occurred (the printer must be closed before commencing the new output on a different printer)
Print_Printer_Not_Open	15003	Printer not open
Print_Printer_Open_Failed	15005	Open of printer failed
Print_Quarto	15	Quarto 215 x 275 mm
Print_Restricted	15019	
Print_Statement	6	Statement 5½ x 8½ inches
Print_Stopped	15016	Print progress dialog **Stop** button was clicked
Print_Successful	0	The print was successful
Print_Tabloid	3	Tabloid 11 x 17 inches
Print_Tabloid_Extra	52	Tabloid Extra 11.69 x 18 inches
Print_TextOut_Error	15004	Text output to printer failed
Print_Unformatted_failed	15025	Printing of unformatted text failed; that is, the **printUnformatted** method request failed

RPSTransitionHaltCode Category

The global constants that enable you to determine whether the Relational Population Service (RPS) table alter script will be loaded automatically or whether manual action is required from the RDB administrator when a schema instantiation is replayed by an RPS node and the data pump application and database tracking are halted to achieve a schema transition.

When the event **RPS_SchemaTransition** (event type 220) is caused on the system instance, the **userInfo** parameter passed to the **userNotification** callback method contains one of the global constants listed in the following table. (The **RPS_SchemaTransition** event is represented by a global constant in the **SDSEventTypes** category.)

Global Constant	Integer Value	Description
RPS_HaltAutoScript	1	An automatic initiate alter script was generated (will be automatically loaded by the data pump application if configured to automatically restart)
RPS_HaltManualScript	2	A manual alter script was generated (requires administration user intervention to apply changes to RDB before tracking can be resumed)
RPS_HaltMappingDeleted	3	The RPS mapping was deleted on the primary database, rendering the RPS node and associated RDB defunct
RPS_HaltNoScript	0	Changes do not affect RDB, so no script was generated

SDSConnectionState Category

The global constants for the Synchronized Database Service (SDS) connection state are listed in the following table.

Global Constant	Integer Value
SDS_Connected	2
SDS_Connecting	3
SDS_ConnectionFailed	4
SDS_ConnectionStateUndefined	0
SDS_Disconnected	1

These are used in return values or dynamic object attribute values by the **JadeDatabaseAdmin** class **sdsGetMyServerInfo** method.

SDSDatabaseRoles Category

The global constants for the Synchronized Database Service (SDS) database roles are listed in the following table.

Global Constant	Integer Value
SDS_RolePrimary	1
SDS_RoleSecondary	2
SDS_RoleUndefined (returned when the method is invoked on a non-SDS-capable or non-RPS-capable system)	0
SDS_SubroleNative	1
SDS_SubroleRelational	2

These are used in return values or dynamic object attribute values by the **JadeDatabaseAdmin** class **sdsGetMyServerInfo** or **sdsGetDatabaseRole** method.

SDSEventTypes Category

The global constants for the Synchronized Database Service (SDS) event types are listed in the following table.

Global Constant	Integer Value
RPS_SchemaTransition	220
SDS_ConnectionStateChange	17386
SDS_JournalTransferStopped	17385
SDS_RoleChangeEvent	22
SDS_RoleChangeProgress	17387

Global Constant	Integer Value
SDS_TrackingDisabled	17388
SDS_TrackingEnabled	17389
SDS_TrackingHalted	17390
SDS_TrackingStarted	17389
SDS_TrackingStopped	17388

These are used in return or dynamic object attribute values by the **JadeDatabaseAdmin** class **sdsGetMyServerInfo** method and in the case of the **RPS_SchemaTransition** global constant, in the **userInfo** parameter passed to the **userNotification** callback method that contains one of the **RPSTransitionHaltCode** category global constants.

SDSReorgState Category

The global constants for the Synchronized Database Service (SDS) reorganization state are listed in the following table.

Global Constant	Integer Value
SDS_ReorgStateNotReorging	1
SDS_ReorgStateOfflinePhase	5
SDS_ReorgStateReorgingFiles	4
SDS_ReorgStateRestarting	6
SDS_ReorgStateSeekingApproval	2
SDS_ReorgStateStarting	3

These are used in return values or dynamic object attribute values by the **JadeDatabaseAdmin** class **sdsGetMyServerInfo** method.

SDSSecondaryState Category

The global constants for the Synchronized Database Service (SDS) secondary state are listed in the following table.

Global Constant	Integer Value
SDS_BlockWrite	2
SDS_JournalSwitch	1
SDS_StateCatchingUp	1
SDS_StateDisconnected	0
SDS_StateReorging	5
SDS_StateSynchronized	2
SDS_StateTrackingHalted	4
SDS_StateTransferHalted	3

These are used in return values or dynamic object attribute values by the **JadeDatabaseAdmin** class **sdsGetMyServerInfo** method.

SDSStopTrackingCodes Category

The global constants for the Synchronized Database Service (SDS) stop tracking are listed in the following table.

Global Constant	Integer Value
SDS_AuditStopTrackingAll	1
SDS_AuditStopTrackingNative	2
SDS_AuditStopTrackingRdb	3
SDS_ReasonAdminAudited	1
SDS_ReasonAdminDirect	2
SDS_ReasonAutoUpgradeMismatch	6
SDS_ReasonDeltaModeEntered	12
SDS_ReasonEnablingDbCrypt	13
SDS_ReasonErrorHalt	8
SDS_ReasonRestart	10
SDS_ReasonRpsAdminHalt	4
SDS_ReasonRpsReorgHalt	9
SDS_ReasonRpsRestart	11
SDS_ReasonRpsSnapshot	3
SDS_ReasonTakeover	7
SDS_ReasonTransition	5

These are used in return values or dynamic object attribute values by the **JadeDatabaseAdmin** class **getReasonTrackingStoppedString**, **sdsAuditStopTracking**, **sdsGetMyServerInfo**, and **sdsGetSecondaryInfo** methods.

SDSTakeoverState Category

The global constants for the Synchronized Database Service (SDS) takeover state are listed in the following table.

Global Constant	Integer Value
SDS_HostileTakeoverInitiated	4
SDS_PrimaryRoleActive	1
SDS_PrimaryRoleRelinquished	11
SDS_RelinquishPrimaryRole	5
SDS_RelinquishSecondaryRole	6

Global Constant	Integer Value
SDS_SecondaryRoleActive	2
SDS_SecondaryRoleRelinquished	12
SDS_TakeoverAbandoned	8
SDS_TakeoverConditional	1
SDS_TakeoverFailure	7
SDS_TakeoverForced	2
SDS_TakeoverInitiated	3
SDS_WaitForQuietPoint	9
SDS_WaitForTakeoverDisposition	10

These values are used as the values contained in the **userInfo** parameter for a role change progress event notification. For details, see "Detecting SDS Role Changes", in Chapter 10 of the *JADE Developer's Reference*. See also the **sdsInitiateTakeover** method of the **JadeDatabaseAdmin** class.

SDSTransactionStates Category

The global constants for the Synchronized Database Service (SDS) transaction states are listed in the following table.

Global Constant	Integer Value
SDS_TranDeferred	3
SDS_TranInDoubt	8
SDS_TranInterrupted	2
SDS_TranNormal	1
SDS_TranPrepareToCommit	6
SDS_TranReadyToAbort	7
SDS_TranReadyToCommit	5
SDS_TranWaitingAuditCommit	4

These are used in return values or dynamic object attribute values by the **JadeDatabaseAdmin** class **sdsGetMyServerInfo**, **sdsGetTransactions**, or **sdsGetTransactionsAt** method.

SQL Category

The global constants for the Structured Query Language (SQL) are listed in the following table. The SQL global constants are for internal use only.

Global Constant	Integer Value
SQL_COLLECTION	3

Global Constant	Integer Value
SQL_COLLECTION_METHOD	7
SQL_EXPLICIT_INVERSE	4
SQL_IMPLICIT_INVERSE	5
SQL_INVALID	0
SQL_METHOD	2
SQL_PROPERTY	1
SQL_TYPE_BIGINT	-5
SQL_TYPE_BINARY	-2
SQL_TYPE_BIT	-7
SQL_TYPE_CHAR	1
SQL_TYPE_DATE	9
SQL_TYPE_DATE_VERSION3	91
SQL_TYPE_DECIMAL	3
SQL_TYPE_DOUBLE	8
SQL_TYPE_FLOAT	6
SQL_TYPE_INTEGER	4
SQL_TYPE_INTERVAL_DAY_TO_SEC	110
SQL_TYPE_LONGVARBINARY	-4
SQL_TYPE_LONGVARCHAR	-1
SQL_TYPE_NULL	0
SQL_TYPE_NUMERIC	2
SQL_TYPE_OID	-50
SQL_TYPE_REAL	7
SQL_TYPE_SMALLINT	5
SQL_TYPE_TIME	10
SQL_TYPE_TIMESTAMP	11
SQL_TYPE_TIMESTAMP_VERSION3	93
SQL_TYPE_TIME_VERSION3	92
SQL_TYPE_TINYINT	-6
SQL_TYPE_VARBINARY	-3
SQL_TYPE_VARCHAR	12
SQL_TYPE_WCHAR	-8
SQL_TYPE_WLONGVARCHAR	-10
SQL_TYPE_WVARCHAR	-9
SQL_XKEYDICT	6

Sounds Category

The global constants for the multimedia sounds are listed in the following table.

Global Constant	Integer Value
Snd_Asterisk	#40
Snd_Beep	-1
Snd_Default	0
Snd_Exclamation	#30
Snd_Hand	#10
Snd_Question	#20

The waveform sound for each sound type is identified by an entry in the **Sounds** section of the registry. (Assign sounds to system events by using the **Sounds and Multimedia** program item of the standard Windows Control Panel.)

SystemEvents Category

The global constants for JADE events for which system notifications are sent are listed in the following table.

Global Constant	Integer Value	Description
Any_System_Event	0	Object has been created, deleted, or updated
Object_Create_Event	4	Object has been created
Object_Delete_Event	6	Object has been deleted
Object_Update_Event	3	Object has been updated
System_Base_Event	#80000000	

SystemLimits Category

The global constants for JADE system limits are listed in the following table.

Global Constant	Integer Value
Max_Byte	#FF
Max_Identifier_Length	100
Max_Integer	#7FFFFFFF (equates to 2,147,483,647)
Max_Integer64	#7FFFFFFFFFFFFFFF (equates to 9,223,372,036,854,775,807)
Max_UnboundedLength	#FFFFFFFF
MaximumCollectionBlockSize	#40000
MaximumCollectionDisplay	32000
Min_Byte	#00 (equates to 0)

Global Constant	Integer Value
Min_Integer	#80000000 (equates to -2,147,483,648)
Min_Integer64	#8000000000000000 (equates to -9,223,372,036,854,775,808)

TimerDurations Category

The global constants for timer durations are listed in the following table.

Global Constant	Integer Value	Description
Timer_Continuous	0	Occurs continuously until it is disabled by the **Object**::**endTimer** method
Timer_OneShot	1	Occurs once only

For details, see the **Object** class **beginTimer** method, in Chapter 1 of the *JADE Encyclopaedia of Classes*.

UUIDVariants Category

The global constants for specifying the layout of a generated Universally Unique Identifier (UUID) are listed in the following table.

Global Constant	Integer Value	Description
VariantDce	2	Distributed Computing Environment, which is the scheme used by Qt C++ application development framework, and which is the recommended variant to pass to the **generateUuid** method
VariantMicrosoft	3	Reserved for Microsoft backward compatibility (GUID)
VariantNcs	1	Reserved for NCS (Network Computing System) backward compatibility

UserEvents Category

The global constants for user events are listed in the following table.

Global Constant	Integer Value
Any_User_Event	-1 (to subscribe to all user events)
User_Base_Event	16
User_Max_Event	Max_Integer (#7FFFFFFF, equating to 2147483647)