Developer's Course

Version 2025

jade
platform

Contents

Overview

The course is a five-day course aimed at people wanting to learn how to develop systems in the Jade Platform. There are no prerequisites, although experience in developing in another language would help. (For details about the modules additional to the legacy modules in this course, see "Additional Modules".)

The schedule is as follows.

- Monday
  - Module 1 - Installing the Jade Platform
  - Module 2 - Schemas
  - Module 3 - JadeScripts
  - Module 4 - Application Object
  - Module 5 - Primitive Types
  - Module 6 - Classes (part 1)
- Tuesday
  - Module 6 - Classes (part 2)
  - Module 7 - Root Object
  - Module 8 - Inheritance and Polymorphism
  - Module 9 - Collections
- Wednesday
  - Module 10 - Relationships
  - Module 11 - Forms
- Thursday
  - Module 12 - Applications
  - Module 13 - Exceptions
  - Module 14 - Notifications and Timers (part 1)
- Friday
  - Module 14 - Notifications and Timers (part 2)
  - Module 15 - Nodes, Processes, and Caches
  - Module 16 - Transactions and Locking
  - Module 17 - Printing

At the end of each module, there are a number of exercises for you to practice to build your skills. The exercises enable you to build a simplified banking system, which despite its simplicity, demonstrates many of the important features of the Jade Platform.

Module 1 Installing the Jade Platform

This module contains the following topics.

Introduction

You can download the Jade Platform software and obtain a free developer license from the Jade web site, at https://www.jadeplatform.com/developer-centre/downloads.

Note There is a separate download for the Jade Platform documentation in PDF (print) format.

Exercise 1.1 - Installing the Jade Platform

Follow these instructions to install the Jade Platform on your PC or laptop.

1. Request a free developer license by opening https://www.jadeplatform.com/developer-centre/licensing in your browser. A form is displayed for you to enter your information and then request the free license.

   Shortly you will be notified by a message to the e-mail address that you specified when requesting the license of your license name (which is case-sensitive) and license key (not case-sensitive). You can now install the Jade Platform.

2. On the https://www.jadeplatform.com/developer-centre/downloads web page, download the full Jade 2025 for Windows 64-bit (ANSI); that is, the **JADEwin64Ansi.exe** file.

3. Optionally, download the **2025 Documentation Package** (the **JADE Docs.exe** file) from https://www.jadeplatform.com/developer-centre/learn/documentation.

4. Run the **JADEwin64Ansi.exe** setup program and complete the steps of the installation with the actions specified in the following table.

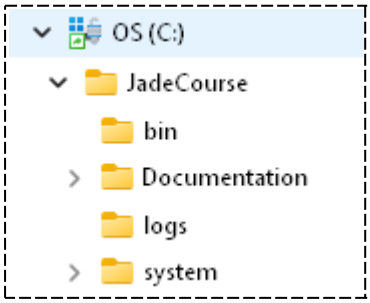
Step	Action
Welcome	Click the **Next** button.
License Agreement	Click the **Yes** button, to agree to the terms of the license.
Installation Type	Select the **Fresh Copy** option, and then click the **Next** button.
Setup Type	Select the **Development** option, and then click the **Next** button.
User Information	Enter the **License Name** and **License Key** from your license, and then click the **Next** button.
Select Installation Folders	Enter **C:\JadeCourse** in the **Install Directory** text box, and then click the **Next** button.
Select Program Folder	Enter **Jade Course** in the **Program Folder** text box, and then click the **Next** button.
Setup Completed!	Click the **Finish** button.

5. If you downloaded the **2025 Documentation Package**, run the **JADE Docs.exe** setup program and specify **C:\JadeCourse** as the **Destination** folder.

6. Check that files have been installed into the correct locations on your **C:** drive.

Jade Folders

The Jade Platform files are installed into a number of folders.



The **bin** folder contains the executable (.**exe**) and library (.**dll**) files.

The **Documentation** folder contains the help (.**pdf**) files in print format. (By default, context-sensitive help launches the web (HTML5) format documentation, as covered in "Context-Sensitive Help", in Module 4 of this course.)

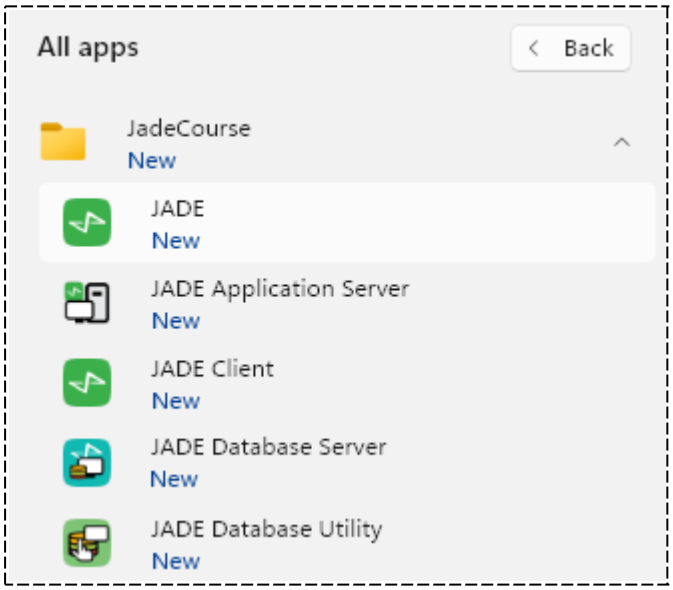
The **logs** folder contains the Jade message log file (**jommsg.log**) and error log files.

The **system** folder contains the database (.**dat**) files, the initialization file (**jade.ini**), and a folder for the database journal files.

Running the Jade Platform in Single User Mode

When you run the Jade Platform in single user mode, the database is automatically opened for your exclusive use.

The installation process creates a group of program shortcuts on the Windows Start menu. You can run the Jade Platform in single user mode by selecting the **JADE** shortcut from the menu.



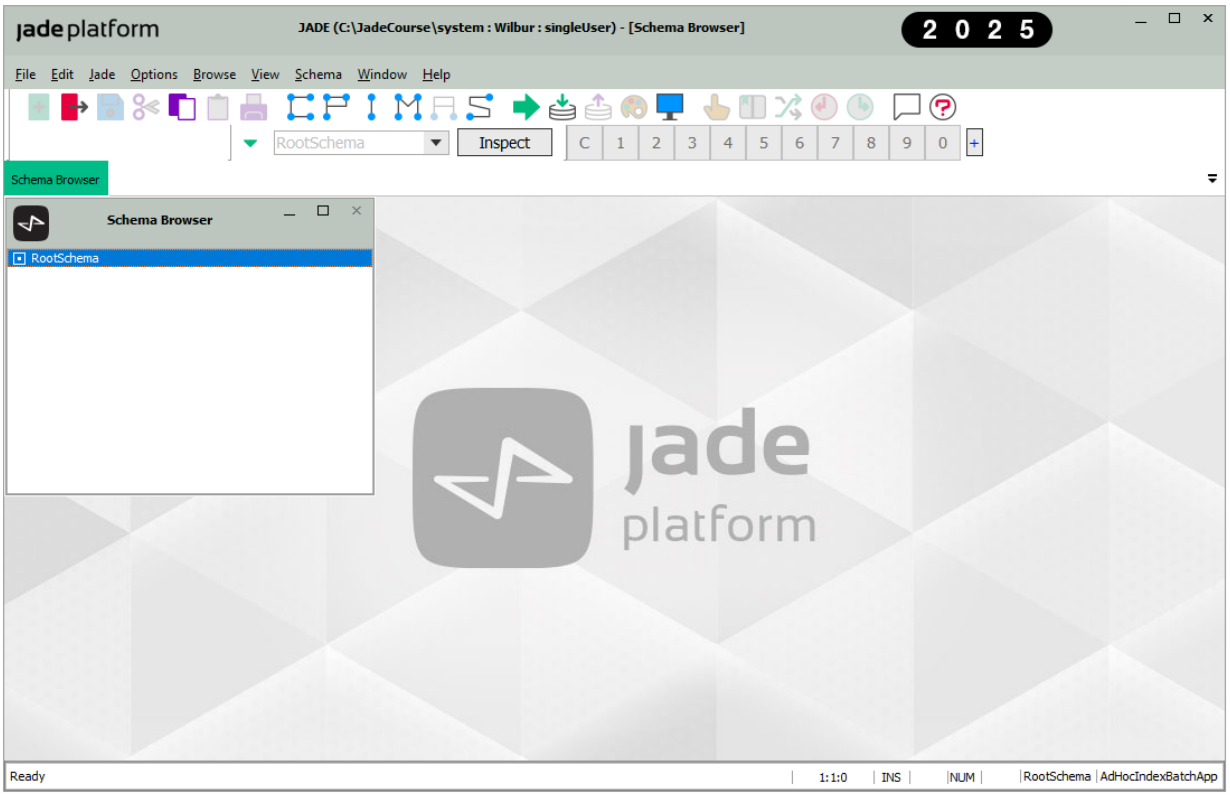
The first form that is displayed is the logon form.

Although you can add a security system to validate the user id and password, by default there is none. Enter your name in the **Username** text box, select the **Browse Classes** option, and then click the **Start** button.

If this is your first time starting the Jade Platform, three popup dialogs are displayed to help you get started. The Jade Release Notes dialog tells you about the new features in the 2025 release, the Tip of the Day dialog gives you handy tips and tricks relating to the Jade Platform, and the Start dialog helps you create your first schema.

You can close all of these dialogs, as this course will guide you through your first usage of the Jade Platform development environment.

You are now in the Jade Platform development environment, with the Schema Browser displayed.



The Jade Platform development environment is written in the Jade language. Jade provides you with a predefined set of classes that comprise a class hierarchy, or framework.

The Jade Platform development environment enables you to define classes, Jade methods, properties, constants, conditions, and form definitions. (For details, see Chapters 1 through 5 in the *Development Environment User's Guide*; for example, the 2025 product information is available from https://secure.jadeworld.com/developer-centre/Jade2025/OnlineDocumentation/.)

The integrated editor pane is displayed in the form specified by your editor options; that is, it is user-specific. Use the editor pane to:

- Define new methods or conditions in the selected class, primitive type, or interface
- Maintain existing methods and conditions using the integrated editor pane in a browser
- Compile methods and conditions
- Execute methods in the **JadeScript** class of the Class Browser (if selected)

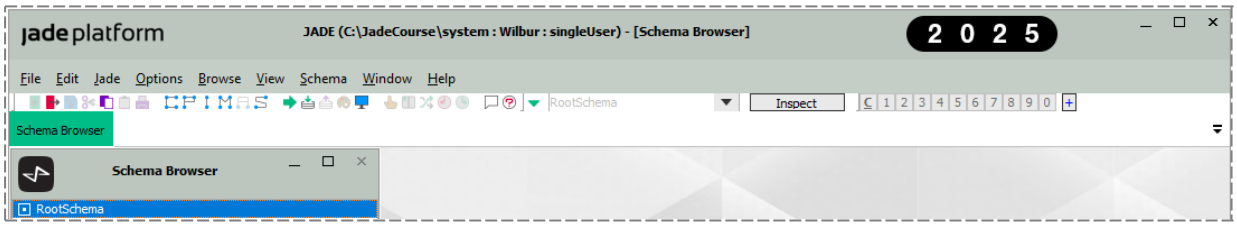
- Change or rename an entity (for example, a property, local constant, variable, or method parameter) selected within the body of a method in the editor pane

Jade provides hierarchy nodes, toolbar buttons, and menus, to enable you to navigate around the Jade Platform development environment. The Jade Platform development environment contains browser windows that provide a hierarchical structure of the browser elements. The Schema Browser is always opened on start-up.

You can access the browser windows from Browse menu commands or associated accelerator keys, or you can access some browsers from toolbar buttons or by using shortcut keys. For details about specifying your browser preferences, see "Maintaining Browser Options", in Chapter 2 of the *Development Environment User's Guide*.

To display smaller toolbar icons

1. Select the Options menu.
2. Select the **Preferences** command.
3. Click the **Browser** tab to display your browser options.
4. In the Toolbar Icon Size group box at right of the sheet, select the **Small** option button so that the background form looks similar to the following image. (Conversely, you could select the **Large** option button.)



When you select the display of small toolbar icons, the editor clipboard toolbar is displayed at the right of the toolbar. You can float this editor clipboard toolbar, which enhances the use of the internal Jade editor clipboards and the Windows clipboard, and you can view the clipboard text in bubble help by moving the mouse over the clipboard buffer.

To hide the display of the:

- Editor clipboard toolbar or the floated Jade Clipboard Text Contents form, uncheck the **Show Clip Board Toolbar** check box on the **Window** sheet of the Preferences dialog or select the **Show Clipboard Toolbar** command in the View menu.
- Quick inspect toolbar, uncheck the **Show Quick Inspect Toolbar** check box on the **Window** sheet of the Preferences dialog or select the **Show Quick Inspect Toolbar** command in the View menu.

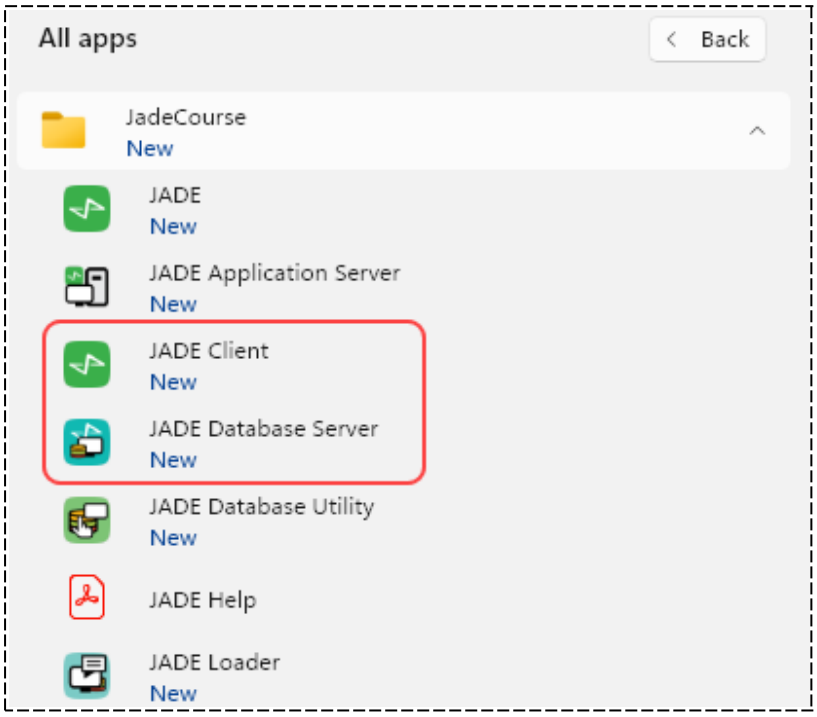
If the editor clipboard toolbar or the quick inspect toolbar is docked in the toolbar of the main development environment window, hiding the main development environment window toolbar also hides the editor clipboard toolbar or quick inspect toolbar.

Tip You can also apply a light or dark color mode or change the skin, by selecting the **Preferences** command from the Options menu, and then selecting the color mode and skin that you want to use in the Color Mode group box and the **Select JADE Skin** combo box, respectively, at the lower right of the **Window** sheet of the Preference dialog. If you select **<None>** in the **Select JADE Skin** combo box, no skin is applied.

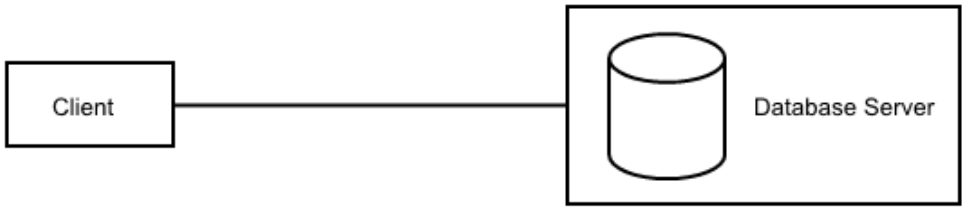
Running the Jade Platform in Multiuser Mode

When you run the Jade Platform in multiuser mode, the database server program must be running before any clients can connect. Many clients can connect to the database server at the same time, by using the TCP/IP network protocol.

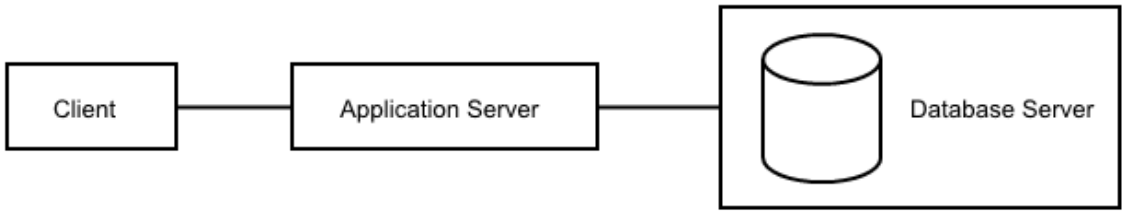
The Jade folder (**JadeCourse**, in this example) contains shortcuts for running the Jade Platform in multiuser mode.



Although you will be running the client and server on the same computer, the programs could be run on separate computers in a distributed way, as shown in the following diagram.



There is also a three-tier connection where a client connects to an application server, which connects to the database server.



Note The **JADE Database Server** program must always be started first.

By default, the **JADE Database Server** program is automatically minimized and an icon is placed in the system tray. The following image is an example of the maximized database server.

When the database server program is running, you can run the **JADE Client** program from the Jade folder. The logon procedure is identical to that for single user mode.

Exercise 1.2 - Running the Jade Platform

Run the Jade Platform in single user mode and multiuser mode by following the steps outlined in previous sections.

Development and Run Time

The multiuser architecture for Jade development (database server, application servers, and clients) is the same as for running applications developed in the Jade Platform. This is hardly surprising, as the Jade Platform development environment *is* a Jade application.

Files for the Course

Copy the **Files** folder to **C:\JadeCourse\Files** on your PC or laptop. If you are attending this course in person, this folder will be provided to you on a USB drive.

You can download the files from a USB drive; otherwise, you can download the files from https://secure.jadeworld.com/developer-centre/Education/DevCourse/JadeDevCourseFiles.zip.

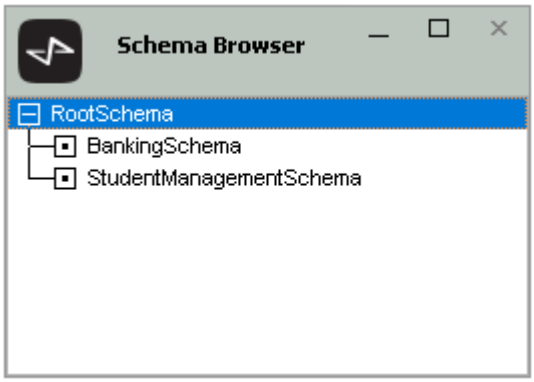
Module 2 Schemas

This module contains the following topics.

Introduction

Schemas provide a mechanism to organize classes. When you install the Jade Platform, the system classes are installed in the **RootSchema**. All other schemas inherit directly or indirectly from **RootSchema**; that is, the functionality of all system classes is available.

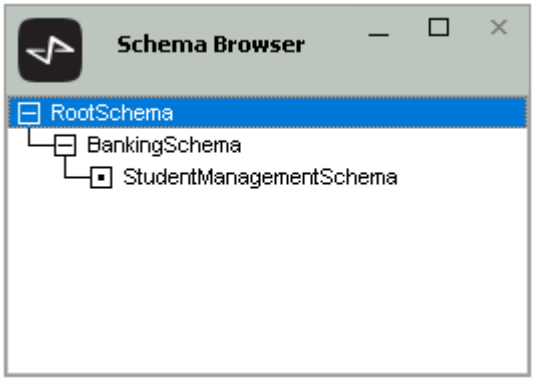
In the following image, a **BankingSchema** and a **StudentManagementSchema** have been added.



The banking classes are not available to the **StudentManagementSchema** and the student management classes are not available to the **BankingSchema**.

Note There is a *package* feature, which enables selected classes to be exported from one schema and imported into another.

One schema could have been added as a subschema of the other, as shown in the following image.



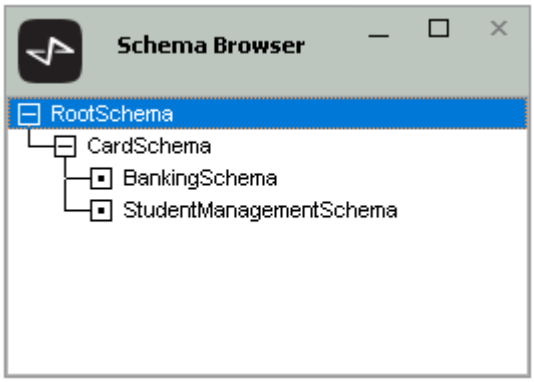
With this hierarchy, the **StudentManagementSchema** inherits all of the classes from the **BankingSchema** along with the system classes from **RootSchema**. This probably does not make a lot of sense.

Note Inheritance works only in the downwards direction, so the **BankingSchema** would not inherit classes from the **StudentManagementSchema**.

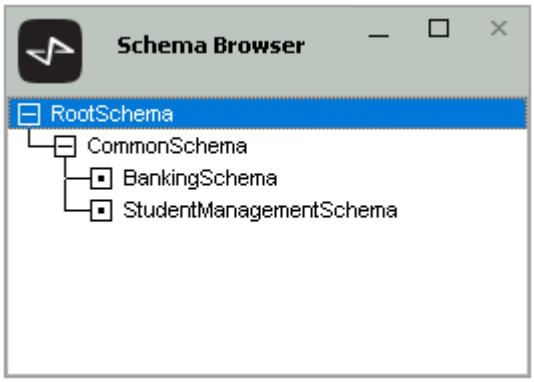
Jade Care is the group within Jade Software Corporation that develops tools to manage Jade systems (and other technologies).

Jade applications that are managed with JadeCare must have the JadeCare Start class library (also known as **CardSchema**) installed as a superschema of each application. It is available to all Jade users who can utilize the classes and applications in the **CardSchema.scm** and **CardSchema.ddx** files in their own systems. The functionality for exception handling, logging, FTP, LDAP, and so on, adds to that available from **RootSchema**. **CardSchema** can be downloaded with a free license from the Jade web site. For more information, see https://www.jadeplatform.com/developer-centre/extensions.

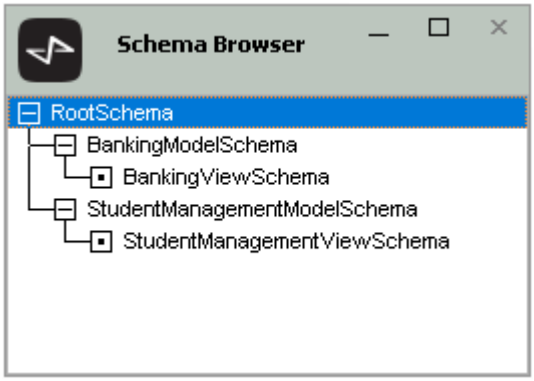
In the following schema hierarchy, **CardSchema** functionality is made available to the **StudentManagementSchema** and to the **BankingSchema**.



Alternatively, you could create a schema containing your own generically useful classes, as shown in the following image.



The *model* (that is, database-related) classes can be separated from the *view* (that is, application-related classes) with the following schema hierarchy.



Other Browser Windows

In the Schema Browser, when you select a schema to work with, you can then open other browser windows for that schema; for example, a Class Browser, which you can use for adding classes to the schema.

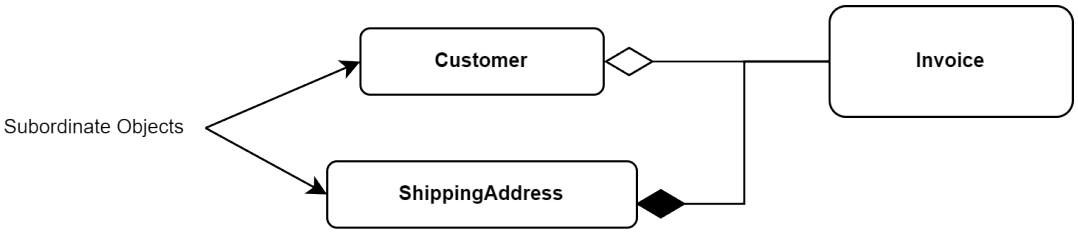
To open a Class Browser, click the **C** button from the Jade Platform development environment toolbar or use the Ctrl+B shortcut keys.

Integrating the Transaction Agent Framework (TAF)

This section discusses a recommended approach to persisting objects using the Jade Platform and it provides examples of a Transaction Agent Framework (TAF) that you can use in your own applications so that you can implement all create, update, and delete functionality using a TAF in your own Jade applications.

Tip For code examples, see "Part 5 -Transaction Agent Framework" in the *Erewhon Demonstration System Reference*, and for details about loading the Transaction Agent Framework so that you can integrate TAFs in your schemas, see "Exercise 2 - Loading Transaction Agent Framework (TAF) Classes", in Module 6.

The following table lists the acronyms (pronounced as a word), initialisms (pronounced as a series of letters that are abbreviations derived from the letters of the words they represent), and terms used in this section.

Entity	Description
CRUD	Create, Read, Update, and Delete.
TA	Transaction Agent.
TAF	Transaction Agent Framework.
TI	Transaction Implementor.
UML	Unified Modeling Language.
Modify	Targeted update that sets specific properties on an object.
Persistent object	Object that exists in the database. Also referred to as *persistent storage*.
Subordinate object	Child object referenced in a parent object; for example, an **Invoice** can contain a reference to a **Customer** object and can also create its own instance of an **Address** object to store the shipping address.
	
Transaction Agent	Class responsible for persisting an object.

For details about the Transaction Agent Framework (TAF), see the following subsections.

Best Practice Guidelines

The following table lists some recommended best practices and coding styles recommended when developing Jade applications.

Entity	Guideline
Collection references	References to collections should be prefixed with **all**. For example, a **Client** class may have a reference to a collection of sales. The reference to the **Sales** collection in the **Client** class should be named **allSales**.
Global constants	Global constants should be used in place of arbitrary values. For example, rather than referring to exception 1048 (*Update outside transaction*), you can create an **Exceptions** category for your schema in the Global Constants Browser and then create a global constant called **UpdateOutsideTransactionException** with a 1048 definition. Global constants can be accessed using the Ctrl+G keyboard shortcut.

Entity	Guideline
Inverse references	Inverse references automatically delete child objects when the parent object is deleted. In most cases, the manual update is on the **my** reference and the automatic update is on the **all** reference. (It can be useful to remember this as "**M**=**M**anual/**m**y and **A**=**A**utomatic/**a**ll".) When the single reference is updated, the collection is automatically updated. It is recommended that you create inverse references whenever a parent object owns a child object or collection of objects.
Object references	References to objects should be prefixed with **my**. For example, a **Sale** class that contains a reference to a **Client** object should be named **myClient**.
Property / properties	Generic name to represent an attribute or reference property.
Self	Use the **self** system variable (keyword) when referencing a property, method, or control from within the same class instance.
Separation of concerns	Methods should be responsible only for performing a specific task. If a method is performing more than one task, it may need to be split into two or more methods. Separating logic improves code readability, unit testing activities, and debugging.
Transient objects	A *transient* object is an object that is local to the Jade process and cannot be created or accessed by another Jade process. Transient objects are stored in a transient cache. When the transient cache gets full, the least-recently used transient objects overflow to an unbound transient database. Because this database is unbound, transient leaks can cause significant amounts of disk space to be used. For this reason, transient objects must be manually deleted. Deleting transient objects is usually performed in the epilog of the method that created the transient object.

Transaction Agent Framework (TAF) Overview

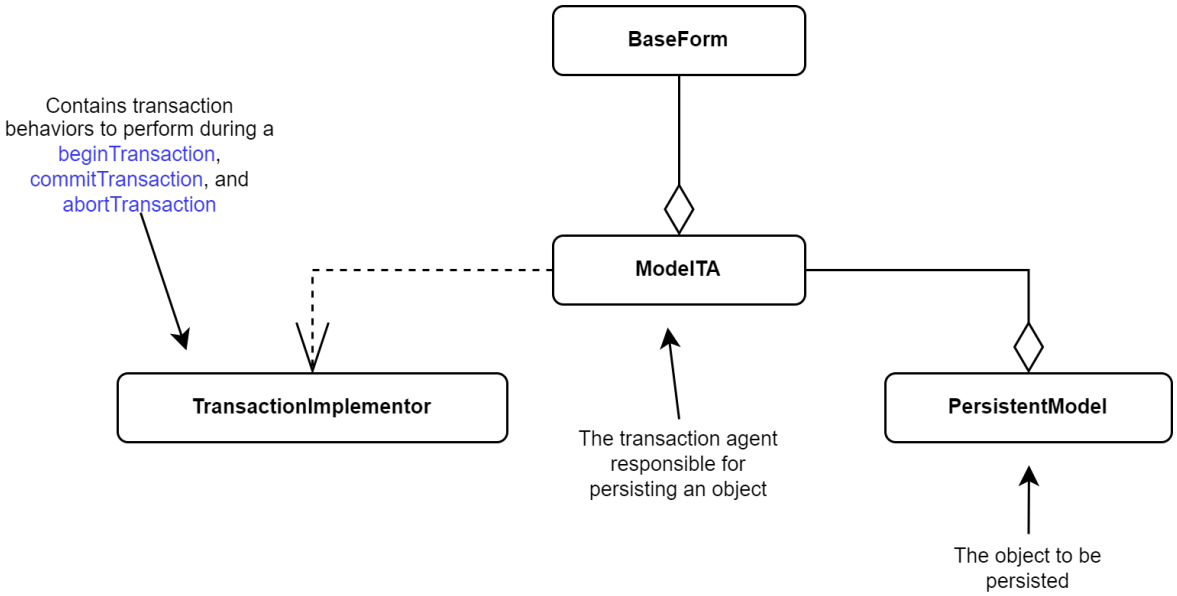
This section provides a basic high-level overview of the framework. Detailed examples and descriptions of the methods used in this section are provided in "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

- What is the Transaction Agent Framework (TAF)?
- Why is a TAF Needed?
- Where Should the TAF Reside?
- How Does the TAF Work?

What is the Transaction Agent Framework (TAF)?

The Transaction Agent Framework (TAF) is a set of classes that work together to perform persistent storage of objects.

The four main classes of the TAF are **PersistentModel**, **ModelTA**, **TransactionImplementor**, and **BaseForm**. As the following diagram shows, these classes work together to commit transactions into a Jade database.



The following table summarizes the four main classes.

Class	Description
PersistentModel	Subclasses of this class represent the objects to persist. They could represent a physical entity such as a **Person** or a concept like a **Sale**, which are essentially objects being saved in the database.
ModelTA	Subclasses of this class are the transaction agents responsible for performing persistent operations. Each **PersistentModel** subclass requires its own transaction agent subclass to be created. For example, when creating a transaction agent for a **Client** object, the transaction agent should be named **ClientTA**. The **ClientTA** class will contain the same properties.
TransactionImplementor	These classes are injected into various transaction agent methods through dependency injection, to determine the type of transaction behavior to perform; for example, begin, commit, and abort operations.
BaseForm	This class contains properties and methods used to connect forms and user interfaces with the TAF. All child forms should inherit this class, and therefore it should be the top-level class of the **Form** object. The **BaseForm** class may not be needed in situations where data is being persisted programmatically; for example, when loading data from a text file or in a situation where a user interface may not be required. For details, see "Manually Persisting an Object" under "How Does the TAF Work?", later in this document.

Why is a TAF Needed?

Software requirements are constantly changing, so new methods and properties may need to be introduced at any stage of development. What the client initially requested is often different from their actual needs.

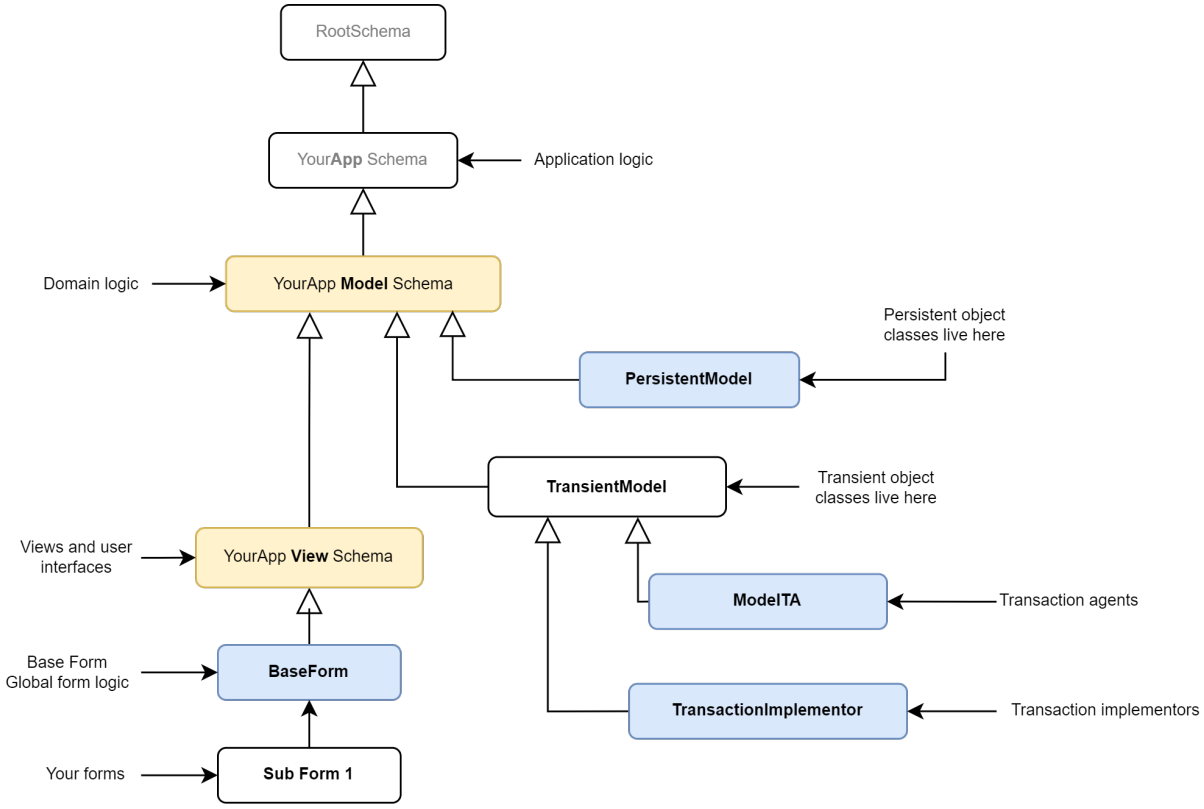
Due to the high possibility of change, software should be written in a way that is open for extension and closed for modification. Simply put, we don't want to modify existing working code in order to develop additional requirements; instead we want to extend the code base with the new logic. This is referred to as the *Open/Closed Principle*.

The Transaction Agent Framework does not use parameterized functions or constructors to create objects, because adding additional parameters when requirements change is likely to cause changes that break existing code. Instead, the TAF creates objects based on the properties stored in the transaction agent class. This method of creating objects not only eliminates parameterized functions and constructors but allows additional properties and logic to be added into the code base without affecting the original code.

Where Should the TAF Reside?

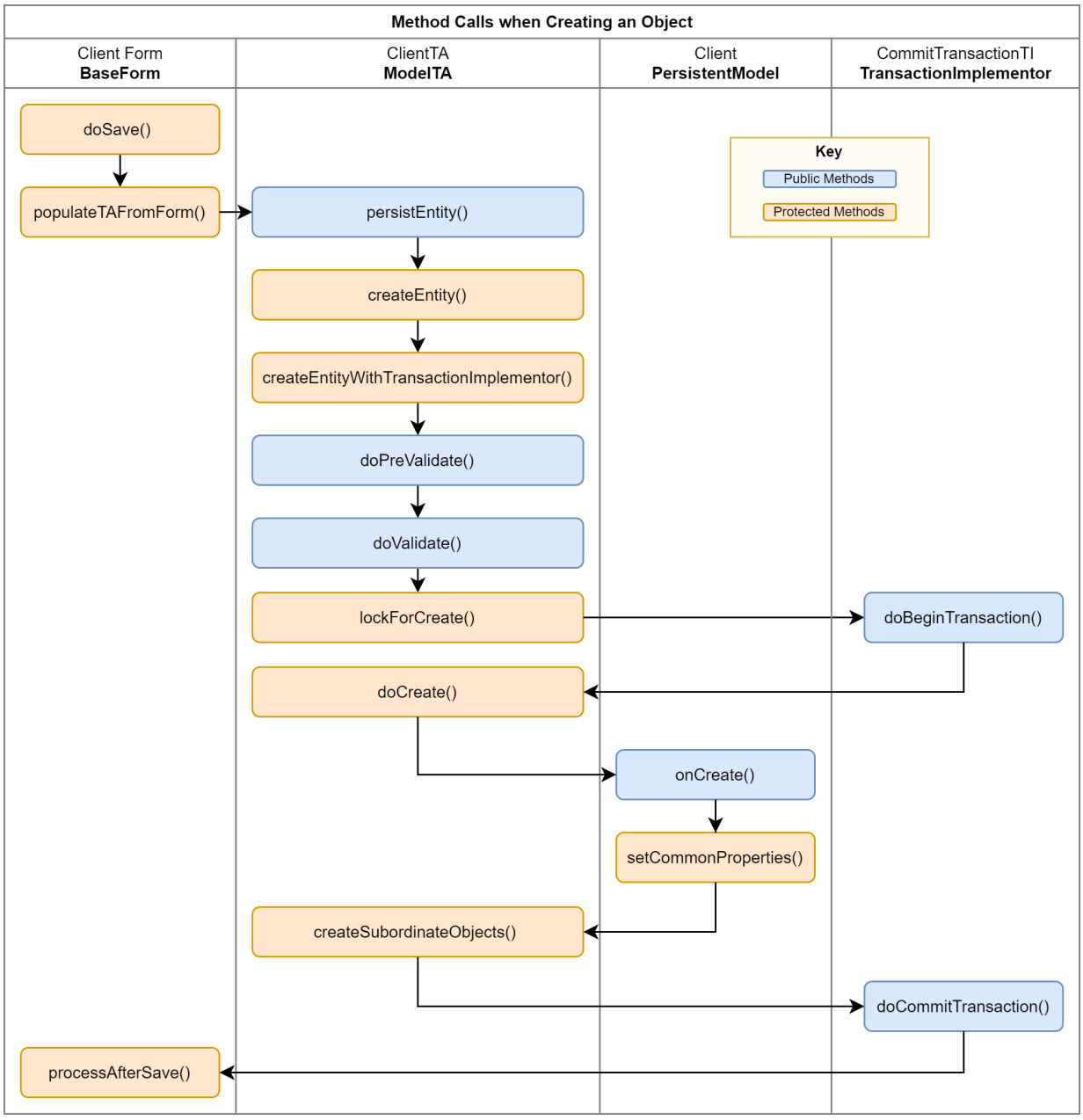
The TAF should be part of the **Model** schema (except for the **BaseForm** class residing in the **View** schema).

The following diagram shows a recommended setup of the Transaction Agent Framework in a standard Jade application.



How Does the TAF Work?

The following diagram shows the process of a **Client** object being created and persisted using the TAF.



Details about the methods in the diagrams in this section are described in "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

Manually Persisting an Object

The Transaction Agent Framework does not have to use the **BaseForm** class to persist objects, as it can be done manually and may be required when creating unit tests or loading data from a text file.

To persist an object without using the **BaseForm** class, transaction agents need to be created as a transient object and populated manually by setting the properties directly or by passing in an object into the **ModelTA** class **populateFromObject** method.

The following method is an example of how to persist an entity by manually populating the transaction agent properties.

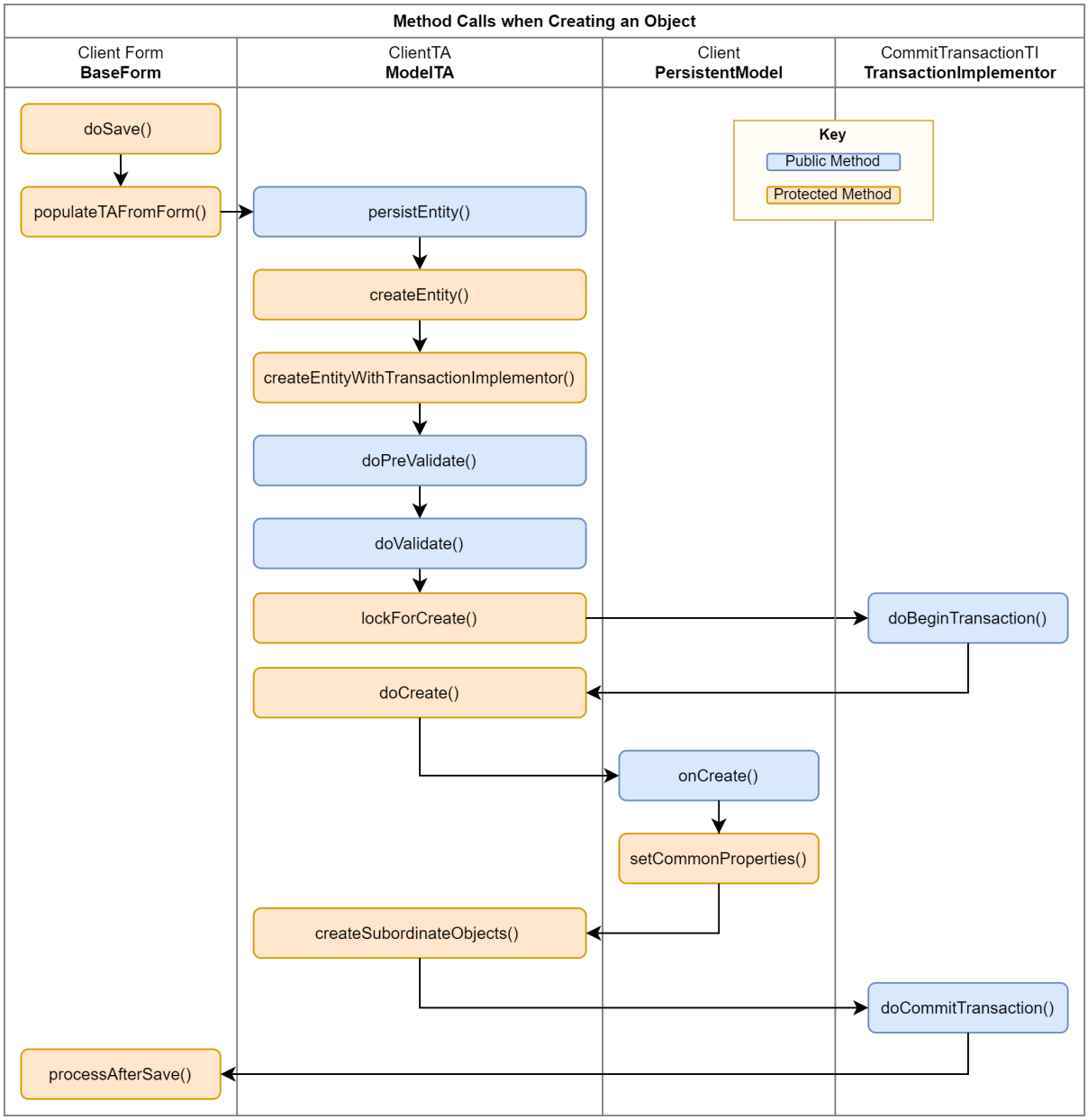
```
createClient();

vars
    clientTA : ClientTA;
    addressTA : AddressTA;
begin
    create clientTA transient;
    create addressTA transient;

    clientTA.name := "Clive Entworth";

    addressTA.street := "555 Fake St.";
    addressTA.city := "Dunedin";
    addressTA.country := "New Zealand";
    addressTA.phone := "555 5555555";
    addressTA.fax := "555 4444444";
    addressTA.email := "CliEnt@E.mail";
    addressTA.webSite := "www.website.com";
    addressTA.myModelTA := clientTA;

    clientTA.persistEntity( TransactionType_Persist );

epilog
    delete clientTA;
end;
```

This example deletes the **ClientTA** transient in the epilog but does not delete the **AddressTA**. This is because the **AddressTA** transient object will be automatically deleted because of the parent/child relationship being set when the reference was created. One of the major benefits of the TAF is ensuring that references are always inversed correctly.

Caution Remember to delete transient objects! Transient objects are stored in a transient cache. When the transient cache gets full, the least-recently used transient objects overflow to an unbound transient database. Because this database is unbound, transient leaks can cause significant amounts of disk space to be used. For this reason, transient objects must be manually deleted, which is usually done in the epilog of the method that created the transient object.

Creating Objects using the BaseForm Class

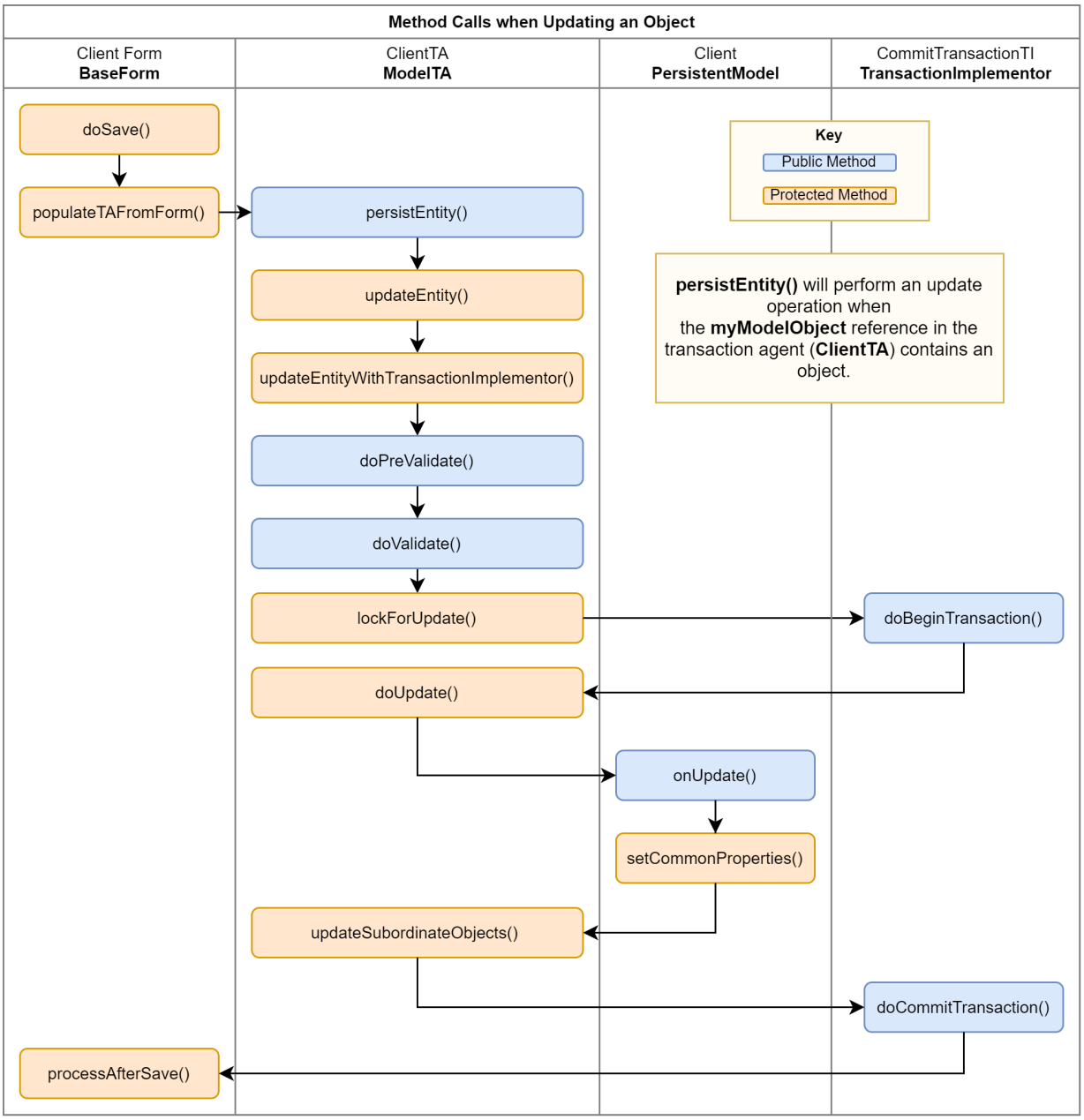
Persistent objects should be created using the TAF, by calling the **BaseForm** class **doSave** method.



Details about the methods in the diagrams in this section are described in "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

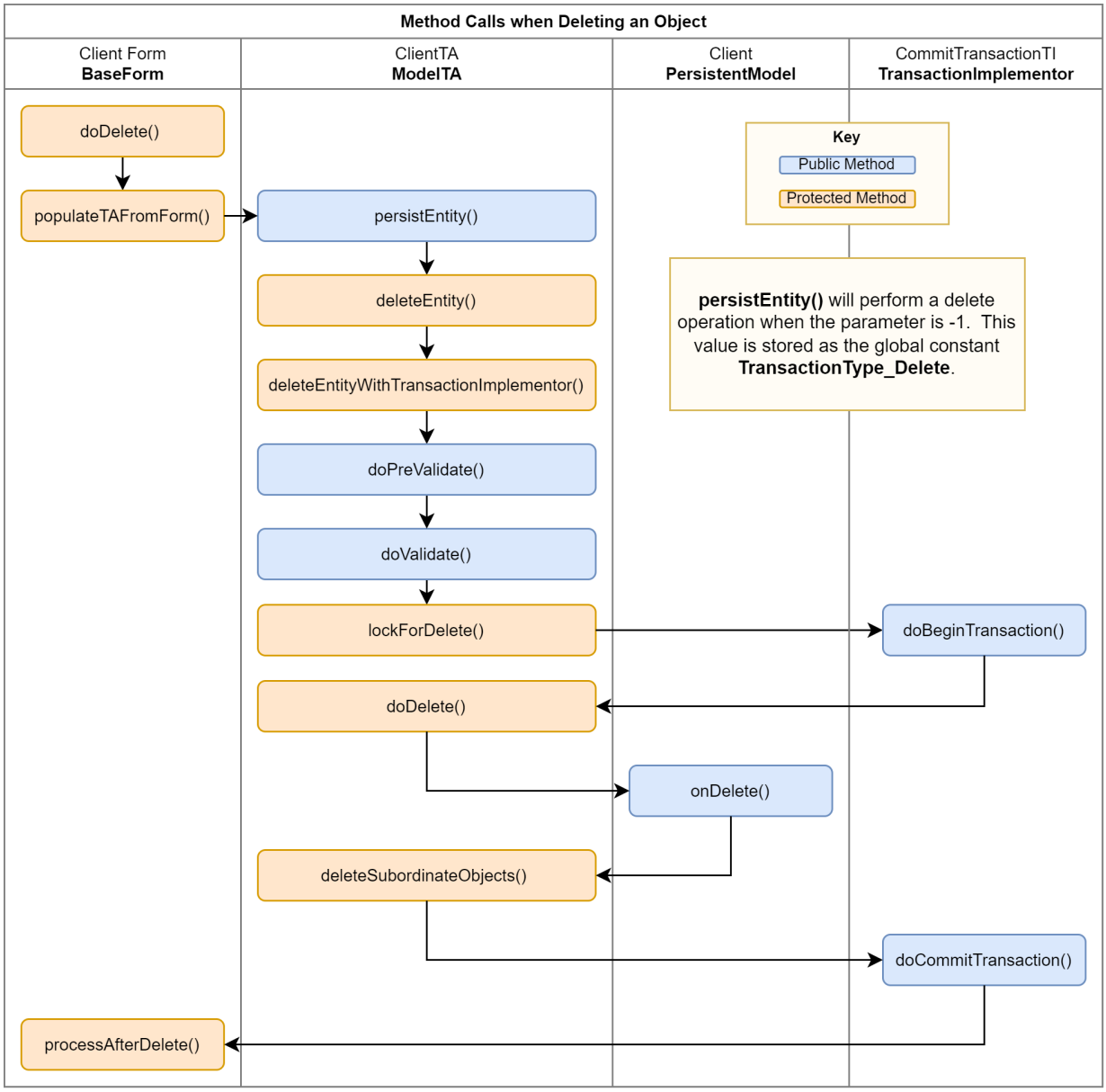
Updating Objects using the BaseForm Class

Updating objects is performed in exactly the same way as creating an object, by calling the **doSave** method on the **BaseForm** class. The transaction agent will check the value of the **myModelObject** parameter in the **persistEntity** method and perform an update if the model object value is not **null**.



Deleting Objects using the BaseForm Class

Deleting objects is performed by calling the **BaseForm** class **doDelete** method.



Reading Data

The transaction agent is not responsible for read operations when the object does not require persistence. Objects being used for display purposes should be retrieved using standard Jade practices.

Locking Objects

You should exclusive-lock persistent objects in a consistent order, including any collections which will be updated by automatic inverse maintenance, to avoid deadlocks when multiple processes attempt to update the same object or attempt to create, update, or delete instances of the same class. Obtaining the exclusive locks in a consistent order will ensure other processes queue behind the process that currently holds the exclusive locks, therefore removing the opportunity for a deadlock to occur.

An example of when locking may be required is an application requiring objects to remain unmodified while an operation is carried out; for example, a trial balance in which account objects are share locked before reading the balance, to guarantee that the latest edition of each account is used. The shared locks are held until the trial balance calculation is complete.

Note Share locking an object does not prevent other processes accessing it, but it *does* prevent them updating it.

Locking a Collection

Locking a collection is used to avoid objects being added or removed, or keys changing in a dictionary collection (member objects in a dictionary can always be updated). A collection should be exclusive locked when performing create, update, and delete operations if a property in the respective object is being used as a key or inverse.

To find the relevant collections for an object, select the *key* property (that is, it will have the key symbol displayed at the left of the property name) in the Properties List of the Class Browser and then click the **Property Details** tab above the editor pane, paying particular attention to the **Used as a key in** and **Inverses** details.

Locking Collection Objects Before a Create Action

All collections that will be updated by automatic inverse maintenance to add the object being created into that collection need to be exclusive locked in a consistent order, to remove the opportunity for a deadlock to occur. This is especially important if the object is generating unique identifiers from the collection.

There is no need to lock collections that use a key path (Key Path is a dictionary key that is not embedded on the object but derived from member objects) for the created object (for example, a **Category** object) because no other objects will reference the new object in their key path and no updates to these collections will happen.

Locking Collection Objects Before a Delete Action

Any collections that contain the object being deleted will need to be exclusive locked in a consistent order prior to deleting the object, to remove the opportunity for a deadlock to occur.

If the object to be deleted is being used as a key path, a check should be performed to ensure that no references to the object exist before the object is deleted. For this reason, there is no need to lock collections that use the object as a key path.

Locking Collection Objects Before an Update Action

If an updated property in the object is used as a key path, any collections that use that key will require exclusive locking; otherwise locking the collections is not required.

PersistentModel Class

The **PersistentModel** class contains subclasses that represent an entity or abstract concept, and it requires database persistence.

Classes should be singularly named, as each instance represents a single object persisted in a database. For example, a class representing an employee should be named **Employee**; not **Employees**.

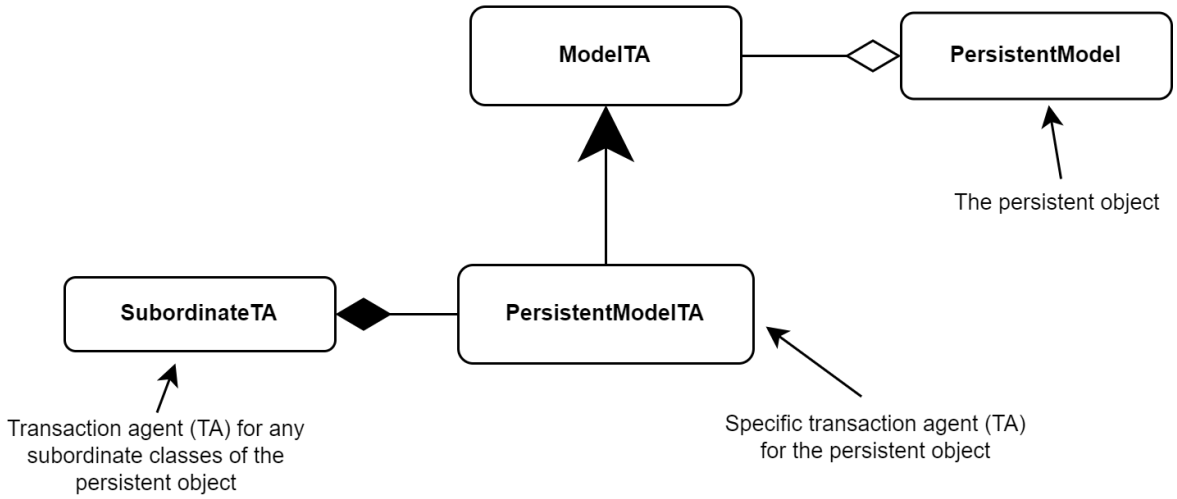
For details about the implementation of this class and its properties and methods, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

ModelTA Class (Transaction Agent)

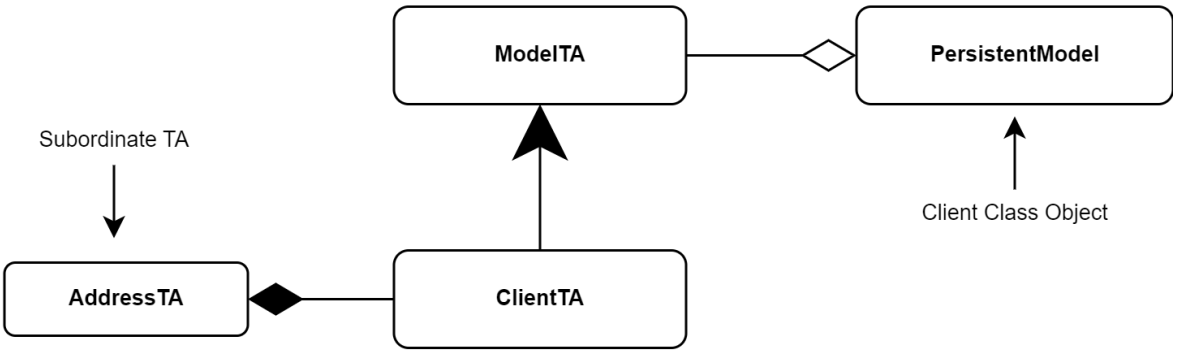
The Transaction Agent Framework is responsible for communication between the views and the database. Persistent operations such as create, update, modify, and delete must be executed using a transaction agent.

Each class that is stored as persistent data should have a transaction agent class containing all of the properties of the **PersistentModel** child class other than automatically maintained inverse references.

The following diagram shows an abstract implementation of the **ModelTA** class and its relationships with other classes.



To better present this concept, an actual implementation of the **ModelTA** class for a **Client** class is shown in the following diagram.



For details about the implementation of this class and its properties and methods, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

SubordinateTA (AddressTA)

Subordinate transaction agents (TAs) are used to create, update, modify, and delete subordinate objects in the parent class. In the previous diagram, the **ClientTA** class has a subordinate transaction agent named **AddressTA**, which is responsible for persisting an **Address** object.

PersistentModel

The **PersistentModel** class holds a **Client** object. **ClientTA** gets the **Client** object by calling the **getModelObject** method, which gets the value stored in the **myModelObject** property and is populated in the inherited **populateFromObject** method.

ModelTA Properties

The read-only properties defined in the **ModelTA** class are summarized in the following table.

Property	Value	Stores...
allErrors	HugeStringArray	A collection of errors (including validation errors) encountered during a persist operation. Errors will prevent the transaction agent from persisting a transaction.
allWarnings	HugeStringArray	A collection of all the warnings during a persist operation. Warnings will not prevent the transaction agent from persisting a transaction.
expectedEdition	Integer	The value of the expected object edition.
focusField	String	A global constant value for the field that receives focus if an error occurs.
lockedByProcessId	Decimal	The process identifier that has acquired a lock on the transaction agent.
lockedByTimeStamp	TimeStamp	The sever time when the transaction agent was locked.
modificationCode	Integer	A global constant value of the specific properties to update during a modify operation. A modify operation is a targeted update that sets only some of the properties on an object.
myModelObject	PersistentModel	The persistent object of type **PersistentModel** for which the transaction agent is responsible.

For details about these properties, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

ModelTA Methods

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

Method	Description
addError	Called in the **doValidate** method to add an error to the **allErrors** collection and assigns the field to receive focus
addWarning	Adds a warning to the **allWarnings** collection and assigns the field to receive focus
checkEdition	Checks if the current edition of the class is the expected version
clearErrors	Called at the start of every persist operation to clear the **allErrors** collection
clearErrorsOnSubordinateTAs	Empty method to be reimplemented on subclasses that use subordinate objects to clear all errors on the subordinate transaction agents
clearWarnings	Called at the start of every persist operation to clear the **allWarnings** collection
clearWarningsOnSubordinateTAs	Clears the **allWarnings** collection on subordinate transaction agents
copyErrors	Copies the errors from the value specified in the **ModelTA** parameter into the calling transaction agent's **allErrors** collection

Method	Description
copyWarnings	Copies the warnings from the value specified in the **ModelTA** parameter into the calling transaction agent's **allWarnings** collection
createEntity	Programmatically called to persist an object when the system is not in a transaction state
createEntityInTransState	Programmatically called to persist an object when the system is already in a transaction state
createEntityWithTransactionImplementor	Programmatically called when creating an object
createSubordinateObjects	Called to persist subordinate objects when the system is in a transaction state
deleteEntity	Programmatically called to delete an object when the system is not in a transaction state
deleteEntityInTransState	Programmatically called to delete an object when the system is already in a transaction state
deleteEntityWithTransactionImplementor	Programmatically called when deleting an object
deleteSubordinateObjects	Programmatically called to delete subordinate objects when the system is in a transaction state
doAbortTransactionCleanup	Programmatically called to perform cleanup behavior if a transaction fails
doAbortTransactionCleanupForSubordinateObjects	Performs cleanup behavior for subordinate objects if a transaction fails
doCreate	Programmatically called to persist an object to the database
doDelete	Programmatically called to delete an object from the database
doModify	Programmatically called to modify a targeted property or subset of properties
doPreValidate	Empty method called before the **doValidate** method to perform pre-validation logic
doUpdate	Programmatically called to update a persisted object and its subordinates
doValidate	Validates the transaction agent properties prior to persisting an object
getFullErrorDetails	Returns a formatted string of all the errors in the **allErrors** collection separated by a carriage return/ line feed characters
getModelObject	Returns the object stored in the **myModelObject** property
getModelObjectClass	Returns the class type used by the transaction agent
hasErrors	Checks the **allErrors** collection to see if it contains errors
hasNoErrors	Checks the **allErrors** collection to see if it contains no errors

Method	Description
hasOnlySubordinatePersistentObjects	Declares the transaction agent to work with various subordinate objects and not a single persistent object or type
initialize	Initializes the transaction agent properties to an uninitialized state
lockForCreate	Manually places an exclusive lock on objects during a create operation
lockForDelete	Manually places an exclusive lock on objects during a delete operation
lockForModify	Manually places an exclusive lock on objects during a modify operation
lockForUpdate	Manually places an exclusive lock on objects during an update operation
modifyEntity	Programmatically called to modify a targeted property when the system is not in transaction state
modifyEntityInTransState	Programmatically called to modify a targeted property when the system is in transaction state
modifyEntityWithTransactionImplementor	Programmatically called to modify targeted properties
modifySubordinateObjects	Programmatically called to modify targeted properties on subordinate objects
persistEntity	Main method called to persist (create, update, modify, or delete) an object to the database when the system is not in a transaction state
persistEntityInTransState	Main method called to persist (create, update, modify, or delete) an object to the database when the system is in a transaction state
populateFromObject	Copies the **PersistentModel** object properties into the properties of the transaction agent
populateSubordinateObjects	Copies the **PersistentModel** object properties for the subordinate objects to the respective subordinate transaction agents
tryLockingObject	Attempts to place an exclusive lock on the object passed in as the value of the **pObject** parameter
updateEntity	Programmatically called to update an object when the system is not in transaction state
updateEntityInTransState	Programmatically called to update an object when the system is in a transaction state
updateEntityWithTransactionImplementor	Programmatically called to update an object by passing a **TransactionImplementor** as an argument to determine if the transaction should be committed to the database
updateSubordinateObjects	Programmatically called to update the subordinate objects

For details about these methods, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

TransactionImplementor Class

The TransactionImplementor (TI) classes contain specific behaviors to perform at different stages of a persistent transaction.

TI classes are injected into the ***xxxxxxx*EntityWithTransactionImplementor** methods (for example, **createEntityWithTransactionImplementor**) in the **ModelTA** class as a dependency, allowing the methods to call the same functions but perform different behaviors specific to the transaction implementor provided. This technique is referred to as *Dependency Injection*.

Each transaction implementor has different behavioral implementations for begin, commit, and abort operations.

For details about the implementation of this class and its subclasses, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

Beginning a Transaction

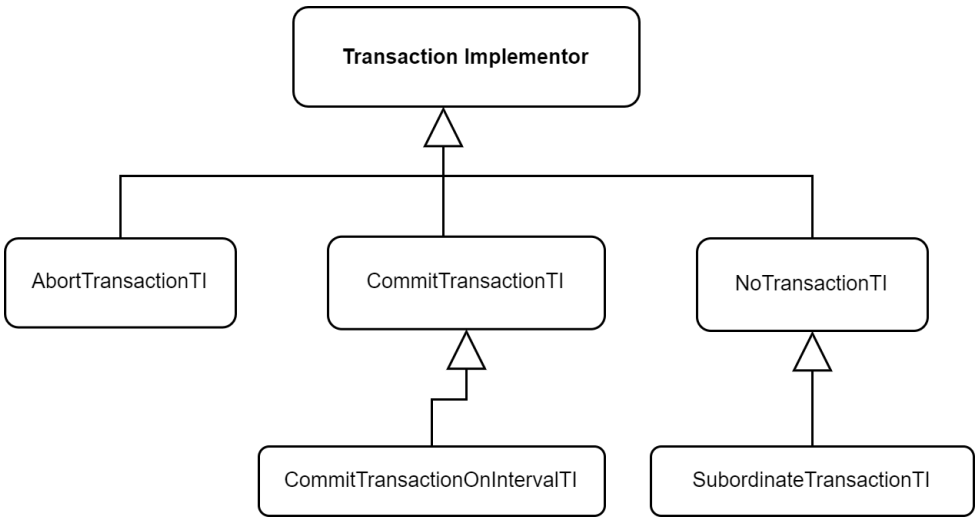
To perform a persistent operation in Jade (saving, updating, or deleting objects from a database), the system must first be placed into a transaction state. This is generally done by calling the **beginTransaction** instruction prior to performing persistent operations. This behavior is handled by the **doBeginTransaction** method in the TI.

Committing a Transaction

To commit a transaction to the database, Jade uses the **commitTransaction** instruction. This functionality is handled by the **doCommitTransaction** method in the TI.

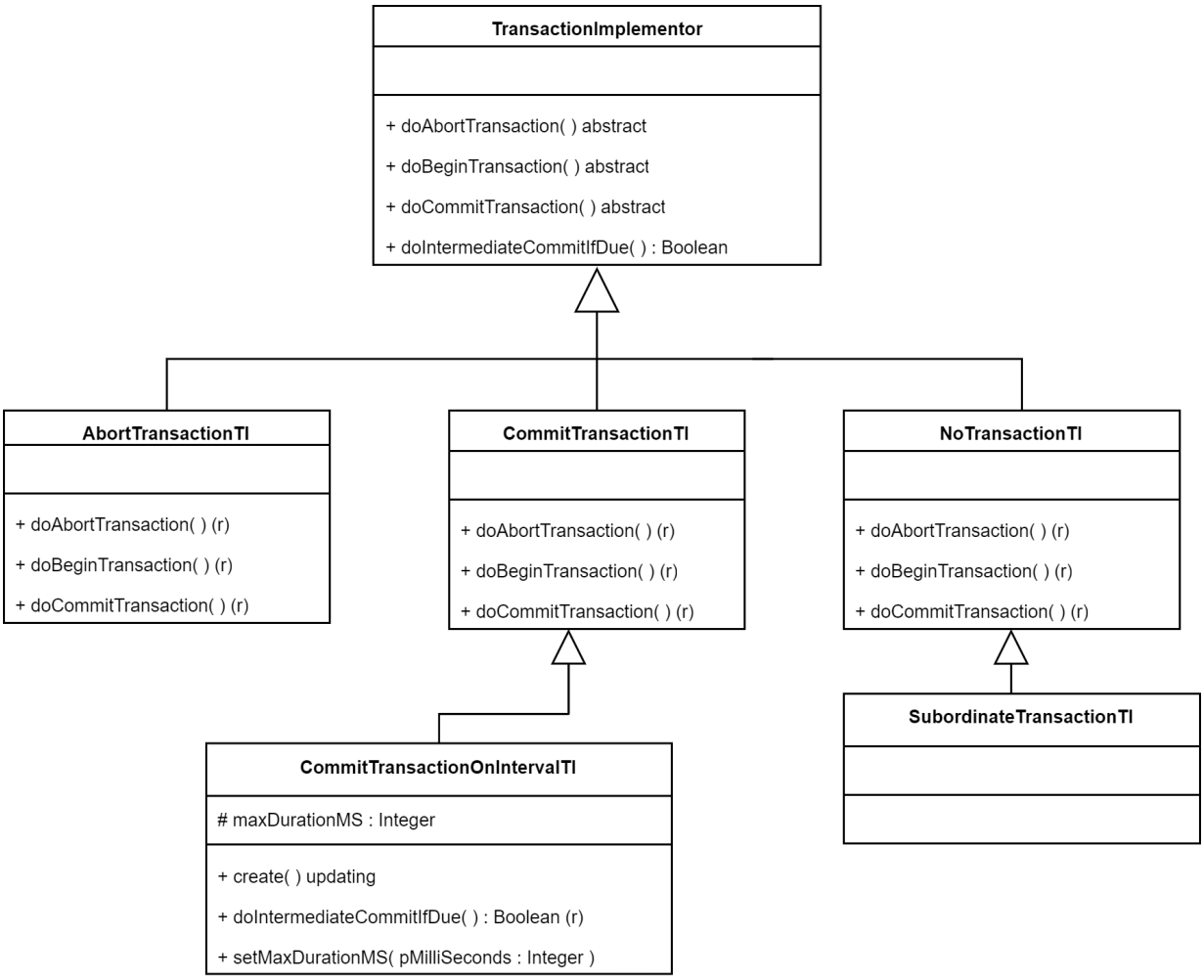
Aborting a Transaction

To abort a transaction (in the case of an error or other reason), Jade uses the **abortTransaction** instruction. This functionality is handled by the **doAbortTransaction** method in the TI.



TransactionImplementor Class Diagram

The following diagram shows the properties and methods defined in the **TransactionImplementor** class.



TransactionImplementor Abstract Class

The abstract transaction implementor class is reimplemented in child classes to ensure that the transaction methods are available in each child class.

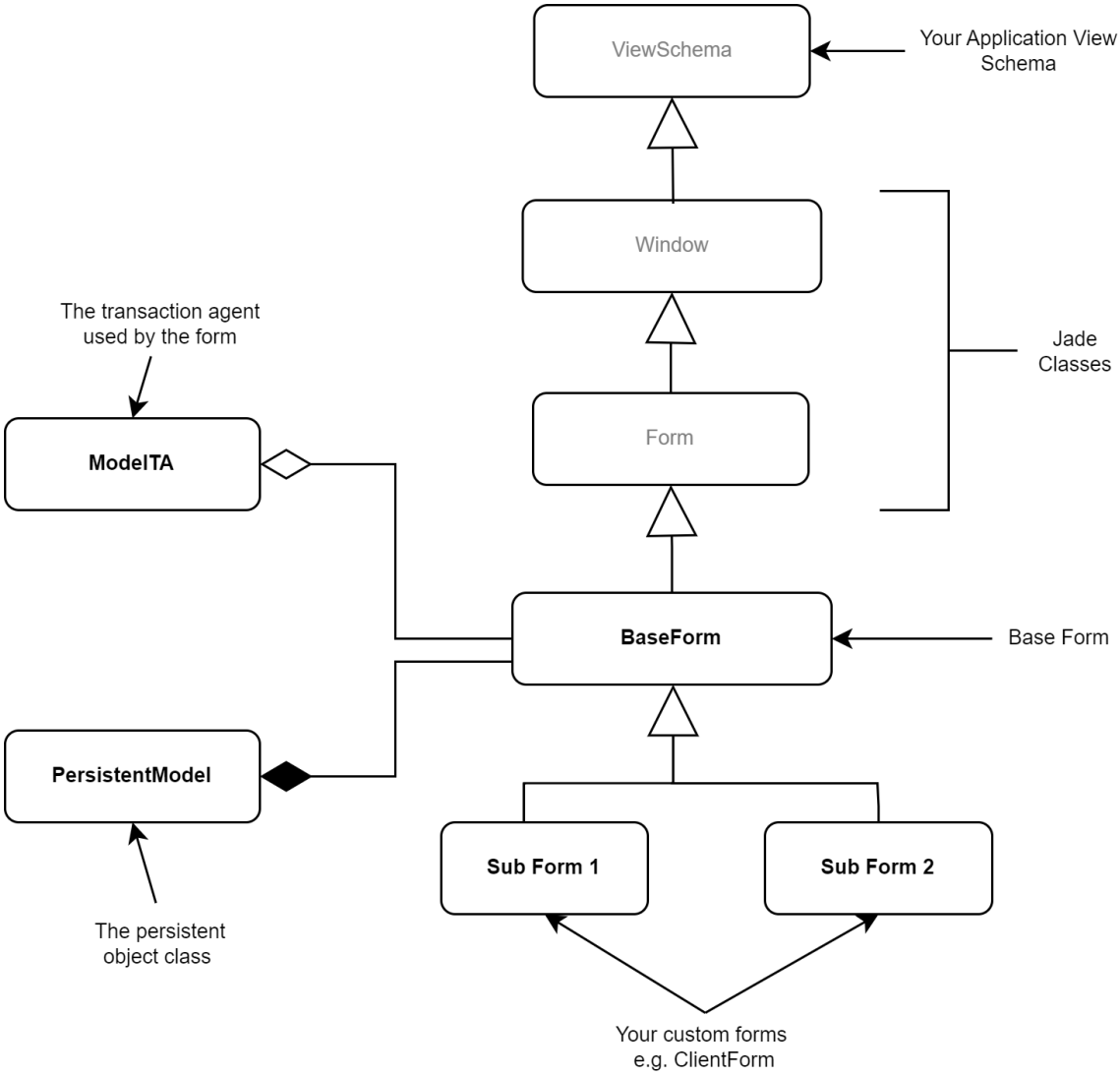
The **TransactionImplementor** abstract class public methods are summarized in the following table. (For details, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.)

Method	Return Type	Description
doAbortTransaction	Not applicable	Generally called to abort a transaction
doBeginTransaction	Not applicable	Generally called to start a transaction
doCommitTransaction	Not applicable	Generally called to commit a transaction
doIntermediateCommitIfDue	Boolean	For use on longer background tasks to commit periodically after an elapsed time

BaseForm Class

Most of the time, the user interacts with the system through forms. These forms need to work with the transaction agent through a **BaseForm** class. For details about the implementation of this class and its properties and methods, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

The **BaseForm** class should be the parent class of all forms that use the Transaction Agent Framework, and it generally does not contain any controls or menus but it *does* contain the properties and methods for interacting with the transaction agent.



The previous diagram shows the **BaseForm** class as a child of the Jade **Form** object. By convention, all user interfaces and views should be a subclass of a ***YourAppNameViewSchema*** schema.

BaseForm Properties

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

Property	Value	Stores the...
myCurrentObject	PersistentModel	Object for which the form and transaction agent are responsible
myTA	ModelTA	Transaction agent responsible for persisting and displaying the **PersistentModel** object

For details about these properties, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

BaseForm Methods

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

Method	Description
displayErrors	Displays the errors stored in the **allErrors** collection after a failed delete or save operation
displayObject	Displays the persistent object properties on the form
doDelete	Performs a persistent delete of the object stored in the **myCurrentObject** property
doSave	Performs a persistent save of the object stored in the **myCurrentObject** property
formLoad	Performs specific form load behavior
formUnload	Performs specific form unload behavior
getCurrentObject	Gets the **PersistentModel** object stored in the **myCurrentObject** property of the form
getTA	Returns the transaction agent stored in the **myTA** property
getTAClass	Returns the type of transaction agent used by the form
populateTAFromForm	Populates the transaction agent of the form with the form control values
processAfterDelete	Performs tasks after a successful delete operation
processAfterSave	Performs tasks after a successful save operation
setContextObject	Populate the **myCurrentObject** property of the form with the **PersistentModel** specified in the **pContext** parameter

For details about these methods, see "Part 5 - Transaction Agent Framework" in the *Erewhon Demonstration System Reference*.

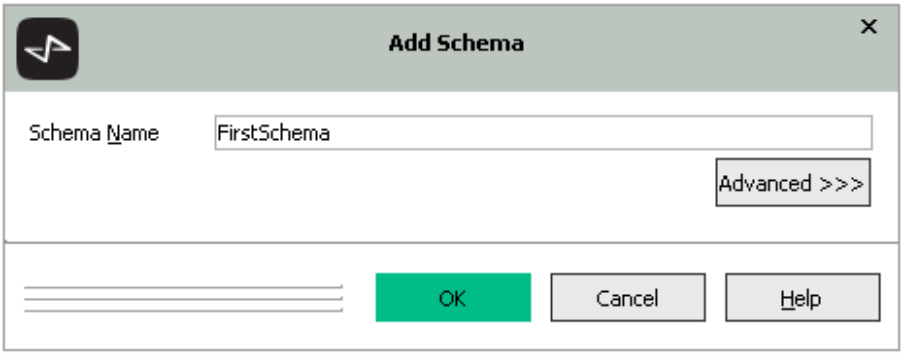
Exercise 2.1 - Adding a Schema

In this exercise, you will add a schema to be used for the early part of the course.

1. Select the Schema Browser by clicking the **S** button from the Jade Platform development environment toolbar or use the Ctrl+Alt+S shortcut keys.

2. Select **RootSchema** in the Schema Browser.

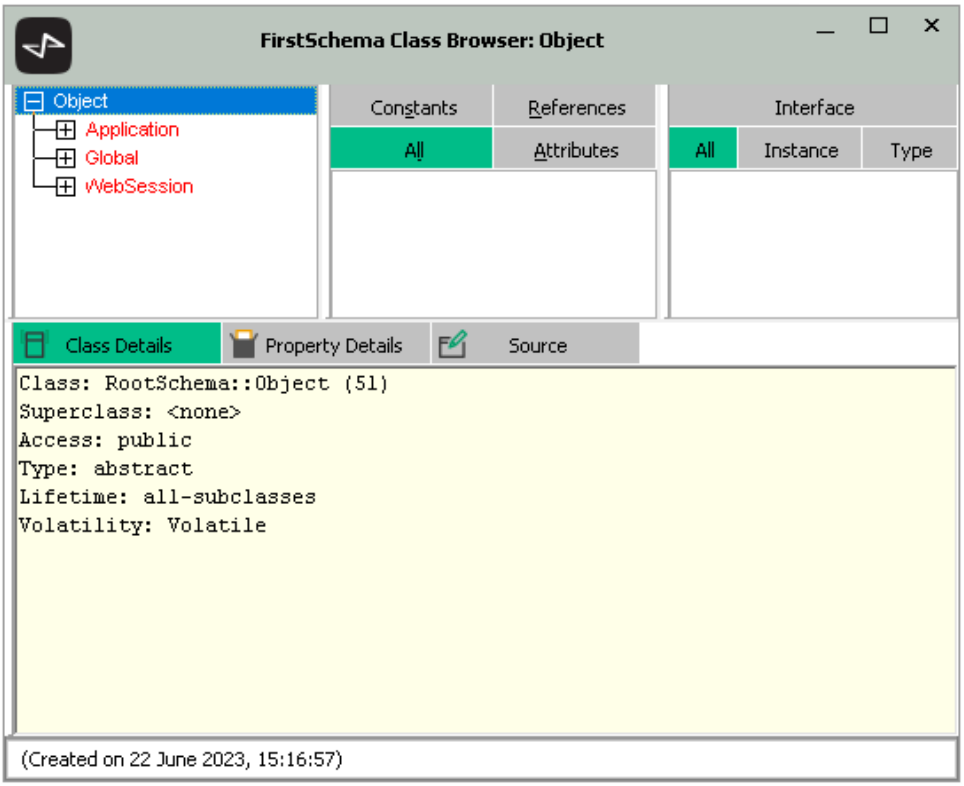
3. Add a schema by selecting the Schema menu **Add** command.
4. Enter **FirstSchema** as the name of the schema, and then click the **OK** button.



Exercise 2.2 - Opening a Class Browser

In this exercise, you will look at the classes in the two schemas in your system.

1. Open a Class Browser for the **FirstSchema**.



2. Open a Class Browser for the **RootSchema**.
3. Estimate the number of classes in **RootSchema**.

Module 3 JadeScripts

This module contains the following topics.

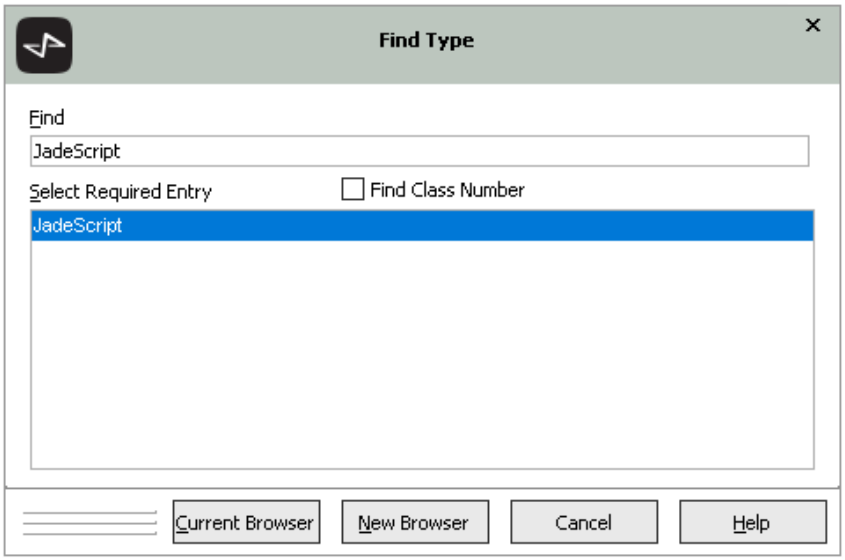
Introduction

This module has a number of exercises that introduce you to the syntax of programming in Jade. It introduces the **JadeScript** class, which is defined in the **RootSchema** and used by developers to write and execute methods directly from the Jade Platform development environment.

JadeScript methods are not designed to be part of a user application, but can be used to:

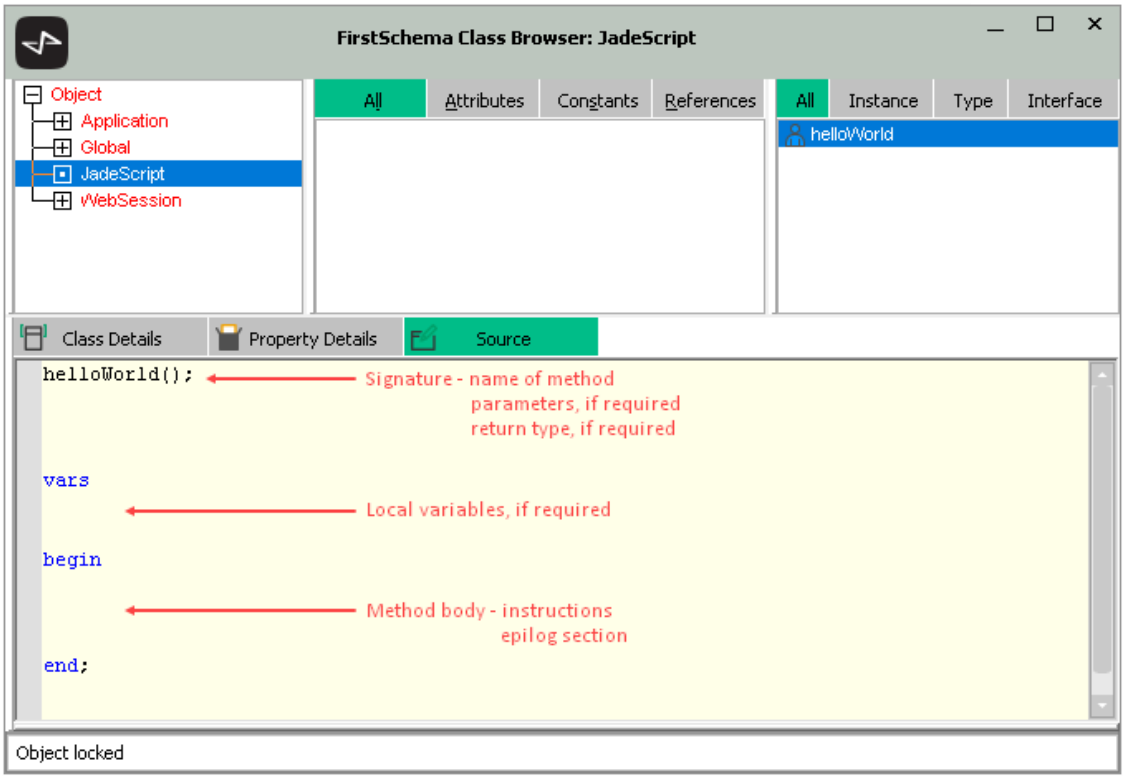
- Create, delete, and fix data
- Experiment, demonstrate, and test code

By default, the **JadeScript** class is not displayed because it is inherited from a superschema. To display the class in the Class Browser, press F4 or use the Classes menu **Find** command.



Structure of a Method

When you add a method to a class using the Methods menu **New Jade Method** command, a method skeleton is displayed in the editor pane ready for you to enter your code.



The top line is the method *signature*.

In the following example, the **canWithdraw** method for a bank account object determines whether there are sufficient funds to meet a proposed withdrawal.

```
canWithdraw(amount: Decimal): Boolean protected;
```

In this method signature:

- **canWithdraw** is the method name. Method names begin with a lowercase letter and contain no spaces.
- **amount** is the parameter, which is of type **Decimal**. It is the value of the proposed withdrawal.
- **Boolean** is the type of the value that must be returned by the method. It will be **true** if there are sufficient funds; otherwise **false**.
- **protected** is the method option. It can be called only by methods in the same class.

The method body can contain an **epilog** section with instructions that you want to be executed even if the method is aborted or exited from with an early **return** instruction. It is often used for tidy-up code; for example, deleting transient objects and changing the mouse pointer back to its default shape.

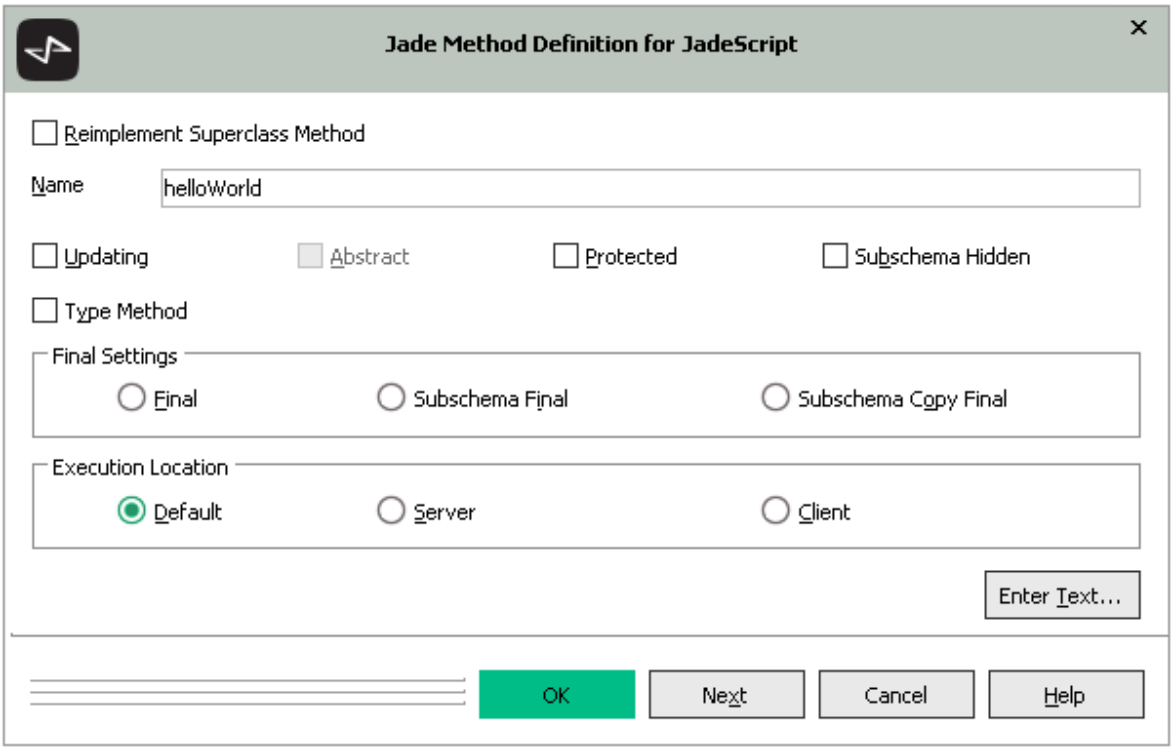
```
begin
    app.mousePointer := Window.MousePointer_HourGlass;
    // other instructions
epilog
    app.mousePointer := Window.MousePointer_Default;
end;
```

Exercise 3.1 - Hello World

In this exercise, you will write and execute a JadeScript method to display the traditional "**Hello World**" greeting. The **write** instruction writes a message to the Jade Interpreter Output Viewer window.

1. Open a Class Browser for the **FirstSchema**.
2. Find the **JadeScript** class.

3. Add a method to the **JadeScript** class by selecting the Methods menu **New Jade Method** command. Enter **helloWorld** as the name of the method, and then click the **OK** button.



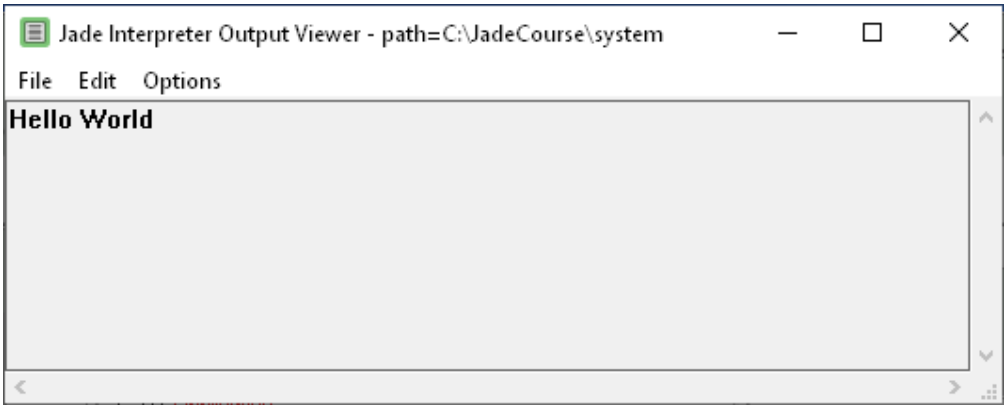
4. Enter the following code.

```
helloWorld();

begin
    write "Hello World";
end;
```

5. Compile the method by selecting the Methods menu **Compile** command or by pressing F8.

6. Execute the method by selecting the Jade menu **Execute it** command or by pressing F9.

   The greeting is then displayed in the Jade Interpreter Output Viewer window.



Tip In the Jade Interpreter Output Viewer, select the Options menu **Always on top** command to prevent the window from being hidden.

In this method:

- The **write** instruction is used to display information.
- Each instruction is terminated with a semicolon (**;**) character.

Exercise 3.2 - *read* and *write* Instructions

In this exercise, you will use the **read** instruction to enable the user to enter information into a User Input dialog.

- Create and execute a **displayYourName** JadeScript method, as follows.

```
displayYourName();

vars
    name: String;
begin
    read name;
    write "Your name is " & name;
end;
```

In this method:

- A variable of type **String** is declared in the **vars** section.
- The **read** instruction prompts the user to enter information, which is stored in the **name** variable.
- The concatenation operator, which is the ampersand (**&**) character, is used to join two strings in the output.

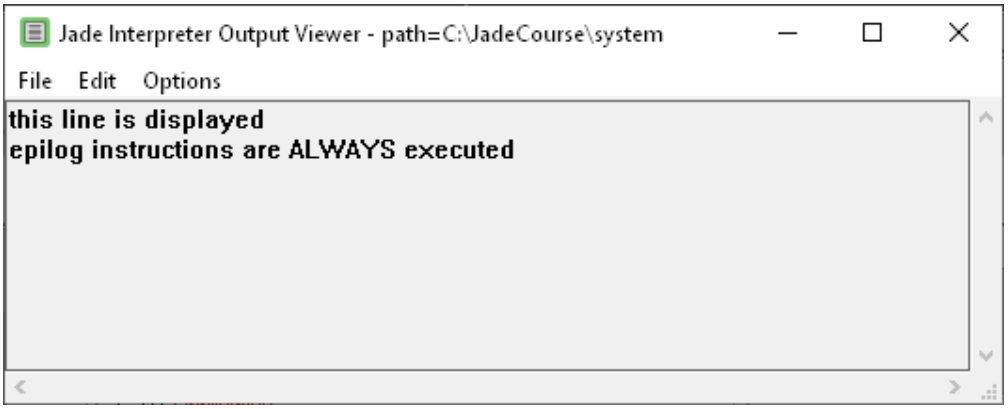
Exercise 3.3 - *return* and *epilog* Instructions

In this exercise, you will use the **return** instruction to exit from the method before all of the instructions have been executed. However, the instructions in the **epilog** section should always be executed.

1. Create and execute a **returnAndEpilog** JadeScript method, as follows.

```
returnAndEpilog();

begin
    write "this line is displayed";
    return;    // Exits from the method
    write "return instruction prevents getting to this line";
epilog
    write "epilog instructions are ALWAYS executed";
end;
```

2. Execute the method. Two lines are written to the Jade Interpreter Output Viewer window, as follows.



In this method:

- The **return** instruction exits from the method before all of the instructions are executed.
- The instruction in the **epilog** section is executed before the method returns.

Exercise 3.4 - Exceptions

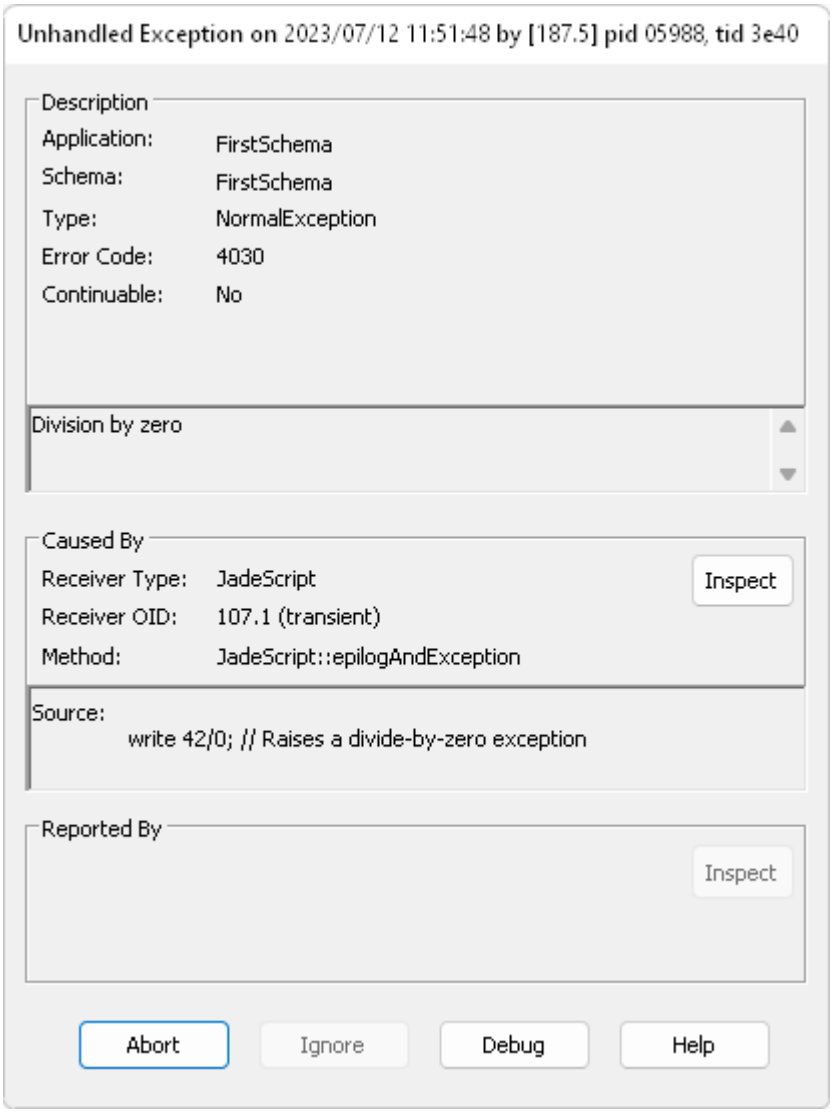
In this exercise, you will code an instruction that Jade cannot execute so that it therefore raises an exception.

When the **Abort** button is clicked on the Unhandled Exception dialog, the instructions in the **epilog** section are always executed before the method is removed from the stack.

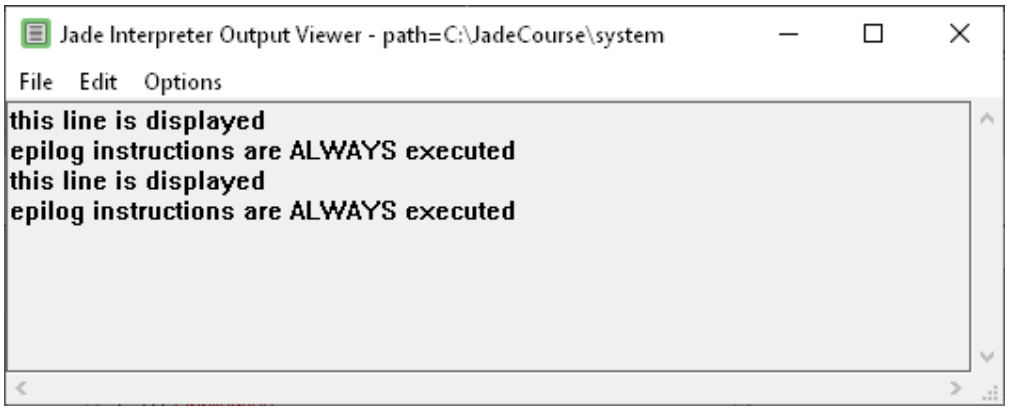
1. Create and execute an **epilogAndException** JadeScript method, as follows.

```
epilogAndException();

begin
    write "this line is displayed";
    write 42/0;    // Raises a divide-by-zero exception
    write "Exception prevents getting to this line";
epilog
    write "epilog instructions are ALWAYS executed";
end;
```

2. The Unhandled Exception dialog is displayed, because one of the instructions cannot be executed.



3. Click the **Abort** button. If the **Clear Display** command from the Jade Interpreter Output Viewer window was not selected, another two lines are written to the Jade Interpreter Output Viewer window, as follows.



In this method:

- The exception instruction occurs before all of the instructions are executed.
- When you click the **Abort** button, the instruction in the **epilog** section is executed before the method is removed from the stack.

Exercise 3.5 - *foreach* Instruction

In this exercise, you will use a **foreach** instruction loop to output your name ten times.

1. Create and execute a **loopWithForeach** JadeScript method, as follows.

```
loopWithForeach();

vars
    name: String;
    i:    Integer;
begin
    read name;
    foreach i in 1 to 10 do
        write i.String & " " & name;
    endforeach;
end;
```

In this method:

- A counter variable with the name **i** of type **Integer** is declared in the **vars** section.
- The **foreach** instruction repeats the instructions between **foreach** and **endforeach** ten times.
- The Integer variable must be cast as a string with the syntax **i.String** before it can be concatenated with a string.

Note *Type casting* is the process of changing a variable from one type to another.

Exercise 3.6 - *while* Instruction

In this exercise, you will use a **while** instruction loop to output your name ten times.

1. Create and execute a **loopWithWhile** JadeScript method, as follows.

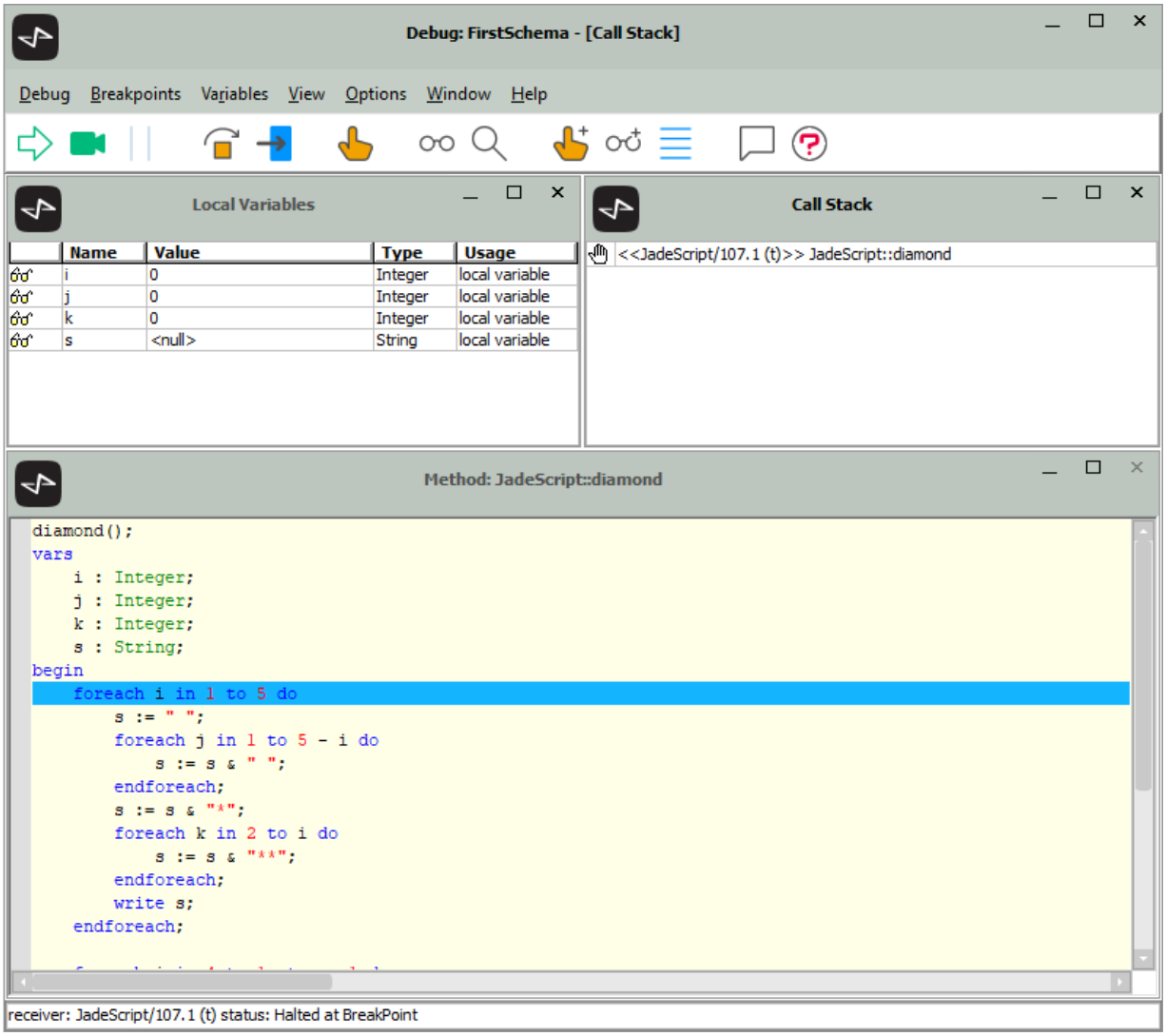
```
loopWithWhile();

vars
    name : String;
    i : Integer;

begin
    read name;

    while i < 10 do
        i += 1;
        write i.String & " " & name;
    endwhile;

end;
```

In this method:

- A counter variable of type **Integer** is declared in the **vars** section.
- While the condition is true, the **while** instruction repeats the instructions between **while** and **endwhile**.

Debugging a JadeScript Method

You can run a JadeScript method through the debugger by selecting the Jade menu **Debug** command or by pressing Shift+F9.



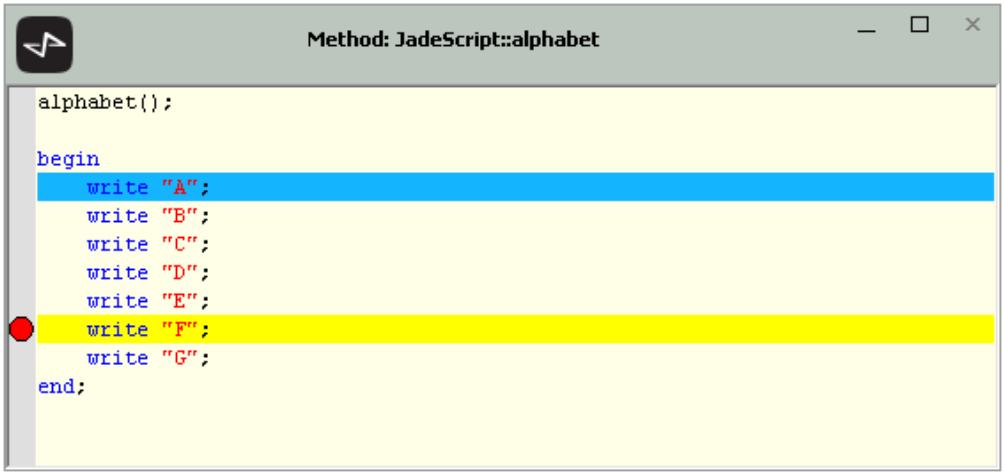
The debugger shows the method code with the next line of code to be executed highlighted with a blue background.

Hover the mouse over a toolbar icon to identify the functionality of that icon (for example, to continue without stopping or to step over or step into the next statement).

You can execute the code one instruction at a time, by clicking the **Step into next statement** and **Step over next statement** buttons in the toolbar. The difference between the two is that if the blue-highlighted statement calls another method, **Step over next statement** executes the called method without debugging, whereas **Step into next statement** debugs the called method.

When you click the **Continue execution** button in the toolbar, the debugger does not step through the code; it executes instructions until it encounters a breakpoint instruction, stopping after executing the instruction immediately before the breakpoint.

You can set a breakpoint in the editor or debugger by pressing the F5 key when your cursor is in a line or by left-clicking on an empty breakpoint margin for a line. The line is then highlighted with a yellow background, to indicate that it is a breakpoint. In addition, a red circle is displayed in the breakpoint margin at the left of the method source window.



For details about the debugger, see "Using the Jade Debugger", in Chapter 7 of the *Development Environment User's Guide* (for example, at https://secure.jadeworld.com/developer-centre/Jade2025/OnlineDocumentation/).

Exercise 3.7 - Jade Debugger

In this exercise, you will write a JadeScript method and then debug it to see how it works.

1. Create and debug a **diamond** JadeScript method, as follows.

```
diamond();
vars
    i : Integer;
    j : Integer;
    k : Integer;
    s : String;
begin
    foreach i in 1 to 5 do
        s := " ";
        foreach j in 1 to 5 - i do
            s := s & " ";
        endforeach;
            s := s & "*";
        foreach k in 2 to i do
            s := s & "**";
        endforeach;
        write s;
    endforeach;
    foreach i in 4 to 1 step - 1 do
        s := " ";
        foreach j in 1 to 5 - i do
            s := s & " ";
        endforeach;
            s := s & "*";
        foreach k in 2 to i do
            s := s & "**";
        endforeach;
        write s;
    endforeach;
end;
```

2. Set a breakpoint on the following line in the JadeScript method (for example, by pressing F5 or Ctrl+Alt+B when the caret is positioned on that line, or by clicking on the breakpoint margin for that line).

```
    foreach i in 4 to 1 step - 1 do
```

   The selected line of code is then highlighted in yellow (or the selected color of your choice).

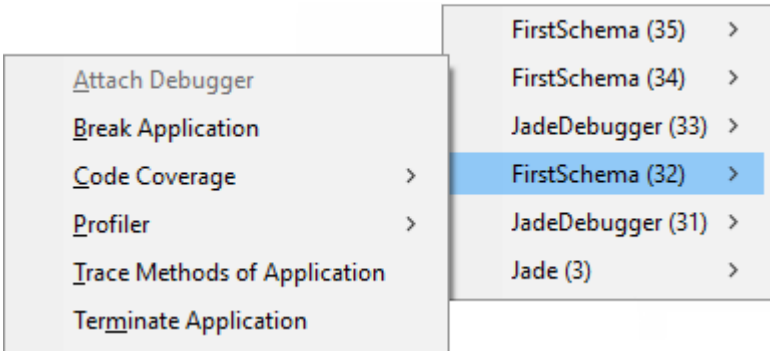
3. Select the Jade menu **Debug** command or press Shift+F9.

4. Execute the code one instruction at a time, by clicking the **Step into next statement** and **Step over next statement** buttons in the toolbar.

5. If you want to execute instructions until a breakpoint instruction is encountered and stop after executing the instruction immediately before the breakpoint, click the **Continue execution** button in the toolbar so that the debugger does not step through the code.

Using the Jade User Interrupt

When you run a user application or a JadeScript method, the Jade User Interrupt icon is displayed in the system tray.



Note For the user interrupt to be displayed, the database must *not* be opened in production mode and the **ShowUserInterrupt** parameter in the [Jade] section of the Jade initialization file must be set to **true**.

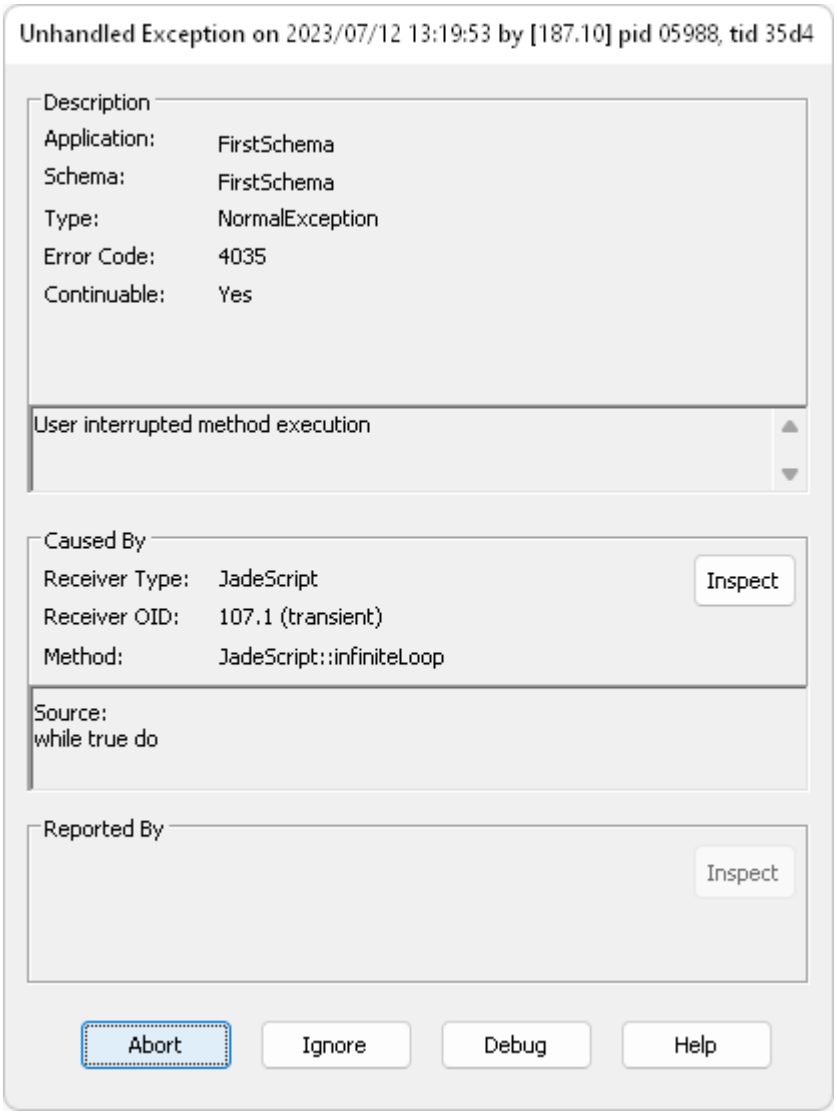
The command options that are available are as follows.

- **Attach Debugger**, which dynamically attaches the Jade debugger when the next method starts
- **Break Application**, which interrupts a running application and displays an exception dialog
- **Code Coverage**, which determines the degree to which the code in methods is executed
- **Profiler**, which records actual and total times spent in methods
- **Trace Methods of Application**, which outputs the method entry and method exit to the interpreter output viewer
- **Terminate Application**, which terminates an application
- **Show an invisible form**, which enables you to terminate an application that has no visible forms

If your code is caught in an infinite loop, the **Terminate Application** message is not received. However, you can use the **Break Application** command.

Tips An alternative way to terminate an infinite loop is to use the **Force Off User** command in the JADE Monitor program.

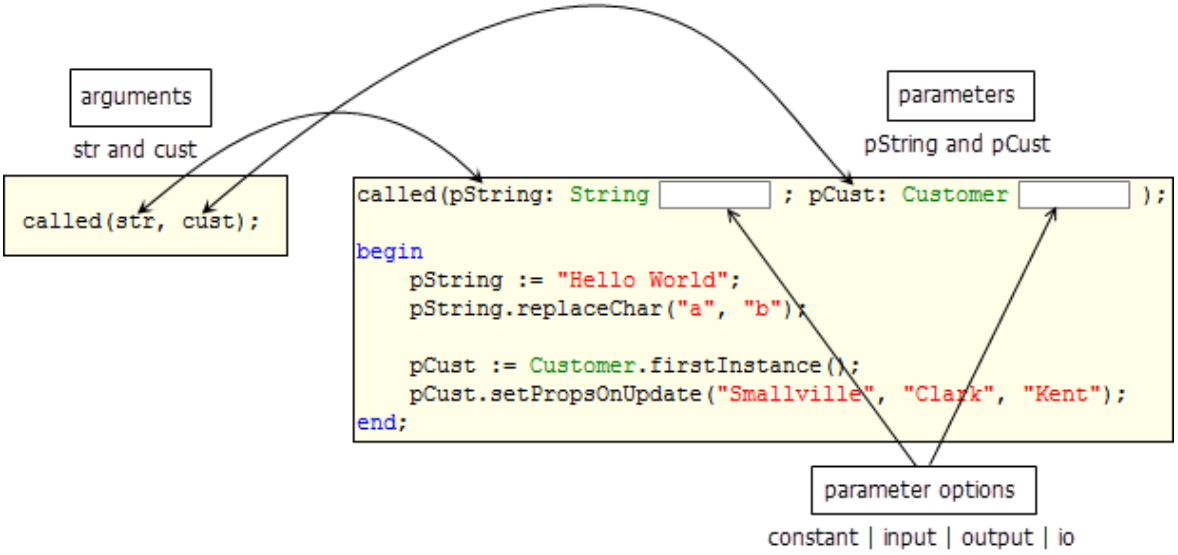
When you use the **Break Application** command, an exception dialog is displayed, enabling you to abort the action.



Parameter Usage Options

Compared with the preceding material in this module, this section is relatively advanced. You may need to return to it at a later stage.

The following diagram shows the **called** method being invoked with arguments **str** and **cust**.



The **called** method is defined with parameters **pString** and **pCust**. Each of these parameters could be followed by a **constant**, **input**, **io**, or **output** method usage option, which affects:

- How the parameter is initialized
- Whether the parameter can be assigned a new value
- Whether the parameter can be updated

If a parameter is assigned a new value or updated, the change is reflected in the argument when the method returns.

The following subsections describe what happens for each method parameter usage option.

constant

constant is the default parameter usage option. If nothing is specified, **constant** is assumed.

The value of a **constant** usage parameter cannot be changed by direct assignment or by calling an **updating** method.

The following method shows the restrictions that apply to **constant** parameters.

```
called(pString: String constant; pCust: Customer constant);

begin
    pString := "Hello World";                          //  NOT allowed
    pString.replaceChar("a", "b");                     //  NOT allowed
    pCust := Customer.firstInstance();                 //  NOT allowed
    pCust.address := "Smallville";                     //  NOT allowed
end;
```

input

For primitive parameters, a usage of **input** is similar to **constant** in that the value cannot be changed by assignment. However, it can be changed by calling an **updating** method.

For object parameters, a usage of **input** specifies that the object the parameter references cannot be changed. However, properties of the object can be updated.

The following method shows the restrictions that apply to **input** usage parameters.

```
called(pString: String input; pCust: Customer input);

begin
    pString := "Hello World";                          //  NOT allowed
    pString.replaceChar("a", "b");                     //  Allowed
    pCust := Customer.firstInstance();                 //  NOT allowed
    pCust.address := "Smallville";                     //  Allowed
end;
```

output

An **output** usage parameter is used to pass a value from the method being called back to the calling method.

Tip **output** parameters are useful when you need to return more than one value from a method.

The value of an **output** usage parameter is initialized to the appropriate **null** value at the start of the method being called; for example, zero (**0**) for an **Integer**, **""** for a **String**, and a **null** reference for an object parameter. Effectively, this means that values are not passed in.

When the method returns, the values of **output** usage parameters are copied back into the caller's arguments.

```
called(pString: String output; pCust: Customer output);

begin
    pString := "Hello World";                          //  Allowed
    pString.replaceChar("a", "b");                     //  Allowed
    pCust := Customer.firstInstance();                 //  Allowed
    pCust.address := "Smallville";                     //  Allowed
end;
```

io

An **io** usage parameter is used to pass a value into the **called** method; that is, parameters are initialized from arguments and are not set to **null** values.

In effect, **io** usage parameters enable arguments to be passed in, updated, and passed back.

Exercise 3.8 - *break* and *continue* Instructions

In this exercise, you will use an **if** instruction inside a loop to control the iteration. Without the **if** instruction, the loop would print your name ten times.

However, the third printing of your name is skipped and the loop is exited before printing your name for the eighth time.

1. Create and execute a **breakAndContinue** JadeScript method through the debugger and step through each instruction.

```
breakAndContinue();

vars
    name : String;
    i : Integer;

begin
    read name;

    while i < 10 do
        i += 1;
        if i = 3 then
            continue;
        elseif i = 8 then
            break;
        endif;
        write i.String & " " & name;
    endwhile;
end;
```

In this method:

- The loop contains an **if** instruction.
- The **continue** instruction skips to the next iteration of a **foreach** or **while** loop.
- The **break** instruction exits from a **foreach** or **while** loop.

Exercise 3.9 - Jade User Interrupt

In this exercise, you will deliberately code an infinite loop.

1. Create and execute an **infiniteLoop** JadeScript method, as follows.

```
infiniteLoop();

begin
    while true do
    endwhile;
end;
```

2. Use the Jade User Interrupt to break out of the infinite loop.

Exercise 3.10 - Parameters and Return Type

In this exercise, you will add one JadeScript method that can call another JadeScript method, passing values as parameters.

1. Add a JadeScript method called **constructMessage**, which is passed a **String** and an **Integer** parameter.

   The parameters are used to construct a long string and then return this value to a calling method.

```
constructMessage(phrase: String; count: Integer): String;

vars
    str: String;
    i:   Integer;
begin
    foreach i in 1 to count do
        str := str & phrase;
    endforeach;
    return str;
end;
```

2. What happens when you attempt to execute this JadeScript method?

   Note A method with parameters must be called from another method so that values for the parameters can be provided.

3. Add a JadeScript method called **start**, which calls the **constructMessage** method.

```
start();

vars
    str: String;
    i:   Integer;
begin
    read str;
    read i;
    write self.constructMessage(str, i);
end;
```

4. Execute the **start** method through the debugger.

   Note The **constructMessage** method cannot be executed directly, because it has parameters. Execute the **start** method, which calls the **constructMessage** method.

5. Use the **Step into next statement** toolbar button to step through all of the instructions.

In this method:

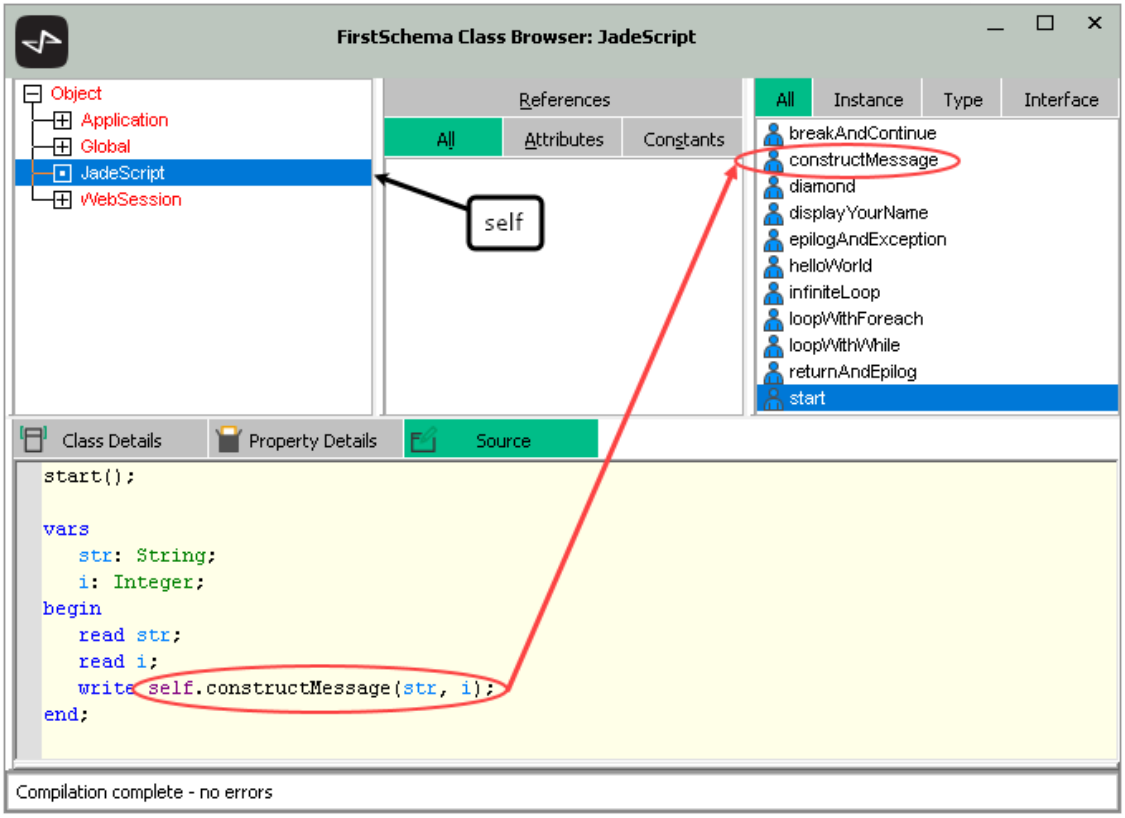
- The assignment operator (**:=**) is used.
- The variable **self** refers to the receiver; that is, the object for which the method is executing, which is a **JadeScript** object.

  Note You can omit the **self.** syntax; for example, **constructMessage(str, i)** is equivalent to **self.constructMessage(str, i)**.

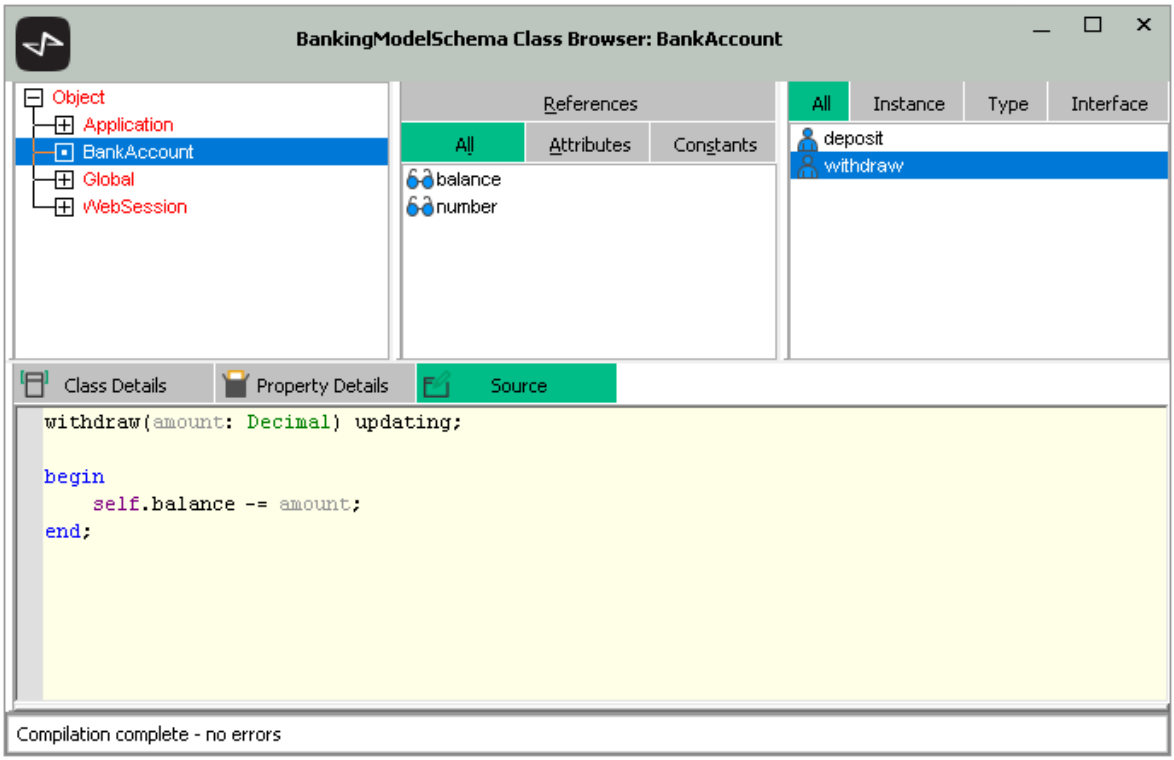
  However, we recommend that you include the **self.** system variable, to avoid any ambiguity.

self Object

In the previous exercise, the **start** JadeScript method called the **constructMessage** JadeScript method (that is, a method in the same class), by sending a message to the **self** object.



In the following example, the **withdraw** method in the **BankAccount** class refers to its **balance** property as **self.balance**.



You can omit **self** from the syntax, as follows, but again we recommend always using **self.** to avoid any ambiguity.

```
withdraw(amount: Decimal) updating;

begin
    balance -= amount;
end;
```

Exercise 3.11 - Parameter Usage Options

In this exercise, you will add a JadeScript method called **threeHellos**, which calls another JadeScript method called **threeWorlds**.

Three strings with a value of **"Hello"** are passed to **threeWorlds**, which attempts to concatenate **" World"**. The value of the resulting string depends on the whether the method parameter usage is **input**, **output**, or **io**.

1. Add a JadeScript method called **threeWorlds**, which is passed three **String** parameters.

   The first parameter has the **input** usage, the second has the **output** usage, and the third has the **io** usage. Instructions attempt to add the string **" World"** to each parameter.

```
threeWorlds(inputStr: String input; outputStr: String output; ioStr: String io);

begin
    // inputStr &= " World"; // Not allowed for constant or input
    outputStr &= " World";
    ioStr &= " World";
end;
```

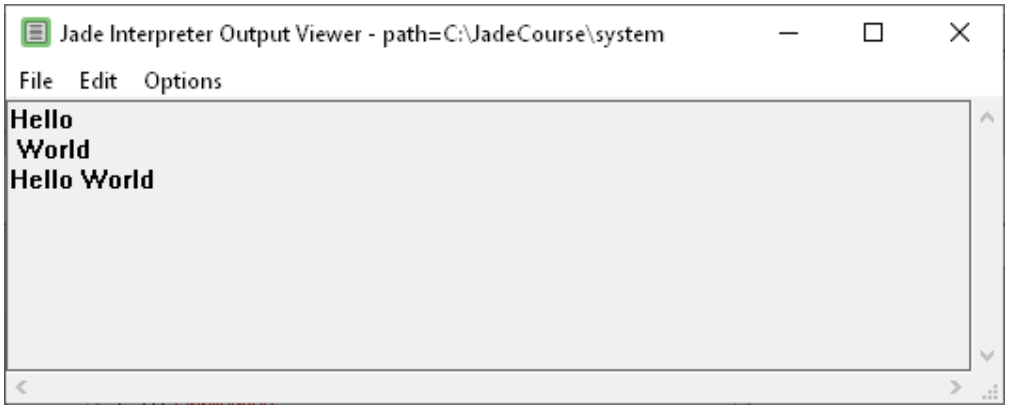
2. Add a JadeScript method called **threeHellos** that calls **threeWorlds**.

```
threeHellos();

vars
    str1, str2, str3: String;
begin
    str1 := "Hello";
    str2 := "Hello";
    str3 := "Hello";
    self.threeWorlds(str1, str2, str3);
    write str1;
    write str2;
    write str3;
end;
```

3. Execute **threeHellos** through the debugger.

   Use the **Step into next statement** toolbar button to step through all of the instructions. Observe how the string values change.

4. Three lines are written to the Jade Interpreter Output Viewer window, as follows.



In this method:

- The **input** parameter **"Hello"** in the **threeWorlds** method cannot be changed.
- The **output** parameter **"Hello"** in the **threeWorlds** method is initialized to a null value before it is concatenated with **" World"**.
- The **io** parameter **"Hello"** in the **threeWorlds** method is concatenated with **" World"**.

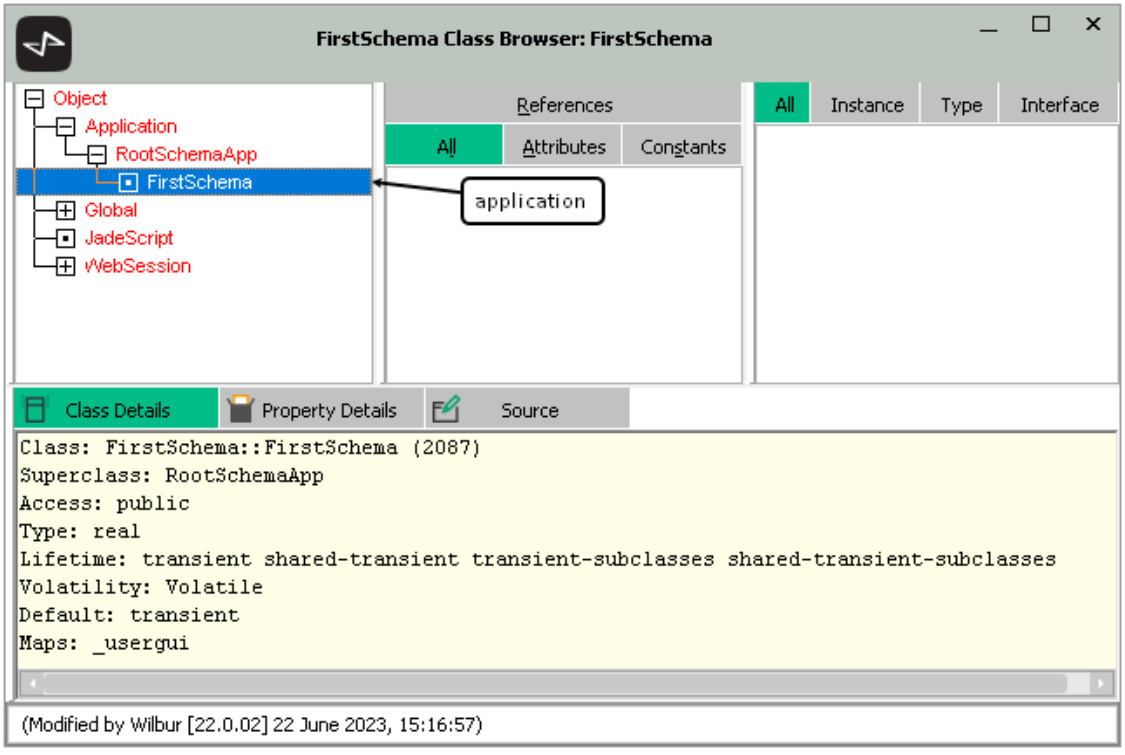
Module 4 Application Object

This module contains the following topics.

Introduction

When you run a JadeScript method or an application, a transient instance of your **Application** subclass is created. The object, like all transient objects, is automatically deleted when the JadeScript method or application finishes. This object inherits a lot of useful functionality from the **Application** class.

You can refer to this transient **Application** object in your code by using the **app** system variable.



The following JadeScript method demonstrates some useful methods provided by the **app** object.

```
appMethods();

    // Copy some text to the clipboard before pressing F9
begin
    app.clearWriteWindow();
    write app.copyStringFromClipboard();
    app.msgBox("Do you want to continue?", "Question", MsgBox_Yes_No);
    write "The method will attend to other events for 10 seconds";
    app.doWindowEvents(10000);
    // Other useful methods
    write app.clock();
    write app.dbPath();
    write app.random(100);
    write app.userName();
end;
```

Context-Sensitive Help

Context-sensitive help is available in the editor pane for Jade instructions and for **RootSchema** types, properties, and methods, as well as for menus, menu items (commands), and dialogs in the development environment.

With the provision of the full product information library in both HTML5 (web) and PDF (print) format, by default, context-sensitive help is obtained from .**htm** topics in the HTML5 web format of the product information.

Context-sensitive help to HTML5 topics is controlled by the **UseJadeWebHelp** parameter in the [JadeHelp] section of the Jade initialization file. This parameter is **true** by default, in which case it reads the **JadeHelpBaseUrl** parameter in that section. If a value is specified for the **JadeHelpBaseUrl** parameter, it uses that URL. If the value is **<default>** or it is empty, the URL is determined by the internal hard-coded URL for the current release. For example, the [JadeHelp] section of the Jade initialization file could contain the following parameter values.

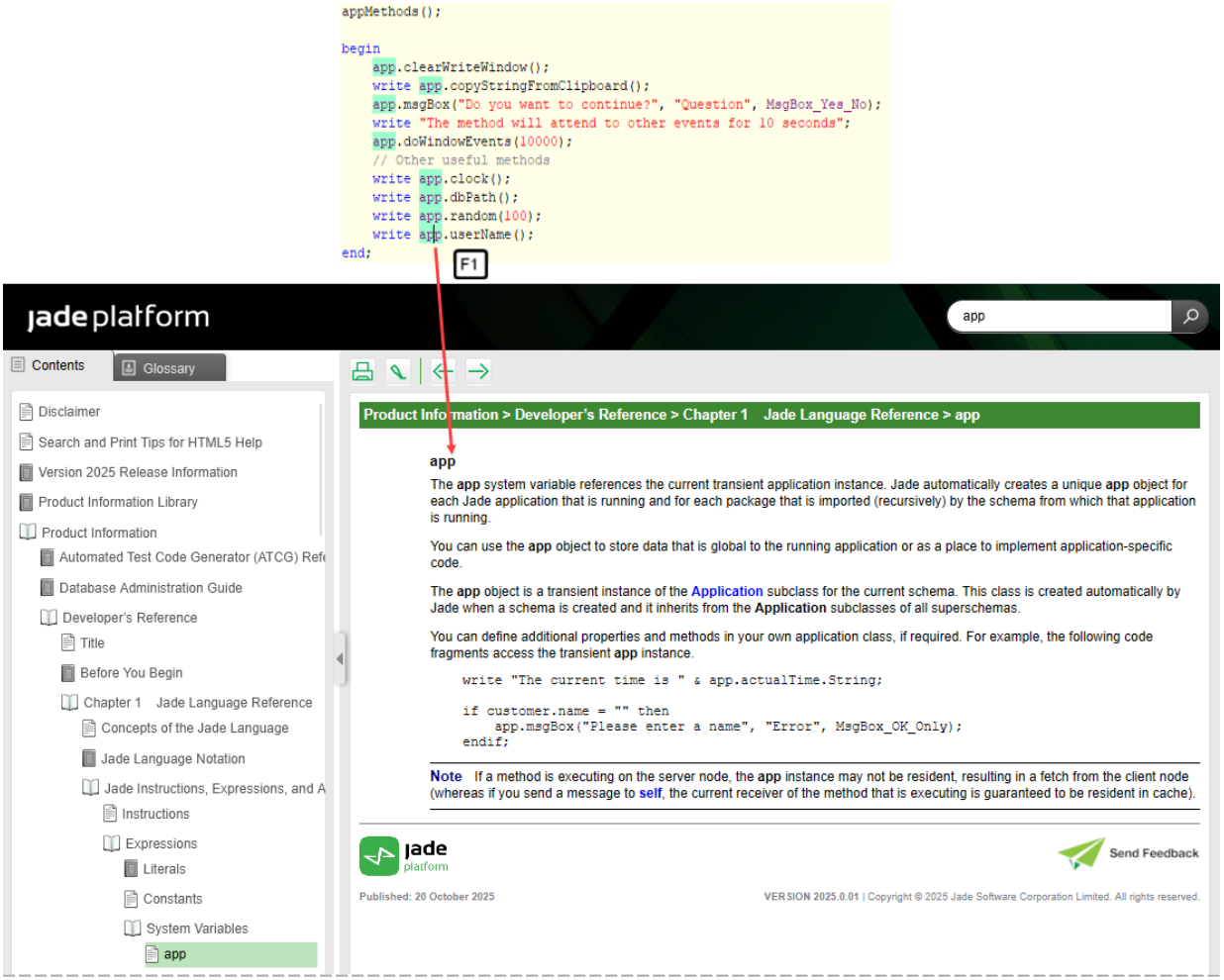
```
[JadeHelp]
UseJadeWebHelp=true
JadeHelpBaseUrl=https://secure.jadeworld.com/developer-centre/Jade2025/OnlineDocumentation/Default.htm
htmlSchemes=<default>
```

Set the value of the **UseJadeWebHelp** to **false** if you want to use context-sensitive help to specific sections in the appropriate PDF files (for example, if you have slow or restricted web access, or if you want to print a range of pages or all of a document).

To access context-sensitive help, perform the following actions.

1. Position the cursor inside the word (for example, **app**).
2. Press F1.

The web help for that topic or the relevant section of a Portable Document Format (PDF) file in Adobe Reader is then displayed, as shown in the following image of the topic in a web browser.



Tip The second item in the navigation pane at the left of any HTML5 page is the "Search and Print Tips for HTML5 Help" topic, which provides information about formulating search expressions and printing a topic.

Exercise 4.1 - Context-Sensitive Help and the *app* Object

In this exercise, you will demonstrate and learn about the functionality of the **app** object, by using context-sensitive help.

1. Add a JadeScript method called **appMethods** and code it as follows.

```
appMethods();


    // Copy some text to the clipboard before pressing F9
begin
    app.clearWriteWindow();
    write app.copyStringFromClipboard();
    app.msgBox("Do you want to continue?", "Question", MsgBox_Yes_No);
    write "The method will attend to other events for 10 seconds";
    app.doWindowEvents(10000);
    // Other useful methods
    write app.clock();
    write app.dbPath();
    write app.random(100);
    write app.userName();
end;
```

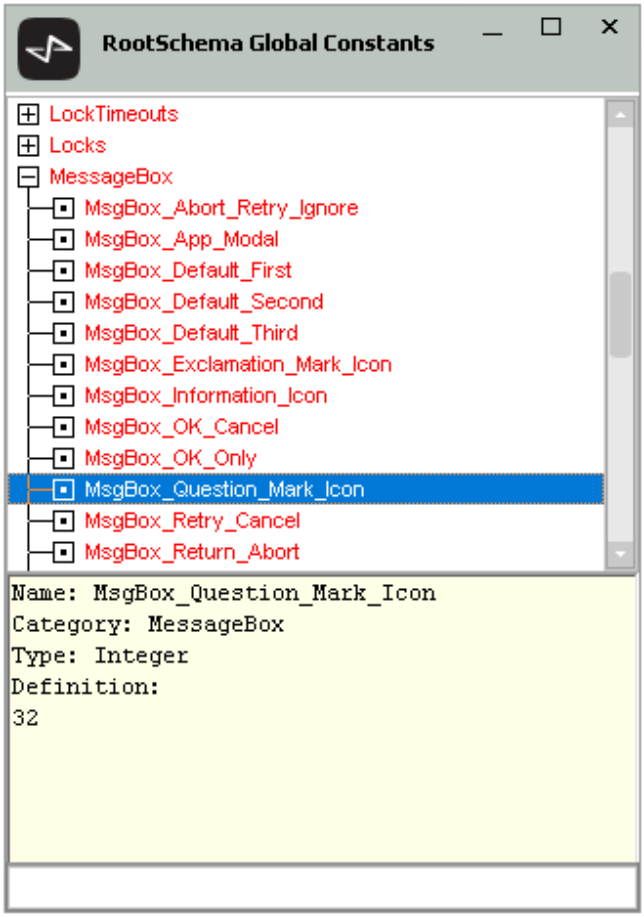
2. Compile the method.
3. Copy some text to the clipboard from any application; for example, Word, Notepad, or a web browser.
4. Execute the method.
5. Position the cursor inside the word **app**, and then press F1 to open context-sensitive help.
6. Position the cursor inside the word **write**, and then press F1.
7. Obtain context-sensitive help for the following entities in the **appMethods** JadeScript method.
   - clearWriteWindow
   - clock
   - copyStringFromClipboard
   - dbPath
   - doWindowEvents
   - msgBox
   - random
   - userName

In this **appMethods** method, single-line comments begin with two forward slash characters (**//**). Multiple-line comments are enclosed between **/*** and ***/** characters.

Global Constants

Global constants are primitive values that can be accessed by any class or method in the current schema and subschemas. Constants are grouped into categories.

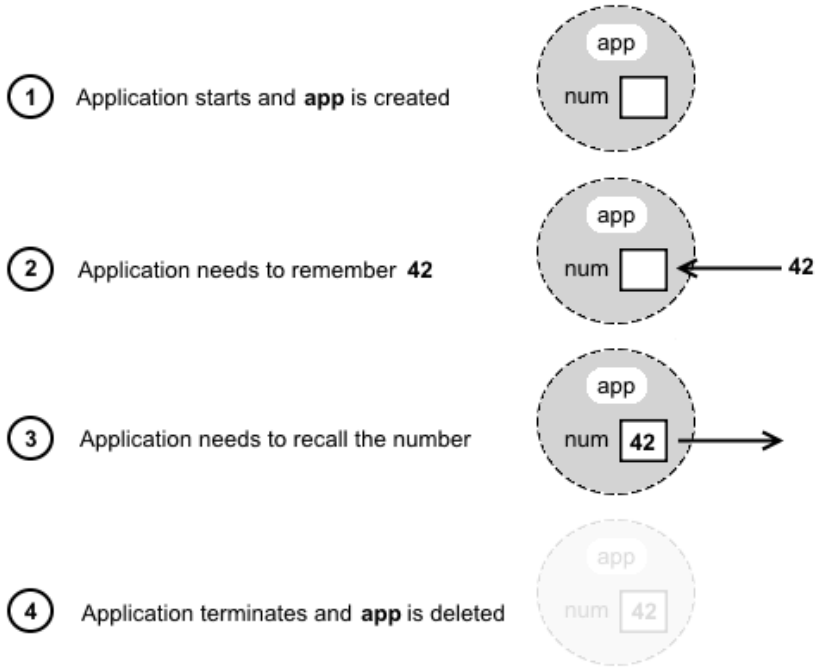
Access the list of categories and the global constants they contain, by using the Browse menu **Global Constants** command. The following image shows the global constants and categories in **RootSchema**.



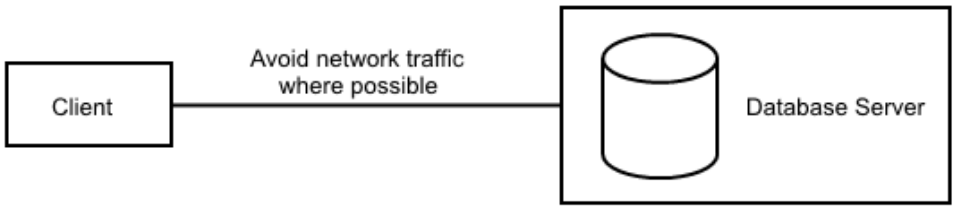
Another Use of the Application Object

You can use the **app** object to *remember* important information for the duration of the application. This is extremely useful for an application but not at all important for a JadeScript.

The following diagram shows the steps required for an application to store a number, and subsequently to recall that number later in the session.



The number could have been stored in and retrieved from a persistent database object. However, that would require communication across the network between the client application and the database server. The **app** object is a transient object, which is accessed more quickly from memory.

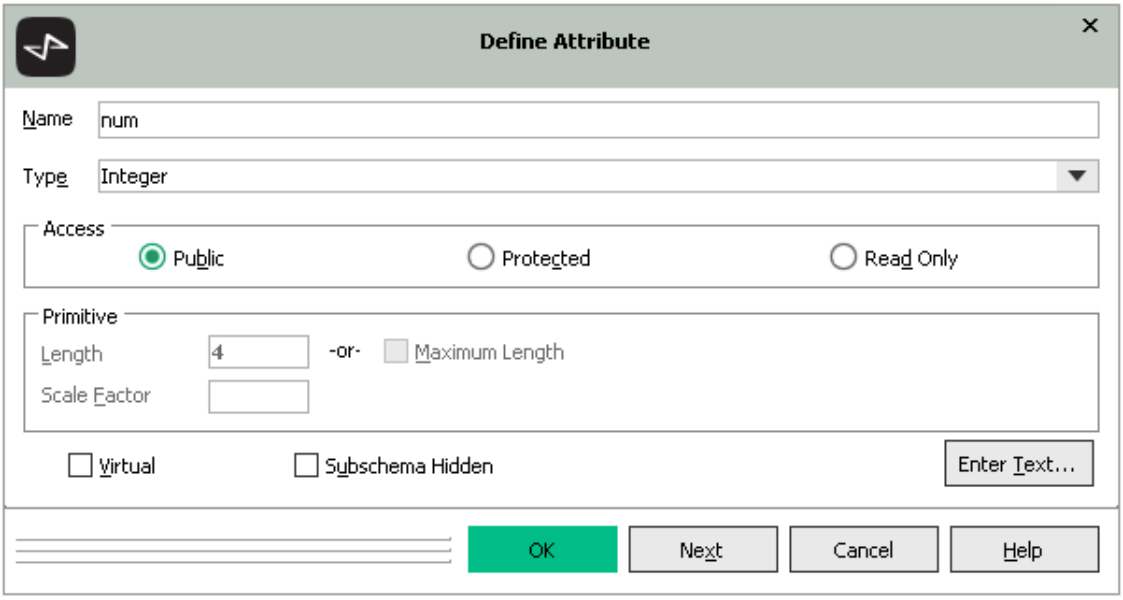


Exercise 4.2 - Adding an Attribute

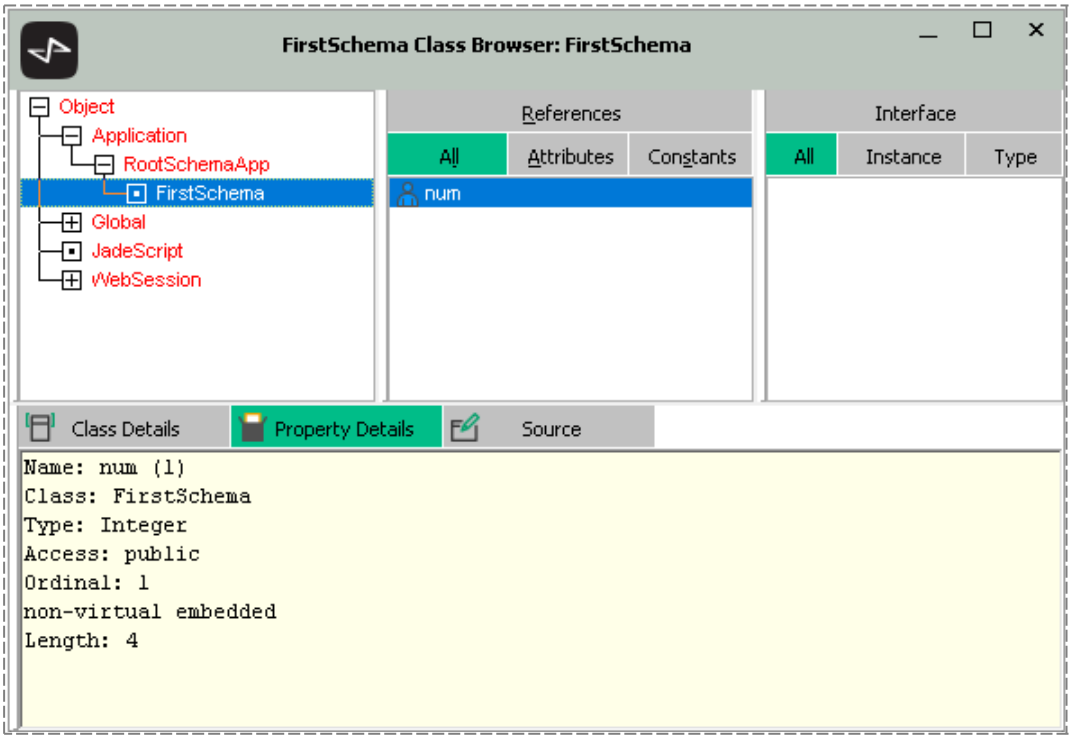
In this exercise, you will add a **num** attribute to your **Application** subclass.

1. Select your **Application** subclass in the Class Browser.
2. Add an attribute, by selecting the Properties menu **Add Attribute** command.

3. Enter **num** as the name of the attribute, select the **Integer** type, and then select the **Public** access option.



4. Click the **OK** button. The **num** property is then displayed in the Properties List of the Class Browser.



Exercise 4.3 - Using *app* to Store a Value

In this exercise, you will use the **num** attribute that you created in the previous exercise.

1. Add a JadeScript method called **remembering**, coded as follows.

```
remembering();

begin
    // Storing a value in app
    app.num := 42;
    // Recalling that value
    write app.num;
end;
```

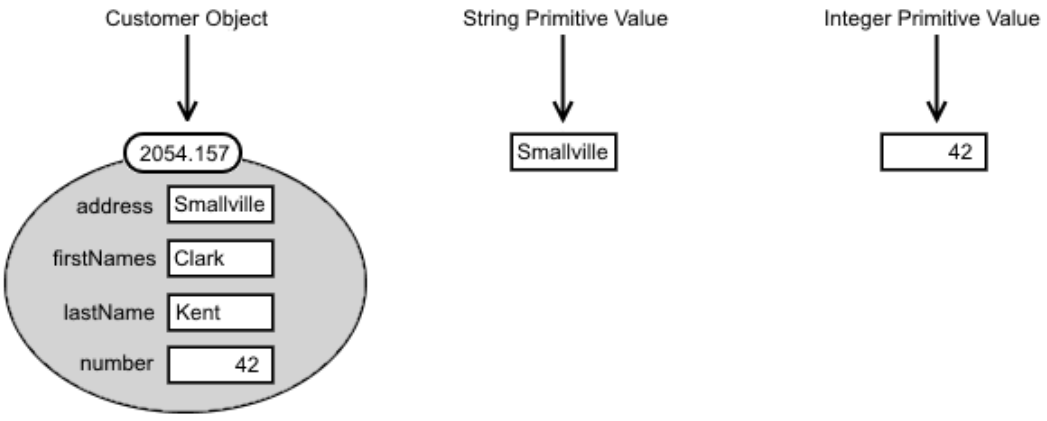
2. Execute the JadeScript method.

Module 5 Primitive Types

This module contains the following topics.

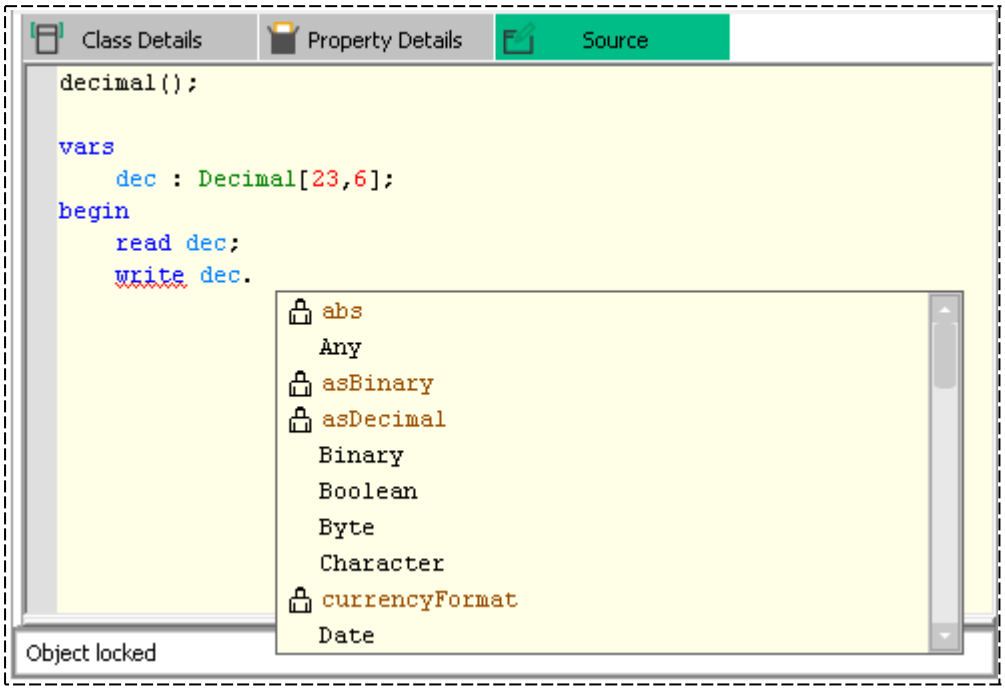
Introduction

Dates, times, strings, and so on, are values of a primitive type rather than instances of a class.



As primitive types are simply values, they do not have properties but they *do* have methods, which are defined in **RootSchema**. You can extend this functionality by adding methods to the primitive types in your schema.

The **AutoComplete** functionality in the editor pane displays methods that can be called for a primitive type.



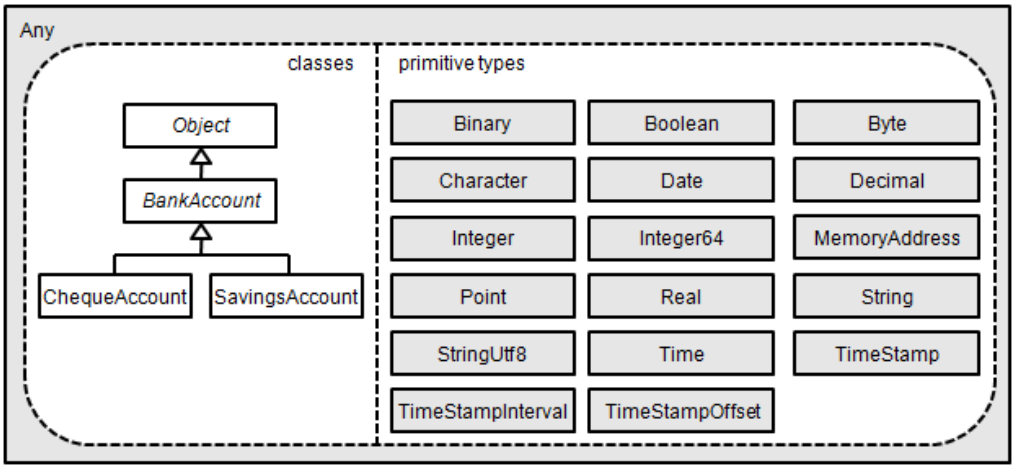
Primitive Types

Simple values such as dates, times, and strings are handled using primitive types rather than objects. A variable or attribute that is a primitive type contains a value as opposed to a reference to an object.

A primitive type, unlike a class type:

- Does not have properties
- Cannot have subtypes

The following diagram shows the available types.



A variable of type **Any** can represent an object or a primitive value, and provides the **isKindOf** method for type checking.

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

Working with Numbers

The numeric primitive types are:

- **Byte**, which is an unsigned integer value in the range zero (**0**) through **255**.
- **Decimal**, which is a number with specified length and number of decimal places.

  The **Decimal** type is the usual choice for currency values. For a **Decimal**, you must specify the number of digits (precision) and the number of decimal places (scale factor).

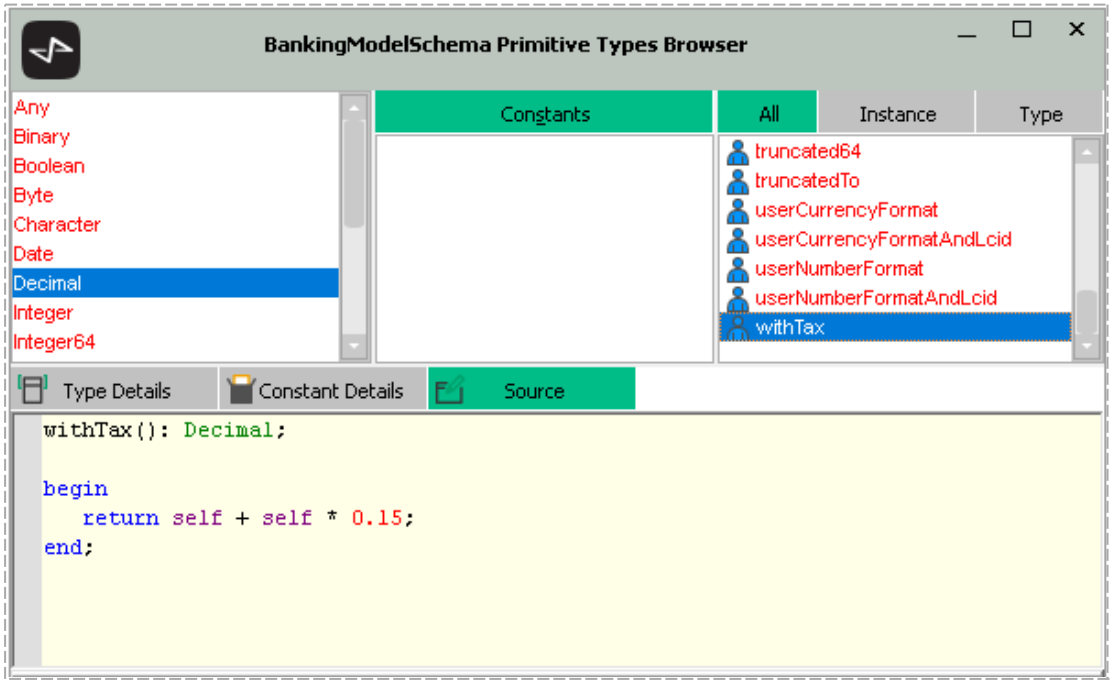
```
vars
    dec: Decimal[6, 2];     // 6 digits altogether
                            // 2 are after the decimal point (so 4 are in front)
                            // Maximum value would be 9999.99
```

- **Integer**, which is a signed 32-bit whole number.
- **Integer64**, which is a signed 64-bit whole number.
- **Real**, which is a floating-point number.

A numeric local variable is initialized to zero (**0**).

Adding Primitive Type Methods

You can add methods to the primitive types to augment the class type methods supplied in **RootSchema**. As an example, when working with prices, the price with tax included is often required. You could add a **withTax** method, as shown in the following image.



To open a Primitive Types Browser, click the **P** button from the Jade Platform development environment toolbar.

When you select the **Decimal** type in the left-hand window (that is, the Primitive Type List), you can display the methods provided by **RootSchema** by selecting the View menu **Superschemas** command. You can add your own method in the same way you previously added JadeScript methods, by selecting the Methods menu **New Jade Method** command.

In a primitive type method, the **self** variable refers to the primitive value for which the method is being run; for example, in the **withTax** method, **self** is the original price to which tax is being added.

The following methods are examples of ways to code a **withTax** method. In the first implementation, **self** (the original price) is not changed. A new decimal value is returned.

```
withTax(): Decimal;

begin
    return self + self * 0.15;
end;
```

In the next implementation, which has the **updating** option in the signature, the value of **self** is changed, and then the new value returned.

```
withTax(): updating;

begin
    self := self + self * 0.15;
end;
```

In the second implementation, when you produce the price with tax, you effectively lose the original price.

Working with Strings

The string primitive types are:

- **Character**, which is a single ANSI or Unicode character
- **String**, which is a sequence of characters
- **StringUtf8**, which is a string encoded in UTF8 format

A **String** or **StringUtf8** local variable is initialized to an empty string (**""**).

A **Character** local variable is initialized to the null character (hexadecimal **00**).

Substring Operator

You can parse a string using a square bracket substring operator, as shown in the following example.

```
vars
    str: String;
begin
    str := "Hello world";
    write str[7];      // "w"        - single character at specified position
    write str[4:5];    // "lo wo"    - substring with specified start and length
    write str[4:end];  // "lo world" - substring from specified start to end
end;
```

Note The first character in a string is at position 1.

pos Method

The **pos** method searches for a specified substring, starting the search from a specified position. It returns the character position where the substring starts, or zero (**0**) if the substring is not found, as shown in the following examples.

```
write "indefinite article".pos("abc", 1); // Outputs 0 - "abc" is not a substring
write "indefinite article".pos("def", 1); // Outputs 3 - "def" is at position 3
write "indefinite article".pos("def", 5); // Outputs 0 - "def" not found beyond 5
```

The **pos** method is often used to test for a substring, as follows.

```
if str1.pos(str2, 1) > 0 then
    // str2 is a substring of str1
else
    // str2 is not a substring
endif;
```

trimBlanks Method

The **trimBlanks** method removes spaces from the start and the end of a string.

```
write "  surrounded by spaces   ".trimBlanks(); // Outputs "surrounded by spaces"
```

It is often used to *clean* data before it is stored in the database.

Working with Dates and Times

The date and time primitive types are:

- **Date**, which is the number of days since the start of the Julian period (24 November -4713)
- **Time**, which is the number of milliseconds since midnight
- **TimeStamp**, which is the combined date and time value
- **TimeStampInterval**, which is the difference between two timestamps
- **TimeStampOffset**, which is the UTC date and time value with a local offset

A **Date** local variable is initialized with today's date. As a **Date** variable is essentially a 32-bit integer, you can use simple arithmetic when working with dates, as shown in the following example.

```
vars
    date: Date;
begin
    write date;                 // Outputs today's date
    write date + 7;             // Outputs the date next week
end;
```

Type Casting

You can convert a value from one primitive type to another by type casting (if such a conversion makes sense). To cast an expression, append a period and the destination type, as shown in the following examples.

```
write 65.Character;          // Outputs "A"
write 65.Date;               // Outputs "28 January -4712"
write "65".Integer + 35;     // Outputs 100
write "65ABC".Integer;       // Outputs 65
```

The **write** instruction converts the expression that follows to a string.

Type-casting instructions can fail at compile time or at run time, as shown in the following examples.

```
write 5.TimeStamp;           // Compile error - invalid type cast
write 500.Byte;              // Runtime error - overflow exception
```

Other Primitive Types

The other primitive types are:

- **Binary**, which is binary data (for example, graphics and multimedia)
- **Point**, which is the x (horizontal) and y (vertical) coordinates of a point
- **MemoryAddress**, which is the address of a C **void*** pointer

Exercise 5.1 - Rounding

Write a JadeScript method that:

1. Declares a variable of type **Decimal** with a length of 12 and a scale factor of 4.
2. Uses the **read** instruction to store a number that is entered by the user in the variable.
3. Rounds the number entered to two decimal places. (Hint: use the **roundedTo** method.)
4. Uses the **write** instruction to display the answer.

Exercise 5.2 - Adding a Primitive Type Method

In this exercise, you will use the **read** instruction to enable the user to enter information.

1. Open a Primitive Types Browser for **FirstSchema**.
2. Select the **Decimal** type.
3. Add and code the **withTax** method, which returns a value that is 15 percent greater, rounded to two decimal places.

4. Test the **withTax** method by adding a JadeScript method, as follows.

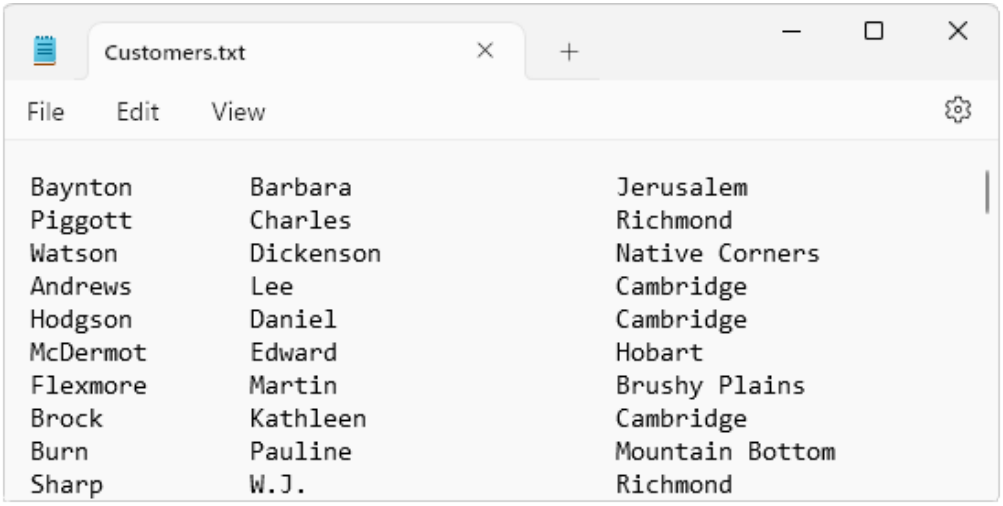
```
testTax();

vars
    dec: Decimal[12,2];
begin
    read dec;
    write dec.withTax();
end;
```

Exercise 5.3 - Substrings

In this exercise, you will work with the first line of text from the **customers.txt** file.

1. Open the **C:\JadeCourse\Files\customers.txt** file with Notepad.

   If you are using a monospaced font (for example, **Courier New**), it will look similar to the following image.



2. Each line of the file contains a person's first name, last name, and address; for example, the first line is **Barbara Baynton** from **Jerusalem**. This file has a fixed-width format; that is, the fields are followed by differing numbers of space characters to maintain the columnar alignment of the data.

   a. At which position in the line does **Barbara** begin?

   b. At which position in the line does **Baynton** begin?

   c. At which position in the line does **Jerusalem** begin?

   d. In this file, what is the maximum possible length of a first name?

   e. What is the maximum possible length of a last name?

   f. What is the maximum possible length of an address?

3. Add a JadeScript method called **parsing** that contains the following code.

```
parsing();

vars
    str, first, last, address: String;
begin
    // Copy of the first line from the customers.txt file
    str := "Baynton        Barbara                  Jerusalem                ";
    // Use the substring operator str[n:m] to complete this method
    first :=   <to be completed>
    last :=    <to be completed>
    address := <to be completed>
    write first & " " & last & " from " & address;
end;
```

Note This method will not compile, because the assignment instructions are incomplete.

Complete the assignment instructions and then execute the method.

Exercise 5.4 - Date Arithmetic

In this exercise, you will determine the number of days until Christmas.

1. Create a **christmas** JadeScript method and code it as follows.

```
christmas();

vars
    today : Date;
    xmas : Date;
    currentYear : Integer;
begin
    /*  Note: As we haven't set a value for "today",
     *  it will default to the current date.
     */
    currentYear := today.year;
    xmas.setDate(25, 12, currentYear);

    write xmas - today;
end;
```

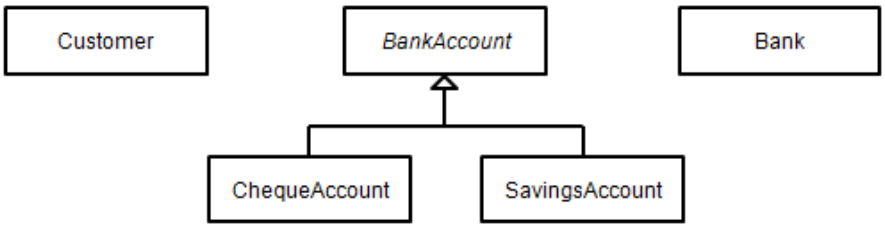
2. Execute the method.

Module 6 Classes

This module contains the following topics.

Introduction

The model for the banking system, which you build during the course, is shown in the following diagram.



Note The name of an abstract class is italicized in a UML class diagram.

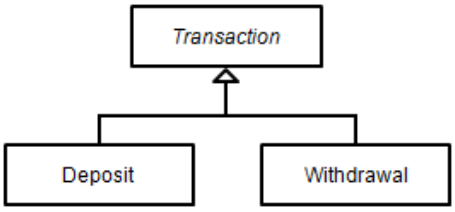
The **Customer** class is the first class that you create.

The **BankAccount** class is the abstract superclass for the hierarchy of bank account classes.

The **BankAccount** contains methods and properties to be inherited by the real subclasses. The **ChequeAccount** and **SavingsAccount** classes are specialized with appropriate additional methods and properties.

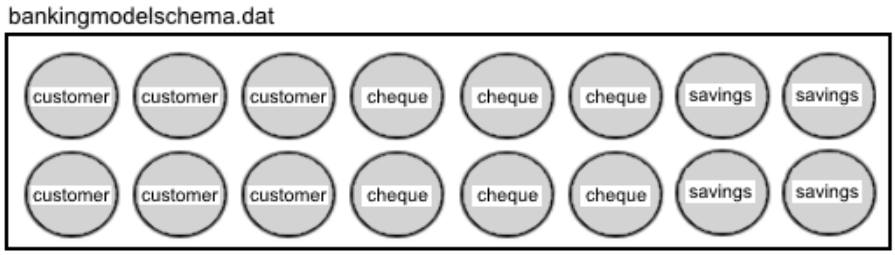
The **Bank** class is the *root object* class for the system. (The purpose of a root object will be explained in a later module.)

For simplicity, classes for depositing and withdrawing money from bank accounts have not been included.

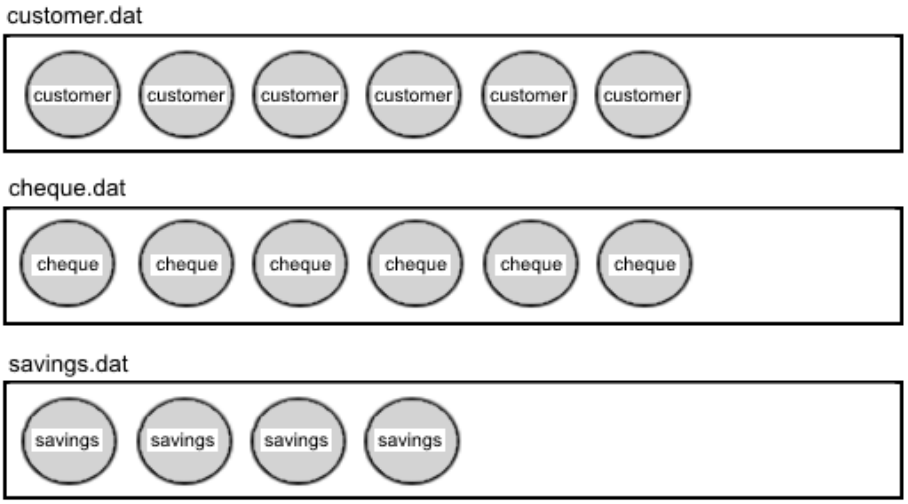


Database Files

The persistent instances of a class are stored in database files, which are files in the **system** directory with a .**dat** extension. Database files are also known as *map* files, referring to the mapping that exists between classes and database files. In the following diagram, the **Customer** class, **ChequeAccount** class, and **SavingsAccount** class are mapped to the **bankingmodelschema.dat** file, the default map file that is created for the schema.



You can create additional database files and map each class to a separate file.

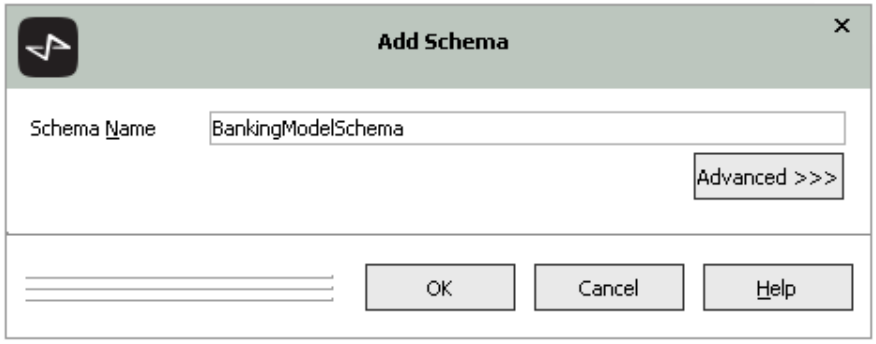


When classes are mapped to separate map files, the impact of a database reorganization can be limited, resulting in saving time because only the affected files need to be reorganized.

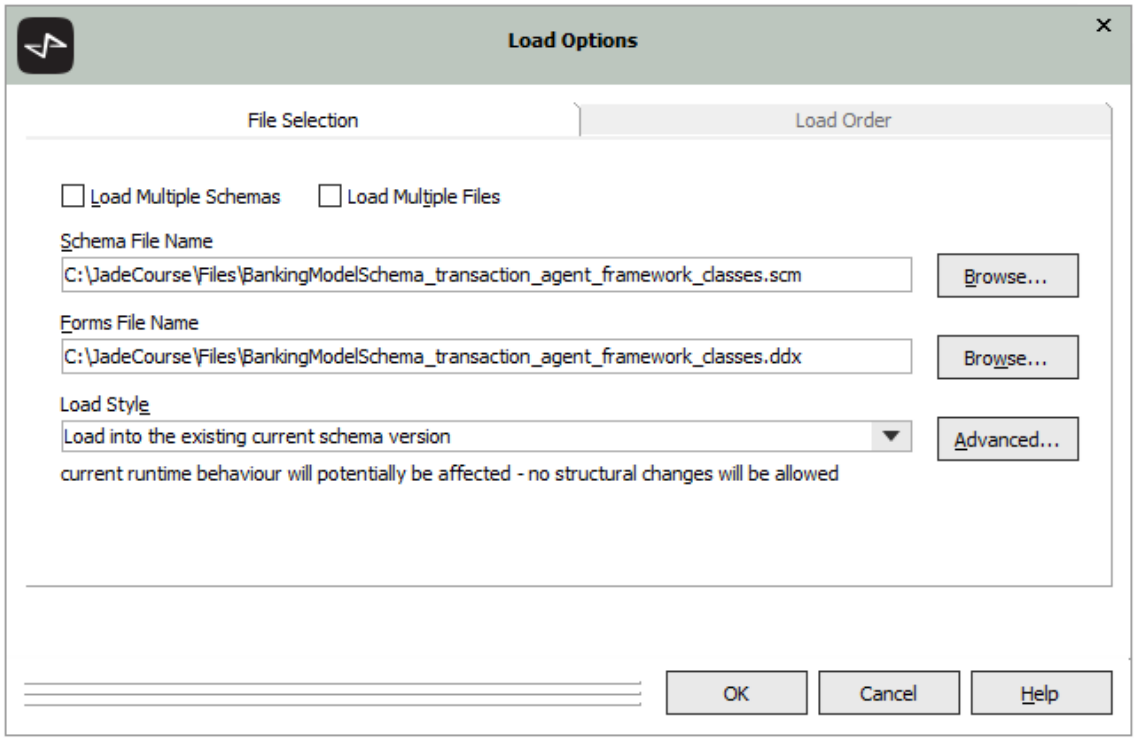
Exercise 6.1 - Adding a Schema

In this exercise, you will add a schema that will contain the database classes for a banking system.

1. Select the Schema Browser by clicking the **S** button from the Jade Platform development environment toolbar.
2. Select **RootSchema** in the Schema Browser.
3. Add a schema by selecting the Schema menu **Add** command.
4. Enter **BankingModelSchema** as the name of the schema and then click the **OK** button.



5. Load a schema into the **BankingModelSchema** by selecting the Schema menu **Load** command, enter the values shown in the following Load Options image, and then click the **OK** button.



Exercise 6.2 - Loading Transaction Agent Framework (TAF) Classes

In this exercise, you will load the Transaction Agent Framework (TAF) into the schema.

1. Select the **BankingModelSchema** in the Schema Browser, if it is not already selected.
2. Select the Schema menu **Load** command.
3. Click the **Browse** button next to the **Schema File Name** text box and then select the **BankingModelSchema_transaction_agent_framework_classes.scm** file included with the course files.
4. Click the **Browse** button next to the **Forms File Name** text box and then select the **BankingModelSchema_transaction_agent_framework_classes.ddx** file included with the course files.
5. For the **Load Style**, select **Load into the existing current schema version** from the drop-down list.
6. Click the **OK** button, to complete the loading of the TAF classes.

Exercise 6.3 - Adding Map Files

In this exercise, you will add map files for the banking system.

1. Select the Maps Browser by clicking the **M** button from the Jade Platform development environment toolbar.
2. Add a map file by selecting the MapFiles menu **Add** command.
3. Enter **customer** as the file name and then click the **OK** button.

Note Do *not* specify the .**dat** extension. It is added automatically.

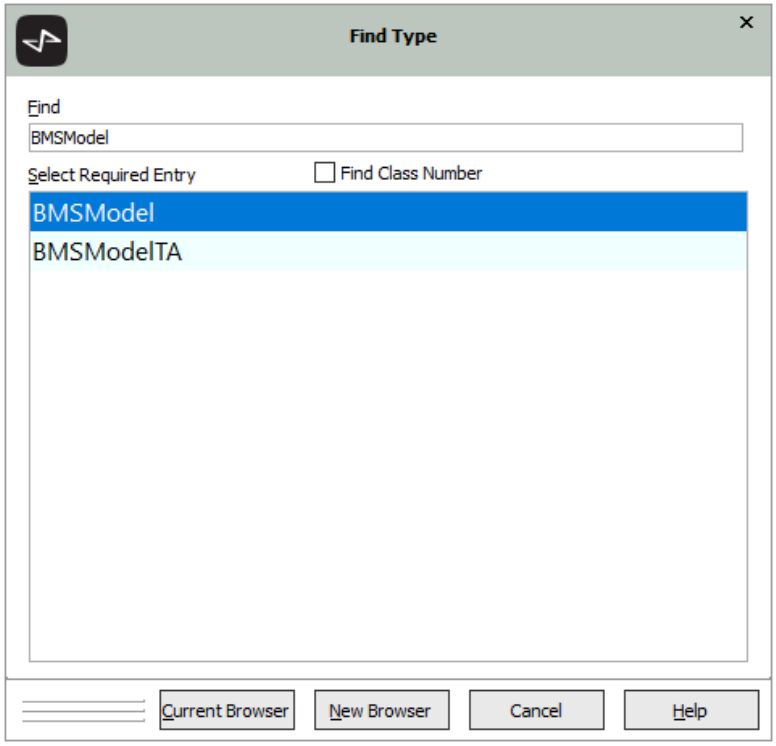
4. Add **cheque** and **savings** map files.

Exercise 6.4 - Adding a Class

In this exercise, you will add a **BMSCustomer** class in the **BankingModelSchema**. If you want to utilize the Transaction Agent Framework (TAF) used in the Erewhon demonstration schemas, you should load the framework classes before you add any of your own classes. This will allow you to add your own classes under the relevant TAF classes, to make it easier to implement that framework in your system. For details, see "Exercise 2 - Loading Transaction Agent Framework (TAF) Classes", earlier in this module.

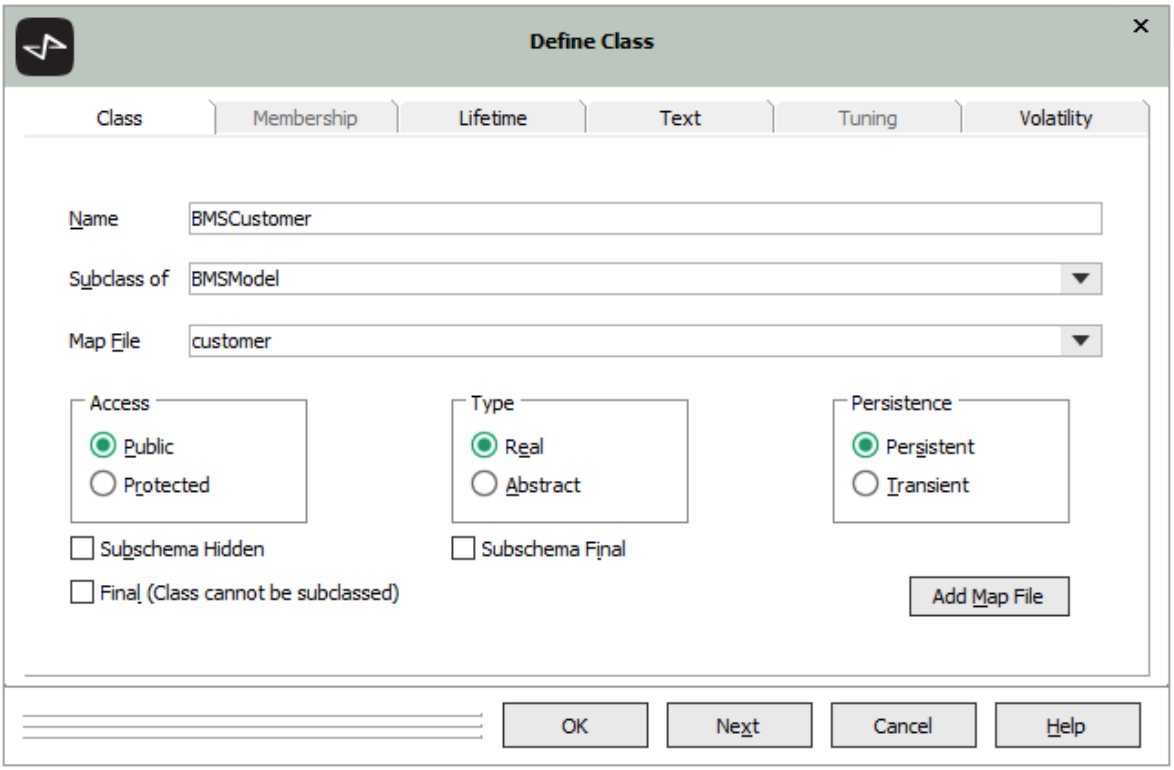
1. Open a Class Browser for the **BankingModelSchema** by clicking the **C** button from the Jade Platform development environment toolbar.
2. Select the **BMSModel** class in the Class Browser.

Because this class won't be visible, you can use the F4 shortcut key to find it, as shown in the following image.



3. Add a subclass to the **BMSModel** class by selecting the Classes menu **Add** command.

4. Enter **BMSCustomer** as the name of the class, select **customer** as the name of the map file, and then click the **OK** button.



Tip Forgotten to add the map file from the Maps Browser? You can also add new map files directly from this dialog, by clicking the **Add Map File** button.

Exercise 6.5 - Adding a Transaction Agent (TA) Class for Customers

In this exercise, you will add a Transaction Agent (TA) class for customers.

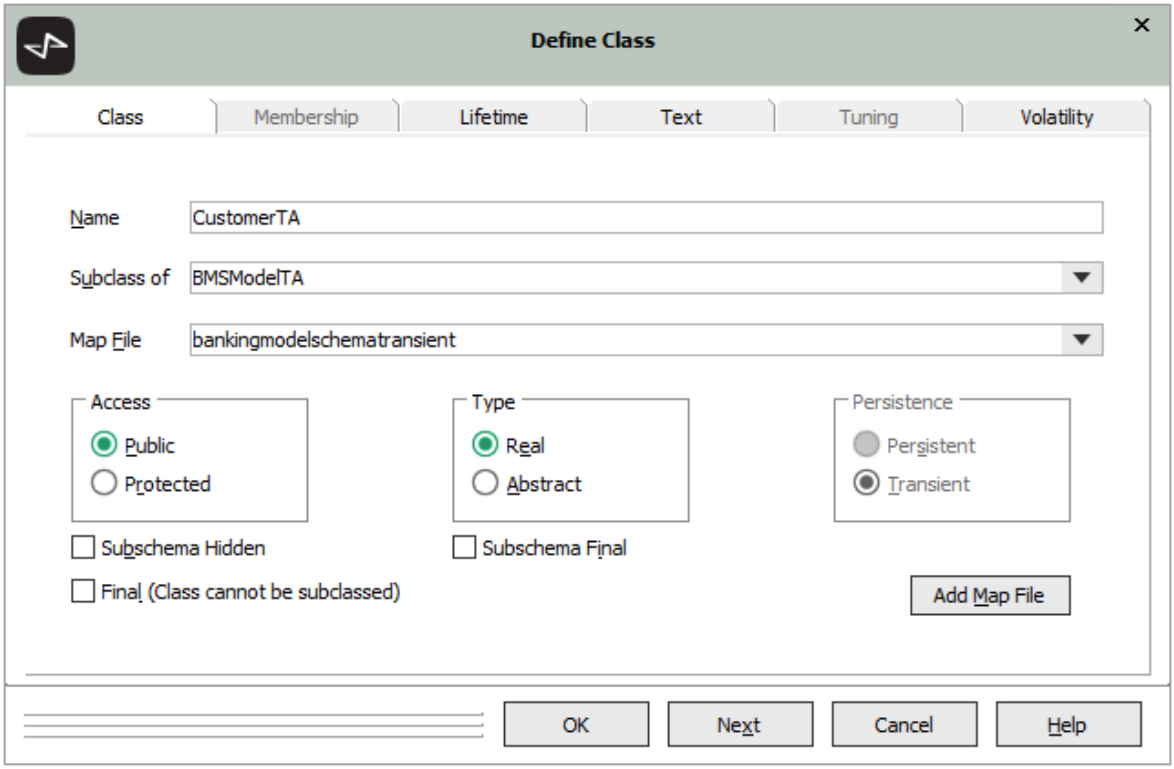
1. Select the **BMSModelTA** class in the Class Browser.

Tip Use F4 to find the class.

2. Add a subclass to the **BMSModelTA** class by selecting the Classes menu **Add** command.

3. Enter **CustomerTA** as the name of the class, select **bankingmodelschematransient** as the name of the map file, and then click the **OK** button.

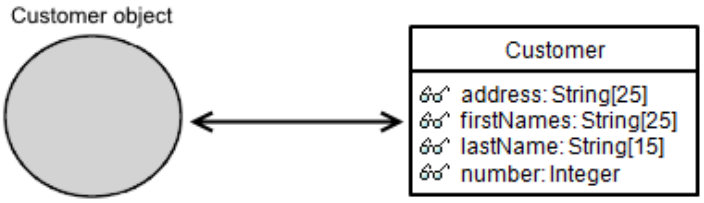
An example of the Define Class dialog is shown in the following diagram.



Instances of a Class

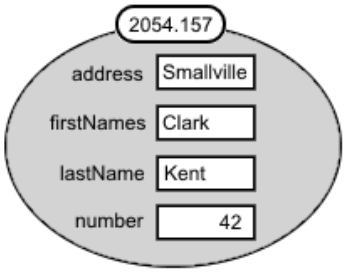
The main component of any Jade application is an object. These objects represent real-world entities. When building a Jade application, you merely mirror reality by creating the components that make up the real-world business system.

An object is an instance of a class. Classes are created by developers as the blueprints or templates that are used to describe and build objects.



At run time, a Jade application works with objects that represent real-world entities; for example, branches, bank accounts, and customers. These objects are instances of a class. They have values that can be changed; for example, the **address** property of a customer.

Each instance has an object identifier (OID), which is assigned to the object when it is created. The OID is used by the Jade Object Manager to keep track of the object. In the following diagram, the OID is 2054.157. The first part (2054) is the class number, so all instances of the **Customer** class begin with 2054. The last part (157) is the instance number, indicating that it is the 157th **Customer** object that was created.



Access to Properties

A property can have one of the following access mode options.

- Public
- Read-only
- Protected

A property can be accessed without restriction by a method in the class in which it is defined (or a subclass). The purpose of the access mode option is to specify what can be done with the property in methods in other classes. As an example, consider the following lines of code involving the **balance** property of **ba**, a bank account object.

```
// Getting the value
write ba.balance;

// Setting the value
ba.balance := 100;
```

Whether the lines of code prevent the method from compiling depends on the access mode option, as shown in the following table.

Access	Getting the value is allowed	Setting the value is allowed
Public	Yes	Yes
Read-only	Yes	No
Protected	No	No

The two extremes are public access, where there are no restrictions on accessing the property, and protected access, where the only way to access the property is through methods that have to be provided in the class. You have to decide the access mode that is appropriate.

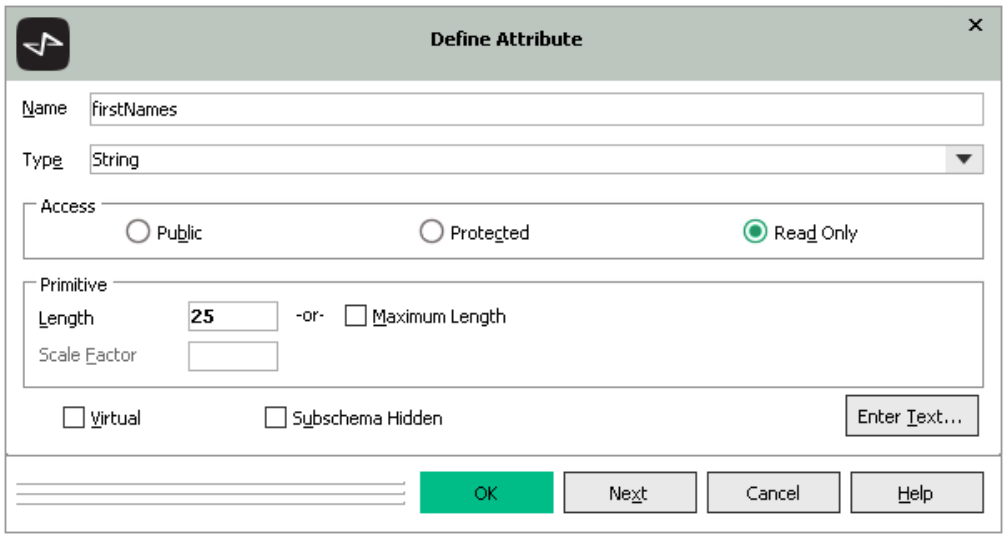
By making a property protected, it cannot be used directly by other classes. It is essentially hidden. However, the motivation for hiding properties is not secrecy. The goal is to provide a simple *interface* to the class; that is, a simple way of working with instances of the class.

In this course, the read-only option (a pragmatic compromise between public and protected) is used for most properties.

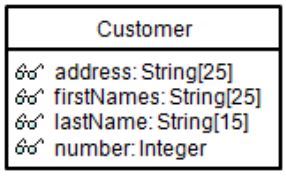
Exercise 6.6 - Adding Attributes

In this exercise, you will add attributes to the **Customer** class.

1. Select the **Customer** class in the Class Browser.
2. Add an attribute by selecting the Properties menu **Add Attribute** command.
3. Perform the following actions on the Define Attribute dialog.
   a. Enter **firstNames** as the name of the class.
   b. Select **String** as the type.
   c. Set the length to **25** characters.
   d. Set the access mode to read-only.
   e. Click the **OK** button.



4. Add the read-only attributes specified in the following UML class diagram.



Make sure that you set the lengths to the values specified in the previous diagram, because the lengths will be relevant later in the course.

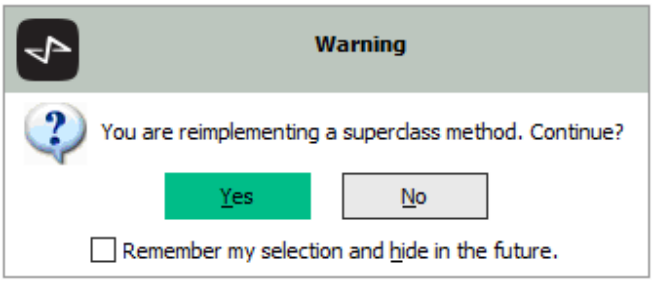
5. Add the **address**, **firstNames**, and **lastName** attribute properties to the **CustomerTA** class but as **Public** attributes rather than **Read Only** attributes.

Exercise 6.7 - Adding a Method to the Customer Class for Persisting Data

In this exercise, you will add a method for persistent data to the **Customer** class.

1. Select the **Customer** class in the Class Browser.
2. Add a method called **setCommonProperties**, by selecting the Methods menu **New Jade Method** command.

   You will get a warning that you are reimplementing a superclass method followed by a second warning that this will recompile referencing methods. Click **Yes** on both of these dialogs, the first of which is shown in the following message box.



3. Code the method as follows.

```
setCommonProperties( pTA : CustomerTA ) updating, protected;

begin
    inheritMethod( pTA );

    self.address := pTA.address.trimBlanks();

    self.firstNames := pTA.firstNames.trimBlanks();

    // Only update the value if it has been changed, to avoid unnecessary inverse maintenance
    if self.lastName <> pTA.lastName then
        self.lastName := pTA.lastName.trimBlanks();
    endif;
end;
```

Note We are deliberately currently not setting the number property in this method. (In a later module, you will code a mechanism to generate a unique value.)

4. Compile the method by pressing F8.

Exercise 6.8 - Adding Required Methods to the CustomerTA Class for Persisting Data

In this exercise, you will add the required methods to the **CustomerTA** class for persisting data.

1. Select the **CustomerTA** class in the Class Browser.
2. Add a method called **populateFromObject**. You will get a warning that you are reimplementing a superclass method. Click **Yes** on this message box.

3. Code the method as follows.

```
populateFromObject( pCustomer : Customer ) updating;

begin
    inheritMethod( pCustomer );

    self.address := pCustomer.address;
    self.firstNames := pCustomer.firstNames;
    self.lastName := pCustomer.lastName;
    self.number := pCustomer.number;
end;
```

4. Compile the method by pressing F8.

5. Add a method called **getModelObjectClass**. You will get a warning that you are reimplementing a superclass method followed by a second warning that this will recompile referencing methods. Click **Yes** on both of these message boxes.

6. Code the method as follows.

```
getModelObjectClass(): Class protected;

begin
    return Customer;
end;
```

7. Compile the method by pressing F8.

8. Add a method called **initialize**. You will get a warning that you are reimplementing a superclass method. Click **Yes** on this message box.

9. Code the method as follows.

```
initialize() updating;

begin
    inheritMethod();

    self.address := null;
    self.firstNames := null;
    self.lastName := null;
end;
```

10. Compile the method by pressing F8.

Exercise 6.9 - Testing with a JadeScript Method

In this exercise, you will add a **createCustomer** JadeScript method to test the **create** method.

1. Select the **JadeScript** class in the Class Browser.

2. Add a method called **createCustomer**, by selecting the Methods menu **New Jade Method** command.

3. We now code the method using the TAF framework class to create the persistent customer, as follows.

```
createCustomer();

vars
    customerTA : CustomerTA;

begin
    create customerTA transient;

    customerTA.address := "Gotham City";
    customerTA.firstNames := "Bruce";
    customerTA.lastName := "Wayne";

    if not customerTA.persistEntity( BMS_Full_update ) then
        write customerTA.getFullErrorDetails();
        return;
    endif;

    write "Customer created successfully";

epilog
    delete customerTA;  // Prevent transient leak
end;
```

4. Compile the method by pressing F8.

5. Execute the method through the debugger, using the **Step into next statement** toolbar button to see the sequence in which code is executed.

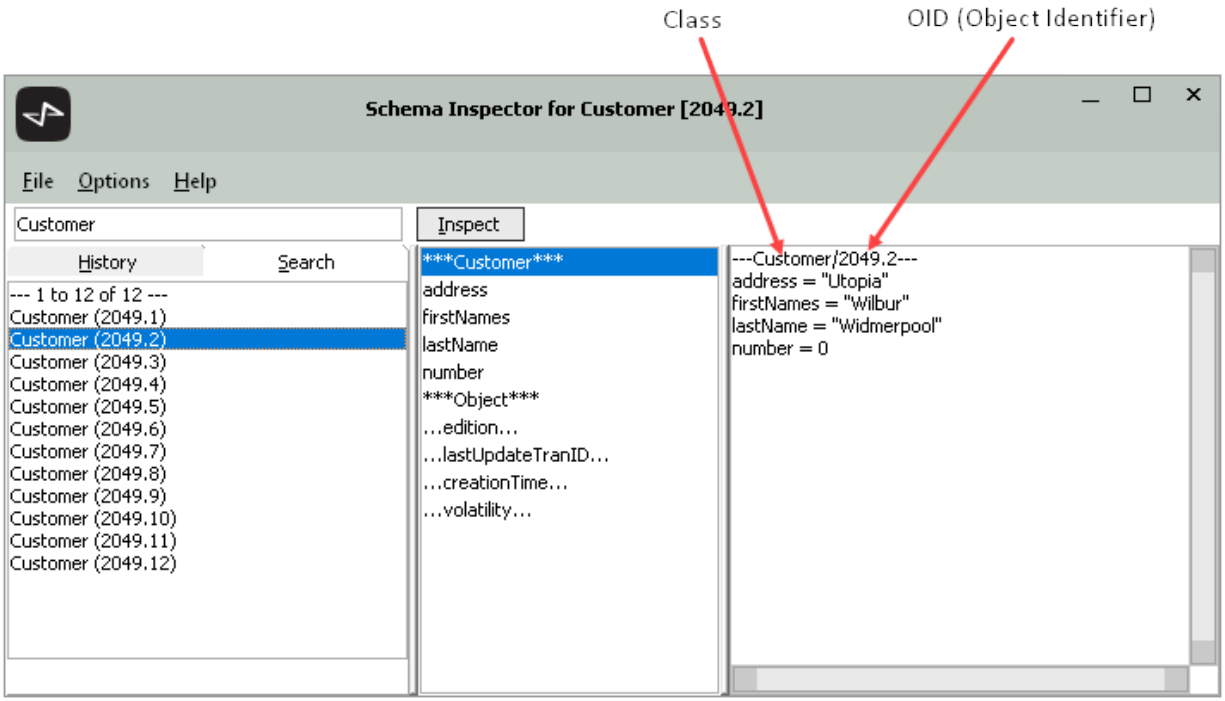
6. Change the data in the **createCustomer** JadeScript method and then execute the method again.

There should be two customers in the database.

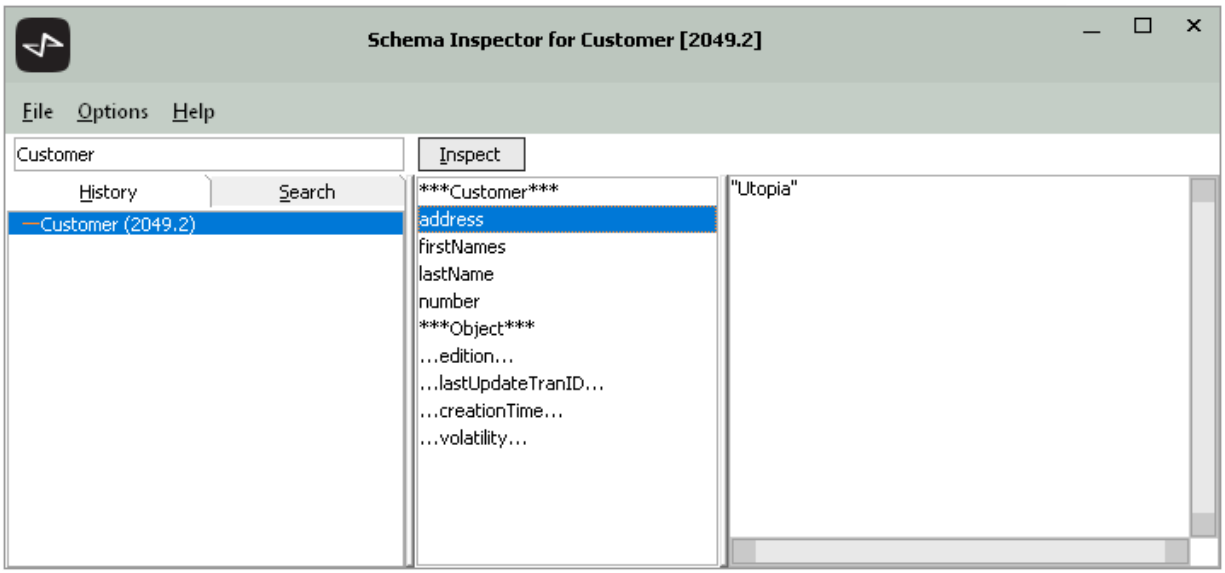
In this method, the **persistEntity** method on the transaction agent class is used to create a persistent instance of the associated model class. Later in the course we will use the same **persistEntity** method to update existing instances of the associated model class.

Inspecting Database Objects

You can inspect persistent database objects using the Object Inspector. The following diagram shows the customer objects that you created in the previous exercise.



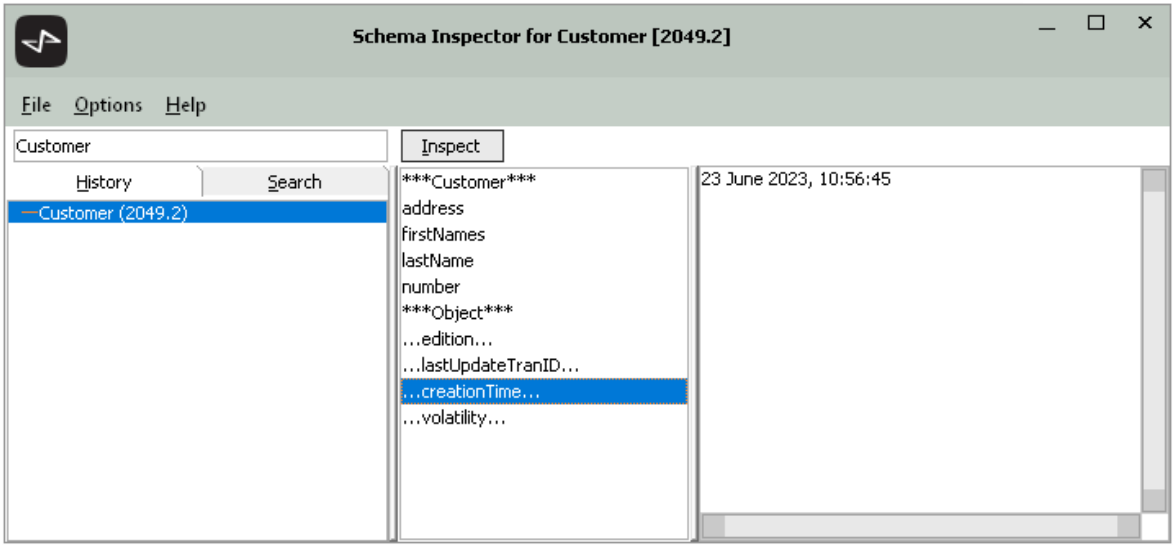
If you double-click an object in the left-hand pane, a new Object Inspector window is opened to display the object in detail.



If you single-click a property in the middle pane, the value of the property is displayed in the right-hand pane. Other information about the object that is displayed is the:

- **edition**, which is one (**1**) for the first transaction as it creates the object, and it is incremented for each subsequent transaction that updates the object.

- **creationTime**, which is the date and time at which the object was created, as shown in the following image.



To use the same form instead of a new Schema Collection Inspector form each time a new object is selected for inspection, click the **Use Same Window** command in the Options menu. When the **Use Same Window** command is checked, each double-click of an object in an Inspector form re-uses the same form to display the selected object, replacing the previously displayed object. A pane at the left of the form contains a hierarchical list box displaying all of the objects that have previously been inspected. The hierarchy indicates the history of how the objects were inspected.

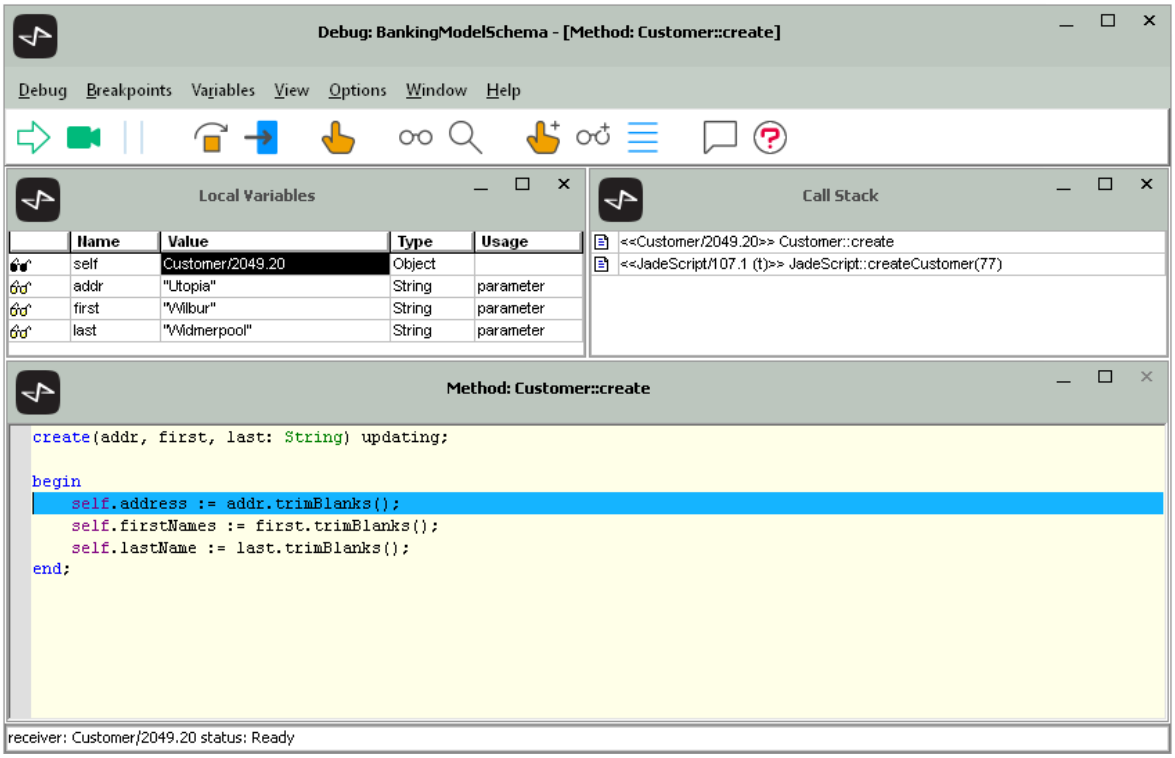
The entries display the value of the name property if it exists in the object, followed by the class name and the Jade object identifier (oid). Clicking on an entry in the hierarchical history list at the left of the form displays the selected object again.

The ways in which you can invoke the Object Inspector are as follows.

- In the Class Browser, select the **Customer** class and then select the Classes menu **Inspect Instances** command (or press Ctrl+I).
- In a method, code one of the following instructions.

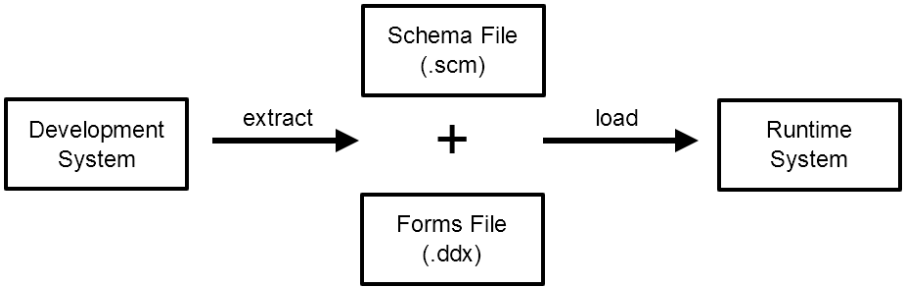
```
cust.inspect();
cust.inspectModal();
```

- In the debugger, select a variable and then press Ctrl+I.



Extracting and Loading Schemas

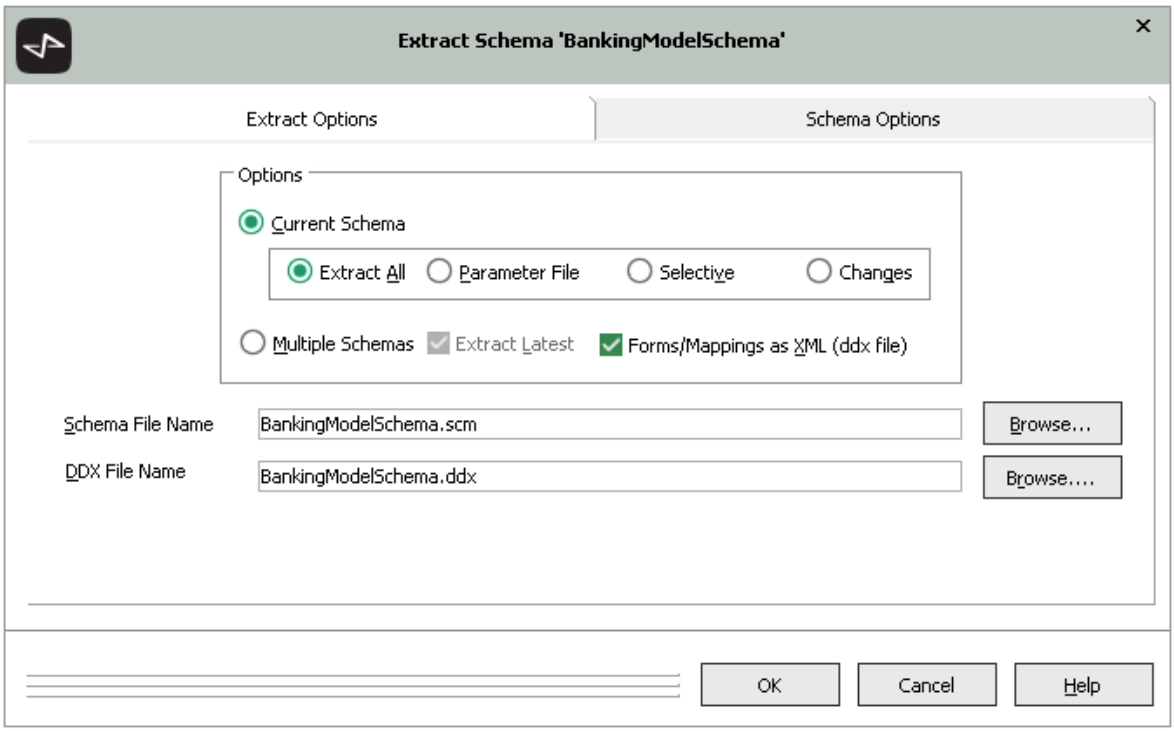
You can extract a complete schema, parts of a schema, or multiple schemas; for example, as a backup before you reorganize your database or you install a new release of the Jade Platform. You can load the extract files into another Jade system. The deployment mechanism for a Jade system is shown in the following diagram.



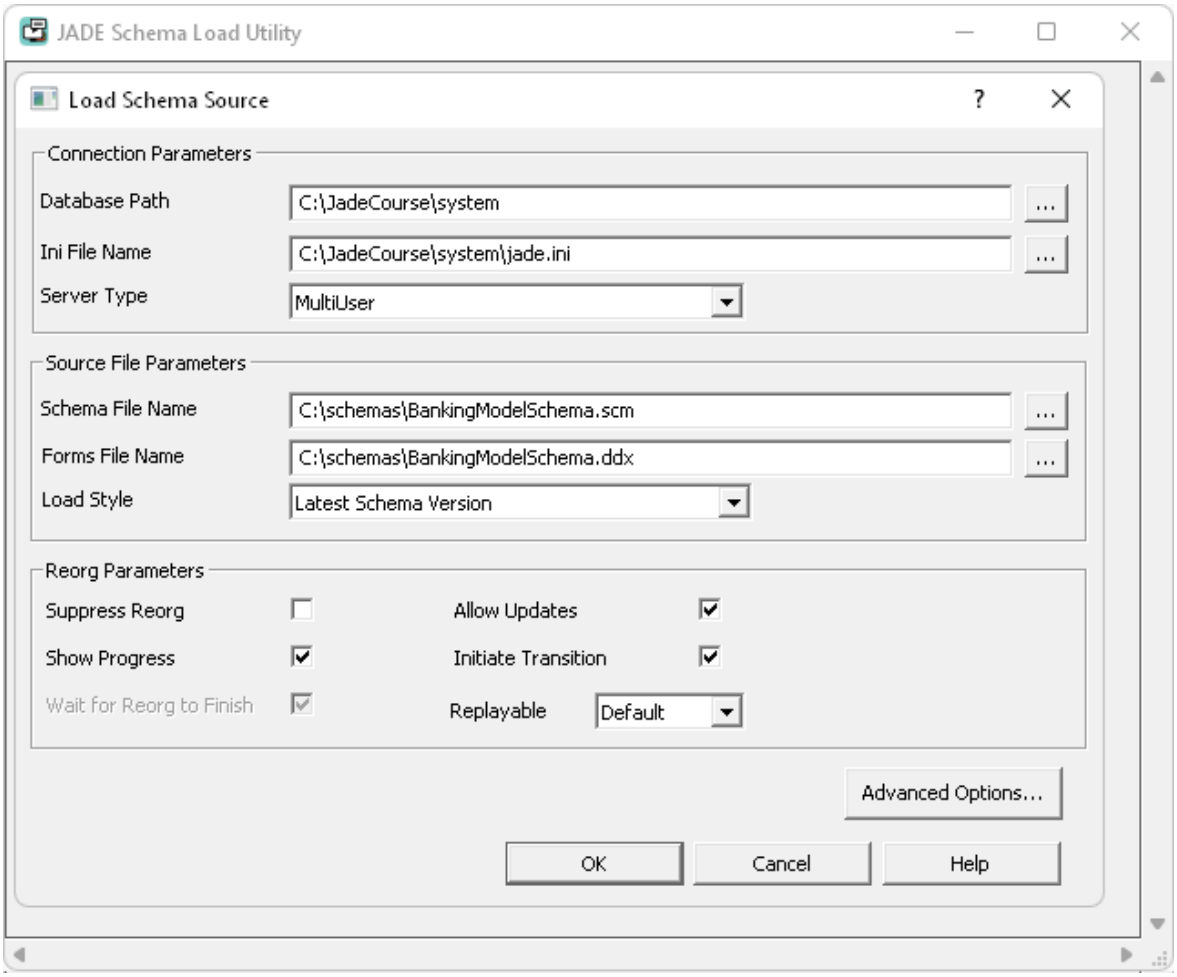
The extract process creates two files.

- The schema file contains class definitions, method code, and so on, from the Class Browser.
- The forms definition file contains the forms that you designed in the JADE Painter.

To extract a schema selected in the Schema Browser, use the Schema menu **Extract** command. Check the **Forms/Mappings as XML (ddx file)** check box to extract the form descriptions in the newer .**ddx** format, which is more human-readable than the legacy .**ddb** format.



To load a schema, use the Schema menu **Load** command from the Schema Browser. Alternatively, if the Jade Platform development environment is not available, you can use the JADE Schema Load utility.



Exercise 6.10 - Inspecting Objects

In this exercise, you will inspect the objects you created in the previous exercise.

1. Select the **Customer** class in the Class Browser.
2. Select the Classes menu **Inspect Instances** command or press Ctrl+I.
3. Inspect two customers.
4. Select the File menu **Close All** command to close the inspector window or all of the open inspector windows if you are not using the same window (that is, the same form).

Exercise 6.11 - Removing Test Objects

In this exercise, you will remove the customers you created previously.

1. Select the **JadeScript** class in the Class Browser.
2. Add a method called **removeTestData**, which is coded as follows.

```
removeTestData();

begin
    beginTransaction;
    Customer.instances.purge();
    commitTransaction;
end;
```

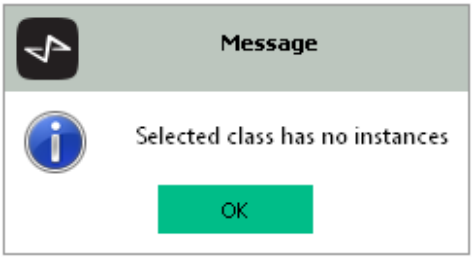
In this method:

- The **instances** property for a class is a collection that is created dynamically from information in the database files.

  Note The **instances** method bypasses the mechanisms in Jade that ensure information is current.

- The **purge** method is a generic method for collections that removes the objects from the collection and then deletes the objects.
- Persistent objects can be deleted only within a transaction.

3. Execute the method.
4. Inspect instances of the **Customer** class. The following message box should be displayed.

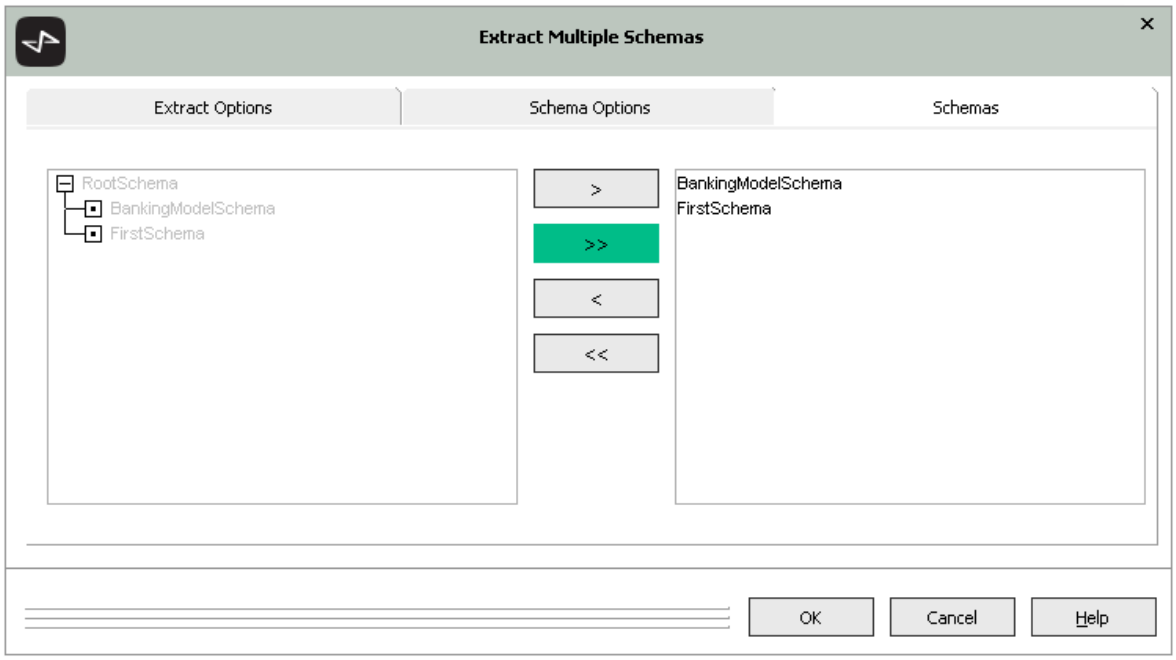


Exercise 6.12 - Extracting Multiple Schemas

In this exercise, you will extract **BankingModelSchema** and **FirstSchema** with a multiple schema extract.

1. Select the Schema Browser.
2. Select the Schema menu **Extract** command.
3. Select the **Multiple Schemas** option.
4. Change the name in the **Multi Extract File** text box to **Banking.mul** and then click the **Browse** button to specify where the extract files should be located.
5. Check the **Forms/Mappings as XML (ddx file)** check box.

6. Select the **Schemas** tab and then click the **>>** button to select both schemas.



7. Click the **OK** button.

8. Open the **Banking.mul** file in Notepad. It lists the schema and forms definition files that were extracted.

```
#MULTIPLE_SCHEMA_EXTRACT
BankingModelSchema.scm BankingModelSchema.ddx
FirstSchema.scm FirstSchema.ddx
```

Module 7 Root Object

This module contains the following topics.

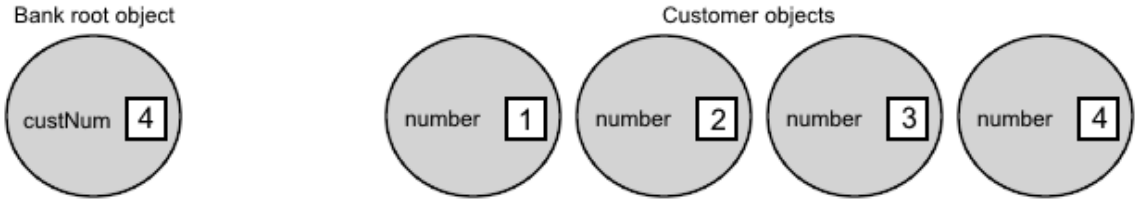
Introduction

A common design strategy is to have a class that has a single instance representing the business or organization that the software serves. The single instance is called the *root object*.

In the banking system, the **Bank** class is the class that will have the root object.

One of the main uses of the root object is to own complete collections of instances of a class, which are needed by the application. You will use collections in a later module to enable a customer to have a collection of his or her bank accounts. However, the application requires a more-comprehensive collection of bank accounts belonging to all customers. The root object is the usual place to store it.

A more immediate use of the root object will be to generate a sequential number for each new customer. The bank root object will store the number used for the latest customer. When a new customer is created, the bank object will increment the stored number and return that value.

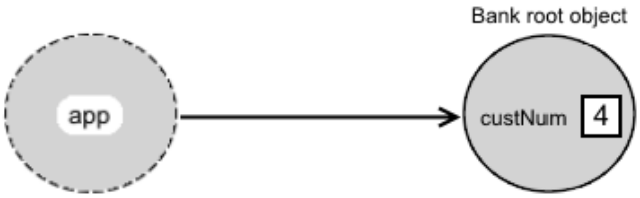


Initializing the Root Object

The root object, which is the single instance of the **Bank** class, must be easily accessible from code anywhere in an application or JadeScript method. You could use the **firstInstance** or **lastInstance** method every time the root object is needed, as follows.

```
Bank.firstInstance()
```

The **firstInstance** or **lastInstance** methods are expensive because they retrieve the OID directly from the database files. A better approach is to use the **app** object to store a reference to the root object.



If the reference to the root object is called **myBank**, using the naming convention of prefixing references to single objects with **my**, the root object can be accessed in code as follows.

```
app.myBank
```

In addition to setting up a **myBank** reference of type **Bank** in your **Application** subclass, you must ensure that:

- An instance of the **Bank** class is created if one does not exist
- The **myBank** reference is initialized to the singleton instance

This will be implemented in an **initialize** method in your **Application** subclass.

Note Before the root object can be accessed with **app.myBank**, an application or JadeScript method must execute **app.initialize**.

Constructor

A constructor is a method in a class that is automatically called when an instance of that class is created. The name of the method must be **create**. A constructor is often used to set default values for properties.

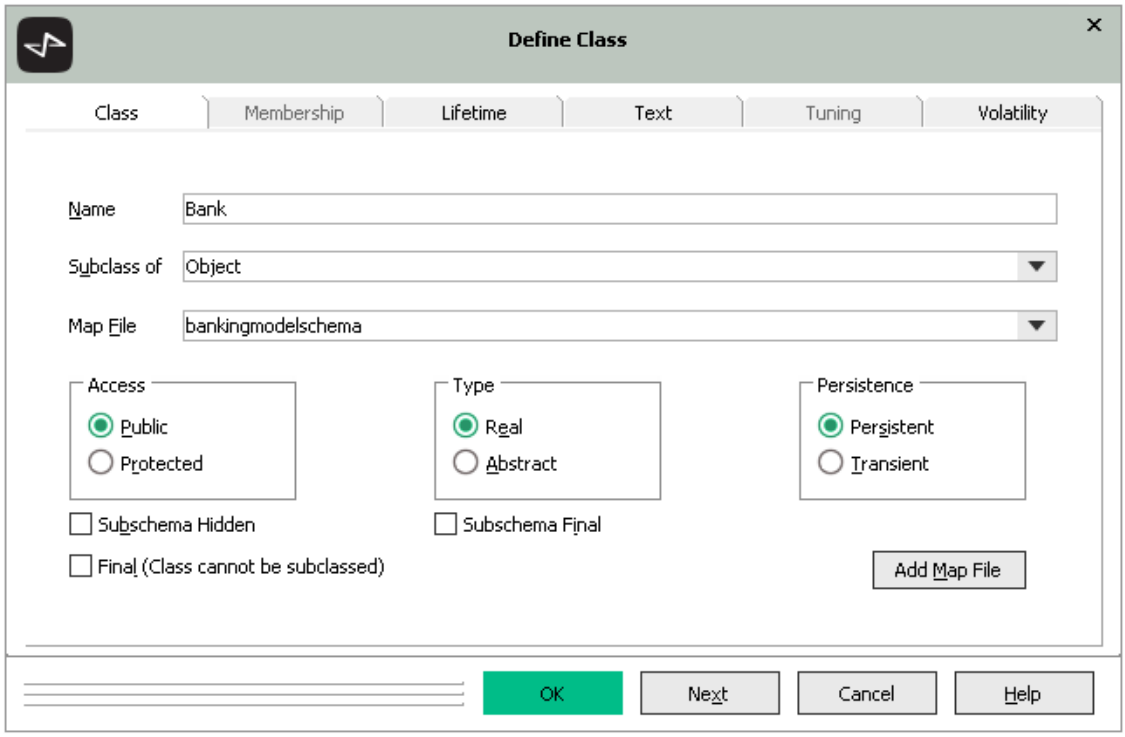
When a **Customer** object is created, you will use a constructor to set the value of the **number** attribute to the value returned by the **nextCustNum** method of the root object.

Exercise 7.1 - Adding the *Bank* Class

In this exercise, you will add the **Bank** class in the **BankingModelSchema**. The class will have a **custNum** attribute and a **nextCustNum** method to increment this value and return the result.

1. Select the **BMSPersistent** class in the Class Browser.
2. Add a subclass to the **BMSPersistent** class by selecting the Classes menu **Add** command.

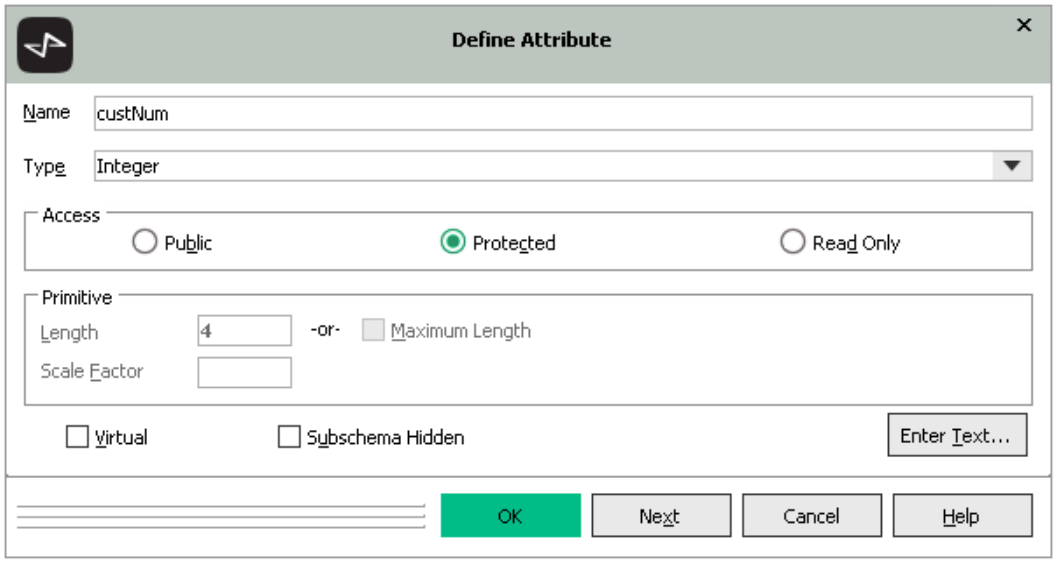
3. Enter **Bank** as the name of the class, select **bankingmodelschema** as the map file, and then click the **OK** button.



Note The default (**bankingmodelschema**) map file is fine, as we will only ever be instantiating one **Bank** object.

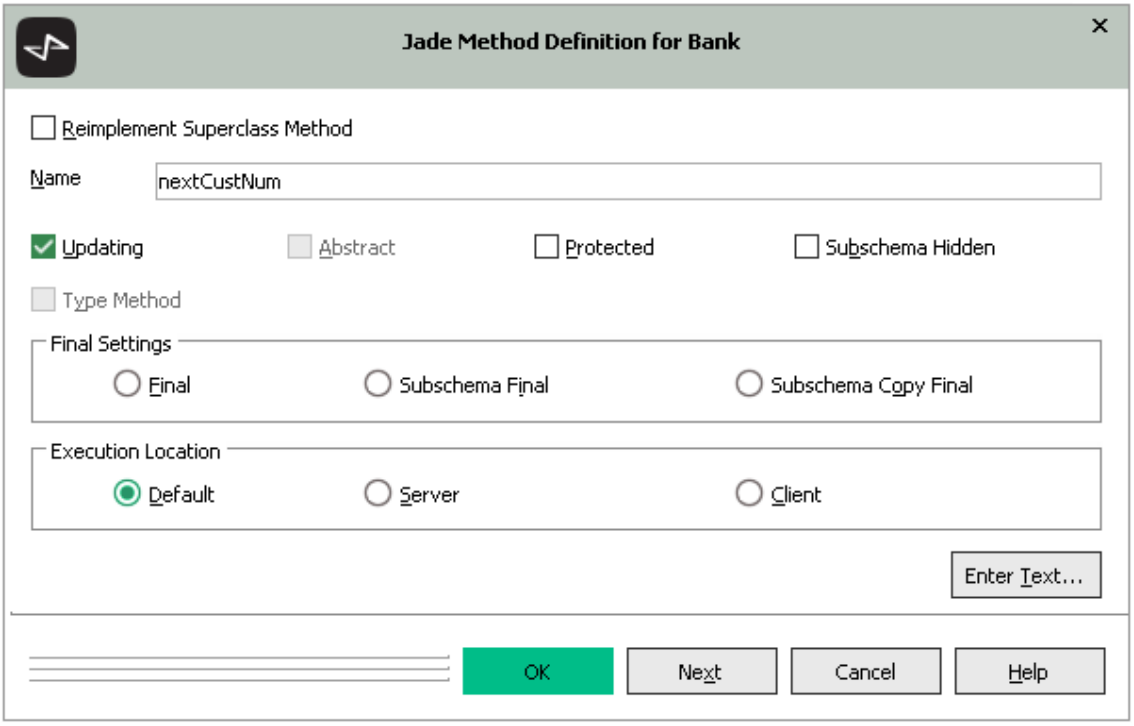
4. Add an attribute called **custNum**, by selecting the Properties menu **Add Attribute** command.

   Select **Integer** as the type, set the access mode to protected, and then click the **OK** button.



5. Add a method called **nextCustNum**, by selecting the Methods menu **New Jade Method** command.

Check the **Updating** option, because the method will increment the **custNum** attribute.



6. Code the method as follows.

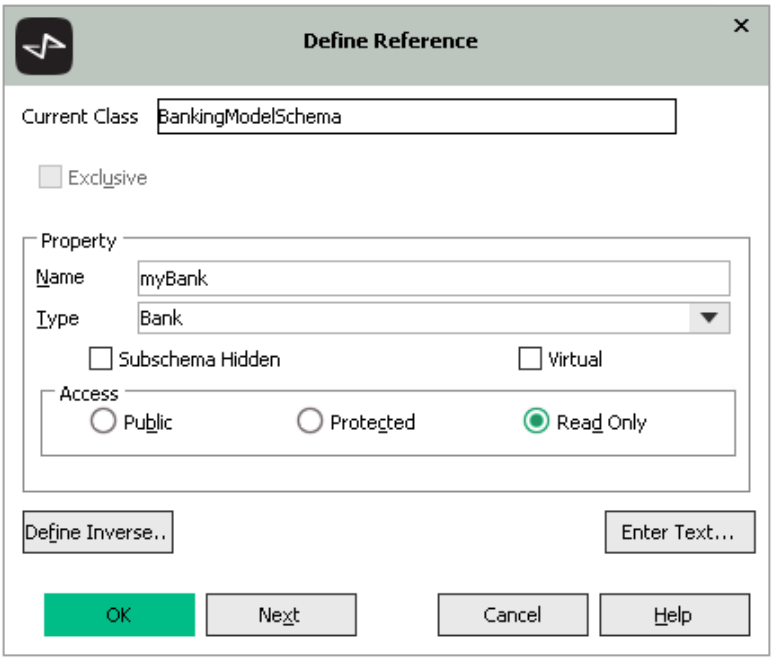
```
nextCustNum() : Integer updating;

begin
    self.custNum += 1;

    return self.custNum;
end;
```

Exercise 7.2 - Adding a *myBank* Reference and *initialize* Method

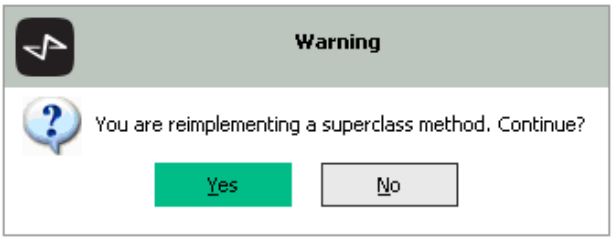
In this exercise, you will add a reference to the root object in your **Application** subclass.

1. Select your **Application** subclass in the Class Browser. This will have the same name as your schema, in this case, **BankingModelSchema**.
2. Add a reference by selecting the Properties menu **Add Reference** command.

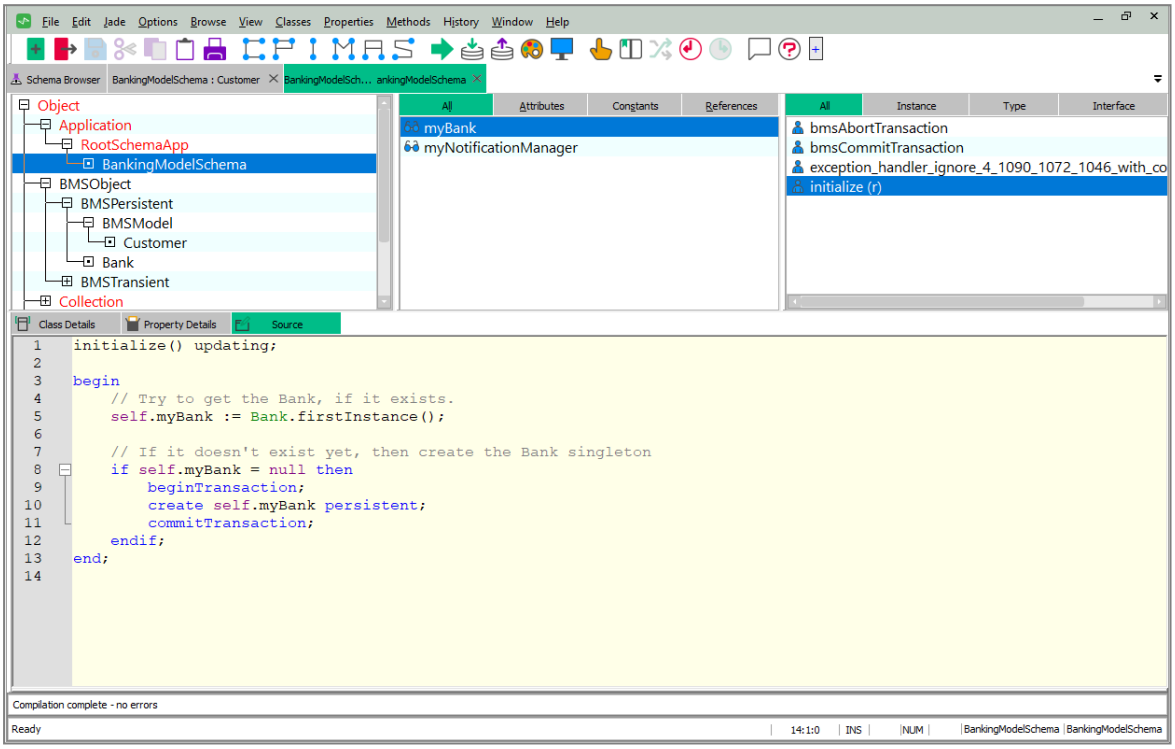
3. Enter **myBank** as the name, select **Bank** as the type, set the access mode to read-only, and then click the **OK** button.



4. Add a method called **initialize**. A message box warns you that there is already a method of that name in the **Application** hierarchy. Click the **Yes** button, to continue.



5. Complete the coding of the **initialize** method, as shown in the following image.



Note Before the root object can be accessed with **app.myBank**, an application or JadeScript method must execute **app.initialize**.

Exercise 7.3 - Modifying the *Customer::onCreate* Method

In this exercise, you will implement the **onCreate** method on the **Customer** class to obtain a unique identifier (ID) number from the **Bank** class.

1. Select the **Customer** class in the Class Browser.
2. Add a method called **onCreate** and code it as follows. You will get a warning that you are reimplementing a superclass method. Click **Yes** on this message box.

```
onCreate( pTA : CustomerTA ) updating;

begin
    inheritMethod( pTA );

    self.number := app.myBank.nextCustNum();
end;
```

3. Compile the method by pressing F8.

4. Test that the **onCreate** method works by adding **app.initialize** to the **createCustomer** JadeScript method, as follows.

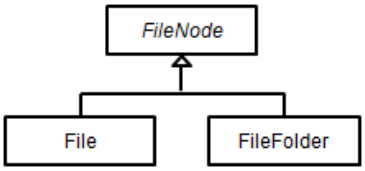
```
createCustomer();

vars
    customerTA : CustomerTA;

begin
    app.initialize();

    create customerTA transient;

    customerTA.address := "Gotham City";
    customerTA.firstNames := "Bruce";
    customerTA.lastName := "Wayne";

    if not customerTA.persistEntity( BMS_Full_update ) then
        write customerTA.getFullErrorDetails();
        return;
    endif;

    write "Customer created successfully";

epilog
    delete customerTA;  // Prevent transient leak
end;
```

5. Execute the JadeScript method twice, using the debugger.

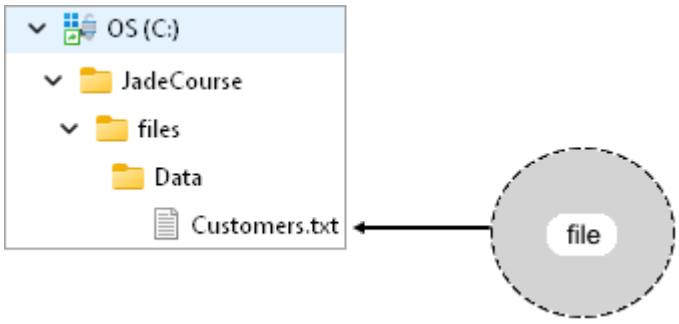
6. Inspect the two new customers. The value of the **number** attribute should be **1** for the first customer and **2** for the second customer.

Working with Files

A **Customers.txt** file has been provided to bulk-load hundreds of customers. In a later exercise, you will write a JadeScript method to open this file, read each line, and then create a customer object from the text that has been read. **RootSchema** has a hierarchy of classes for working with files and folders in your code.



To work with a file, you create a transient instance of the **File** class and set its **fileName** property to the full path name of the file.



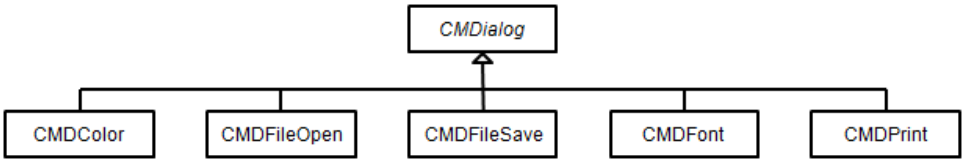
The following methods of the **File** class are used to read the information in a file.

Method	Description
readLine	Returns the text from the next line in the file
endOfFile	Returns **true** when the end of the file is reached

Working with Common Dialogs

Rather than hard-coding the full path name of a file, you can ask the user to select the file by using the Microsoft Open File dialog, which is one of the Microsoft common dialogs. To use one of these dialogs, create an instance of a **CMDialog** subclass.

The **CMDialog** hierarchy of classes is defined in **RootSchema**.



The **open** method of the **CMDFileOpen** class returns zero (**0**), to indicate that the user has successfully opened a file, in which case the **fileName** attribute contains the full path name of the file that was opened. If the user clicks the **Cancel** button, the **open** method returns one (**1**).

Exercise 7.4 - Reading from a File

In this exercise, you will use the data in the **Customers.txt** file to create hundreds of customers.

1. Add a JadeScript method called **createCustomersFromFile** and then code it as follows.

```
createCustomersFromFile();

vars
    inputFile : File;
    nextLineInFile : String;
    customerTA : CustomerTA;

begin
    app.initialize();

    create customerTA transient;
    create inputFile transient;

    inputFile.fileName := "c:\JadeCourse\Files\Customers.txt";
    inputFile.mode := File.Mode_Input;
    inputFile.open();

    while not inputFile.endOfFile() do
        nextLineInFile := inputFile.readLine();
        customerTA.initialize();
        customerTA.address := nextLineInFile[ 41 : end ];
        customerTA.firstNames := nextLineInFile[ 16 : 25 ];
        customerTA.lastName := nextLineInFile[ 1 : 15 ];
        customerTA.persistEntity( BMS_Full_update );
    endwhile;

    inputFile.close();

epilog
    delete inputFile;
    delete customerTA;
end;
```

   Although the **createCustomersFromFile** method executes as expected in an ANSI Jade system, exception 5011 (*Record truncated to maxRecordSize characters*) is raised in a Unicode Jade system, because ANSI text files such as **Customers.txt** file differ from Unicode text files.

   To tell Jade the file type of **Customers.txt**, add one of the following lines after the **create inputFile transient;** line in your JadeScript.

```
file.kind := File.ANSI; // works for ANSI text files

file.kind := File.Kind_Unknown_Text; // works for ANSI and Unicode text files
```

2. Execute the method and then inspect the customers that are created.

In this method:

- **app.initialize** is executed as the first instruction, so that the method can access the root object.
- The condition **not inputFile.endOfFile** tests that there is still more information to be read.
- The transient **File** object is deleted at the end of the method.

  As there is no garbage collection in Jade, you should delete transient objects when they are no longer needed.

> **Note** Deleting the **File** object also closes it, and avoids the file being left in use.

- The **epilog** section contains instructions that should always be executed. If a **return** instruction is encountered before the end of the method or an instruction raises an exception, **epilog** instructions are always executed before the method returns.

Exercise 7.5 - Using the File Open Dialog

In this exercise, you will enhance the **createCustomersFromFile** JadeScript method by using the Microsoft Open File dialog to select the **Customers.txt** file.

1. Execute the **removeTestData** JadeScript method.
2. Modify the **createCustomersFromFile** JadeScript method, as follows.

```
createCustomersFromFile();

vars
    fileOpenDialog : CMDFileOpen;
    inputFile : File;
    nextLineInFile : String;
    customerTA : CustomerTA;

begin
    app.initialize();

    create fileOpenDialog transient;
    if fileOpenDialog.open() <> 0 then
        // Exit early, as the user did not select a file
        return;
    endif;

    create customerTA transient;
    create inputFile transient;

    // inputFile.fileName := "c:\JadeCourse\Files\Customers.txt";
    inputFile.fileName := fileOpenDialog.fileName;
    inputFile.mode := File.Mode_Input;
    inputFile.open();

    while not inputFile.endOfFile() do
        nextLineInFile := inputFile.readLine();
        customerTA.initialize();
        customerTA.address := nextLineInFile[ 41 : end ];
        customerTA.firstNames := nextLineInFile[ 16 : 25 ];
        customerTA.lastName := nextLineInFile[ 1 : 15 ];
        customerTA.persistEntity( BMS_Full_update );
    endwhile;

    inputFile.close();

epilog
    delete fileOpenDialog;
    delete inputFile;
    delete customerTA;
end;
```

Execute the **createCustomersFromFile** method and then inspect the customers that are created.

In this method:

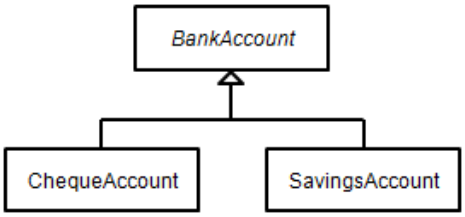
- **app.initialize** is executed as the first instruction, so that the method can access the root object.
- A transient **CMDFileOpen** object is created and it is deleted in the **epilog** section.
- The method is exited from early if the user fails to open a file successfully.

Module 8 Inheritance and Polymorphism

This module contains the following topics.

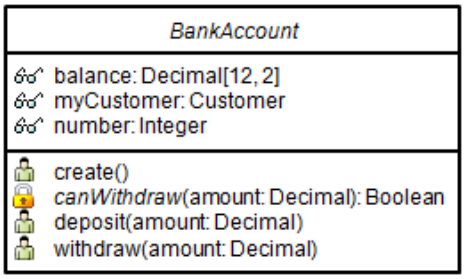
Introduction

In this module, you will create a hierarchy of bank account classes.



In a similar pattern to the **RootSchema** hierarchies of **FileNode** classes and **CMDialog** classes, the bank account classes have an abstract superclass with common properties and methods and real subclasses, which can be instantiated.

The properties and methods of the **BankAccount** class are shown in the following class diagram.



All of the properties are read-only, to limit updating to methods in the class; for example, the **balance** property will be updated only by the **deposit** and **withdraw** methods.

Protected Methods

Methods are either public, which means they are part of the interface of the class, or they are protected. A protected method (sometimes known as a *helper* method) can be called only by a method in the same class or a subclass. Unlike public methods, it is not part of the interface of the class.

The purpose of the **canWithdraw** method in the **BankAccount** class is to check that there are sufficient funds in the account for the withdrawal to proceed. It is called by the **withdraw** method and if it returns **true**, the withdrawal is allowed. If it returns **false**, a message box is displayed, advising the user that there are insufficient funds, and that consequently the withdrawal is not possible.

The **canWithdraw** method is not called under any other circumstances. For that reason, it has been made protected by adding the word **protected** to the method signature.

```
canWithdraw(amount: Decimal): Boolean protected;
```

Real versus Abstract

The terms *real* and *abstract* apply to classes and to methods.

The consequences of making the **BankAccount** class abstract are:

- Instances of the **BankAccount** class itself are not allowed. (You can create instances of the **ChequeAccount** and **SavingsAccount** subclasses.)
- Methods can be abstract or real. (Real classes like the **Customer** class cannot have abstract methods.)
  - Real methods have an implementation; that is, a method body for instructions.

```
some_method();

vars
    // Local variables
begin
    // Your code here
end;
```

  - Abstract methods have only the signature line. The implementation is deferred to the subclasses.

```
some_method() abstract;
```

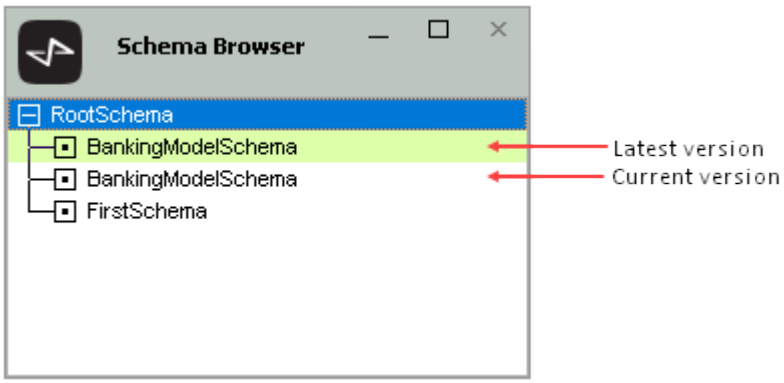
An abstract method specifies the parameters and return type that the implementation of the method inherits.

The code for the **canWithdraw** method is different for **ChequeAccount** objects and **SavingsAccount** objects. For **ChequeAccount** objects, a withdrawal will be allowed provided that the overdraft limit is not exceeded. For **SavingsAccount** objects, there is no overdraft facility so the requirement is that the **balance** attribute should not be allowed to become negative.

The **canWithdraw** method is abstract in the **BankAccount** class, to defer the implementation to the subclasses.

Schema Versions

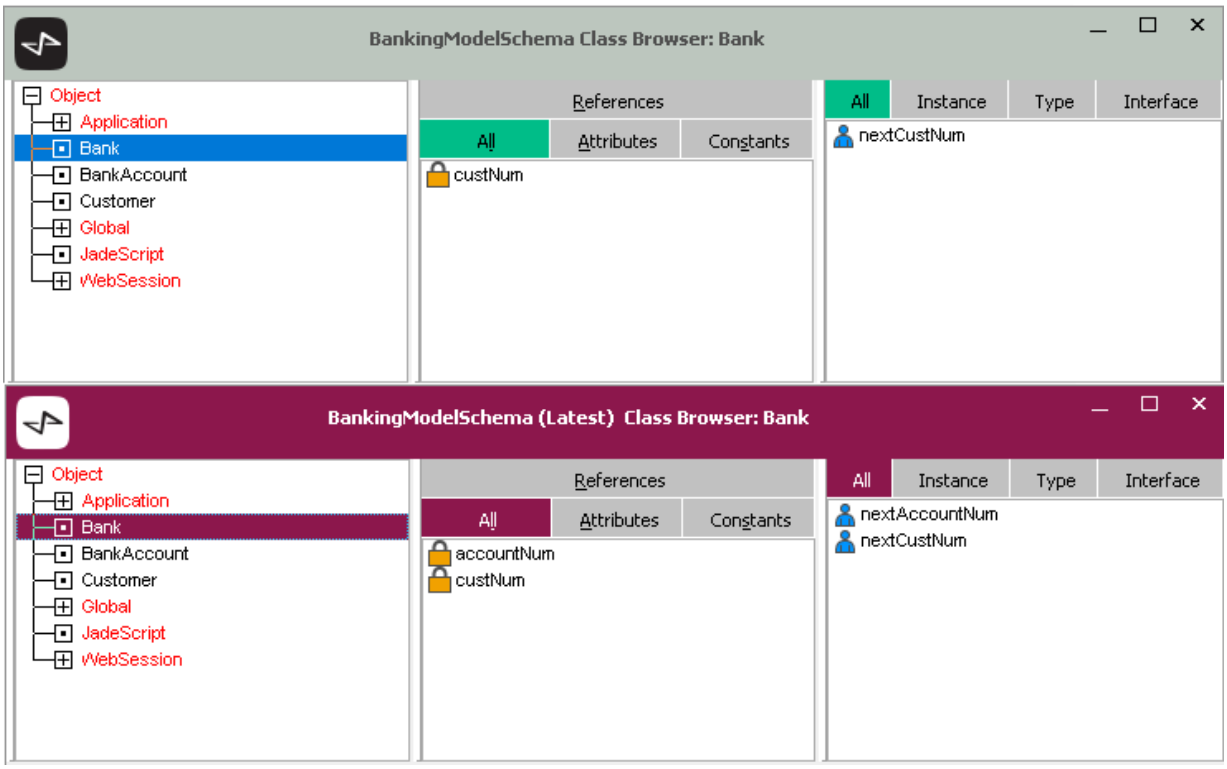
From the schema browser, you can create another version of your schema.



The current version of a schema contains the current definitions of the classes. Applications and JadeScript methods can be run only with the current version.

The latest version contains changed class definitions that have yet to be implemented; that is, brought into effect.

The browsers for the current and latest version are colored differently. The following image shows the current definition of the **Bank** class and the changed definition in the latest version, which has an additional property and method.



The changes in the latest version can be brought into effect by selecting Schema menu **Reorg Schema** command, or by pressing the **Schema Needs Reorg** toolbar button.

The reorganization restructures the data to be consistent with the latest version. After the reorganization, there is a single schema version; the latest version ceases to exist.

Alternatively, if you want to abandon the changes and not perform a reorganization, you can use the Schema menu **Unversion** command to discard the latest version.

The advantages of making changes in the latest schema are:

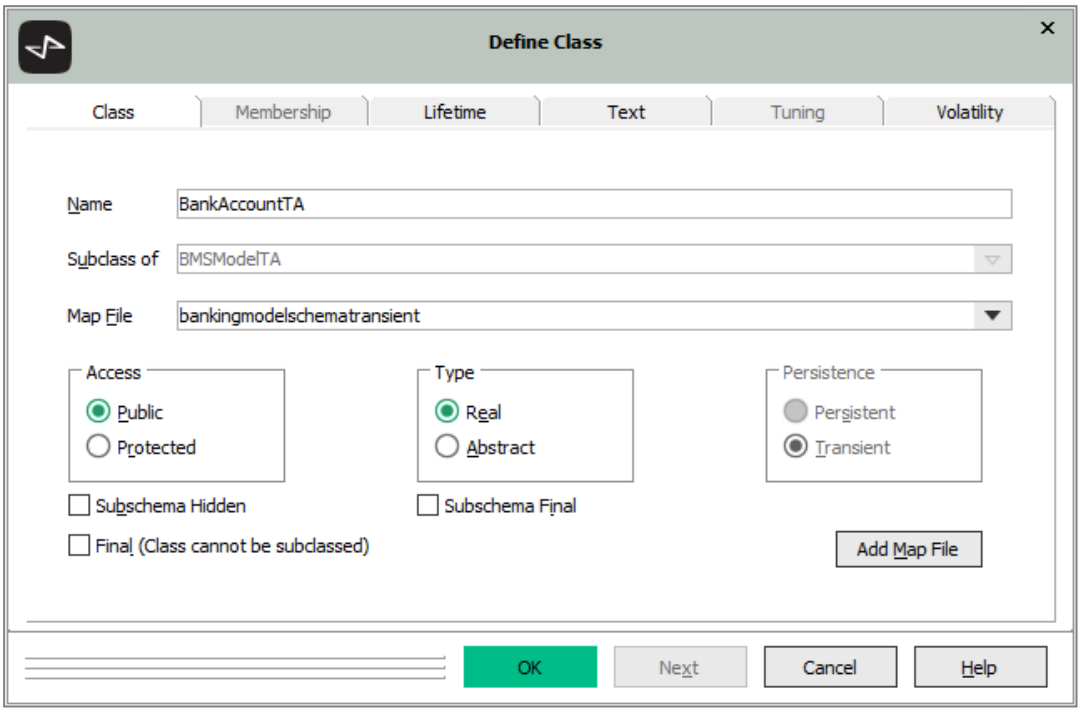
- Implementation of changes can be deferred until the most-convenient time.
- The current version is available while the latest version is reorganized. Only the final transition step requires the system to be offline.

Exercise 8.1 - Adding the BankAccountTA Class

In this exercise, you will add a **BankAccountTA** class in the **BankingModelSchema**.

1. Select the **BMSModelTA** class in the Class Browser.
2. Add a subclass to the **BMSModelTA** class by selecting the Classes menu **Add** command.

3. Enter **BankAccountTA** as the name of the class, select **bankingmodelschematransient** as the map file, and then click the **OK** button.



4. Add a public **Balance** attribute of type **Decimal**, length **12**, and scale factor **2**.
5. Add a public **Number** attribute of type **Integer**.
6. Add a public **myCustomer** reference of type **Customer**.
7. Add a new **initialize** method.
8. Code the method as follows.

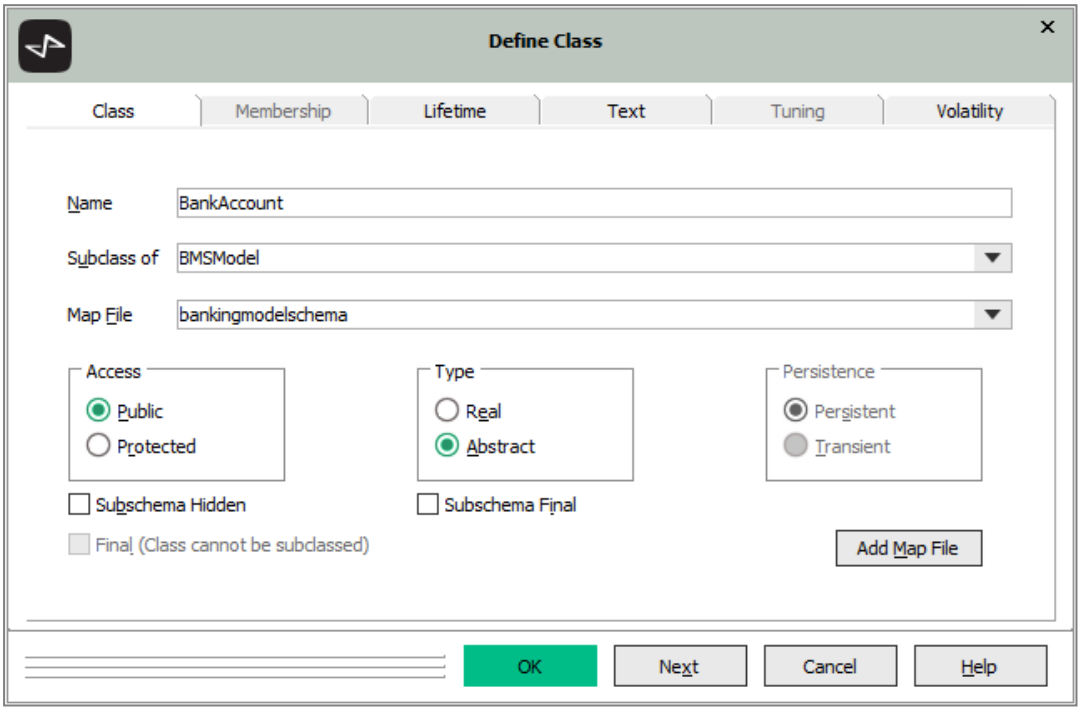
```
initialize() updating;

begin
    inheritMethod();

    self.balance := null;

    self.myCustomer := null;
end;
```

Exercise 8.2 - Adding an Abstract Class

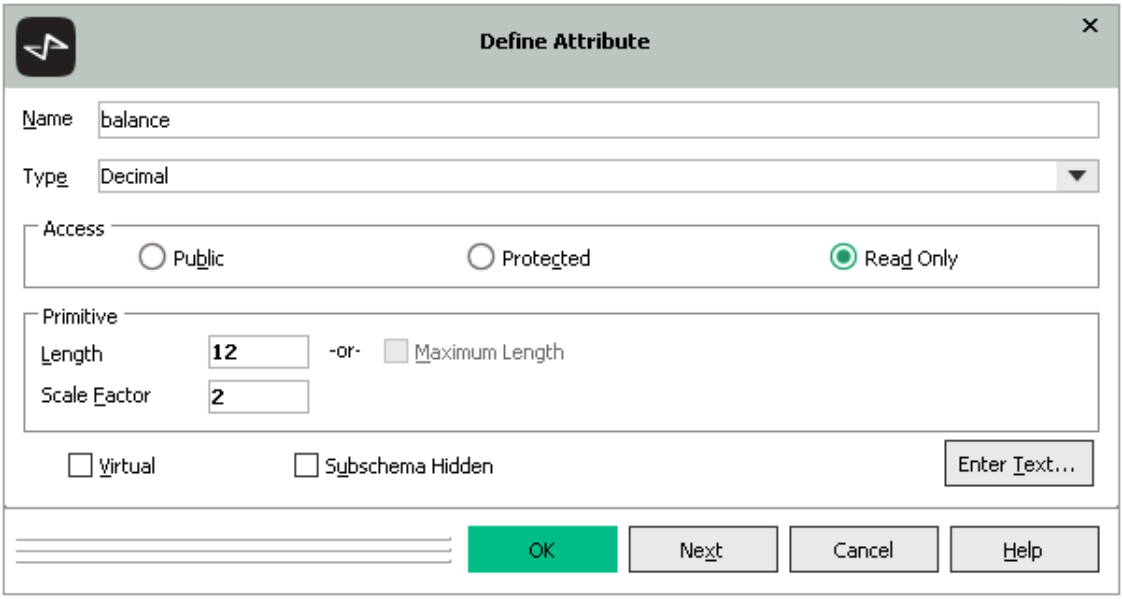
In this exercise, you will add an abstract **BankAccount** class in the **BankingModelSchema**. The properties and methods will be those specified in the UML class diagram under "Introduction", earlier in this module.

1. Select the **BMSModel** class in the Class Browser.
2. Add a subclass to the **BMSModel** class by selecting the Classes menu **Add** command.

3. Enter **BankAccount** as the name of the class, select **bankingmodelschema** as the map file, select the **Abstract** option, and then click the **OK** button.



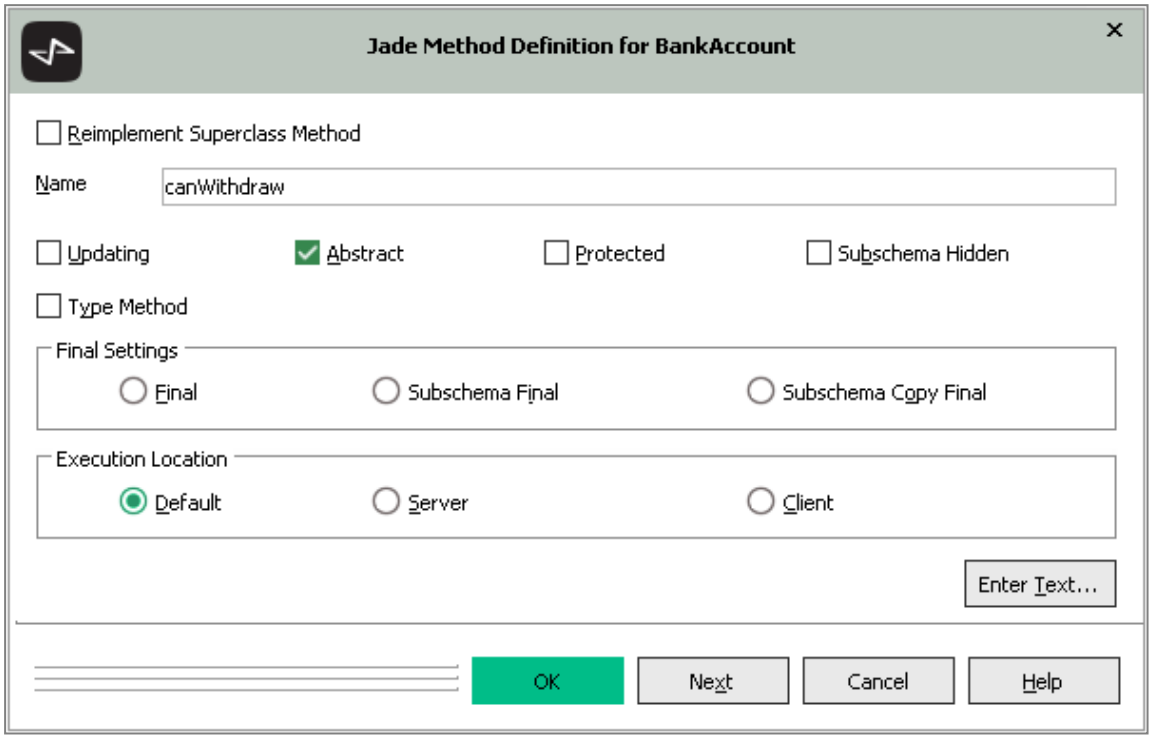
4. Add a read-only **balance** attribute of type **Decimal** with a length (precision) of **12** and a scale factor (number of decimal places) of **2**.



5. Add a read-only **number** attribute of type **Integer**.

6. Add a read-only **myCustomer** reference of type **Customer**.

7. Add a **canWithdraw** method that is **abstract**.



8. Change the signature to include an **amount** parameter and to return a **Boolean** type.

```
canWithdraw( pAmount : Decimal ) : Boolean abstract;
```

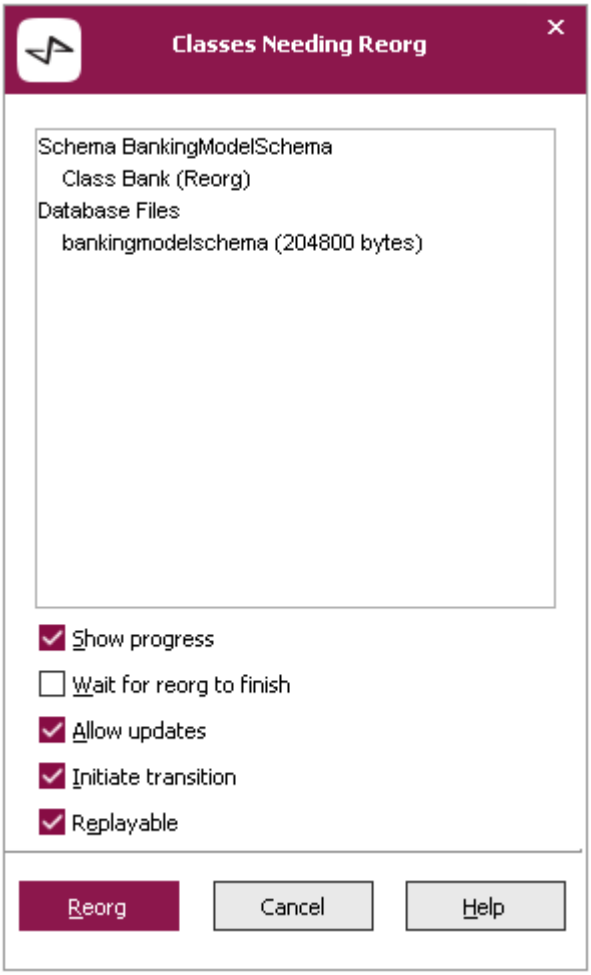
Exercise 8.3 - Changing the *Bank* Class

In this exercise, the **Bank** root object will be changed to store the number used for the most-recently created bank account, in addition to storing the number used for the most-recently created customer. You will also add a method to increment the account number and return the next number to be used.

1. Select the **Bank** class in the Class Browser.
2. Add an attribute called **accountNum** by selecting the Properties menu **Add Attribute** command.

   Select **Integer** as the type, set the access mode to **protected**, and then click the **OK** button.
3. You are warned that a reorganization is required. Click the **Yes** button.
4. The schema is then automatically versioned. Click the **OK** button.
5. Start the reorganization by clicking the **Schema Needs Reorg** toolbar button.

6. Click the **Reorg** button in the Classes Needing Reorg dialog.



7. Add an updating method called **nextAccountNum**, by selecting the Methods menu **New Jade Method** command.

8. Code the method as follows.

```
nextAccountNum() : Integer updating;

begin
    self.accountNum += 1;

    return self.accountNum;
end;
```

9. Compile the method.

Note Possible improvement: the duplication of code in the **nextAccountNum** and **nextCustNum** methods suggests the abstraction of a purpose-built **SequenceNumber** class.

Exercise 8.4 - Implementing the *BankAccount::onCreate* Method

In this exercise, you will add the **onCreate** method to the **BankAccount** class, which will assign a new value to the **number** attribute.

1. Select the **BankAccount** class in the Class Browser.
2. Add a method called **onCreate**.
3. Code the method as follows.

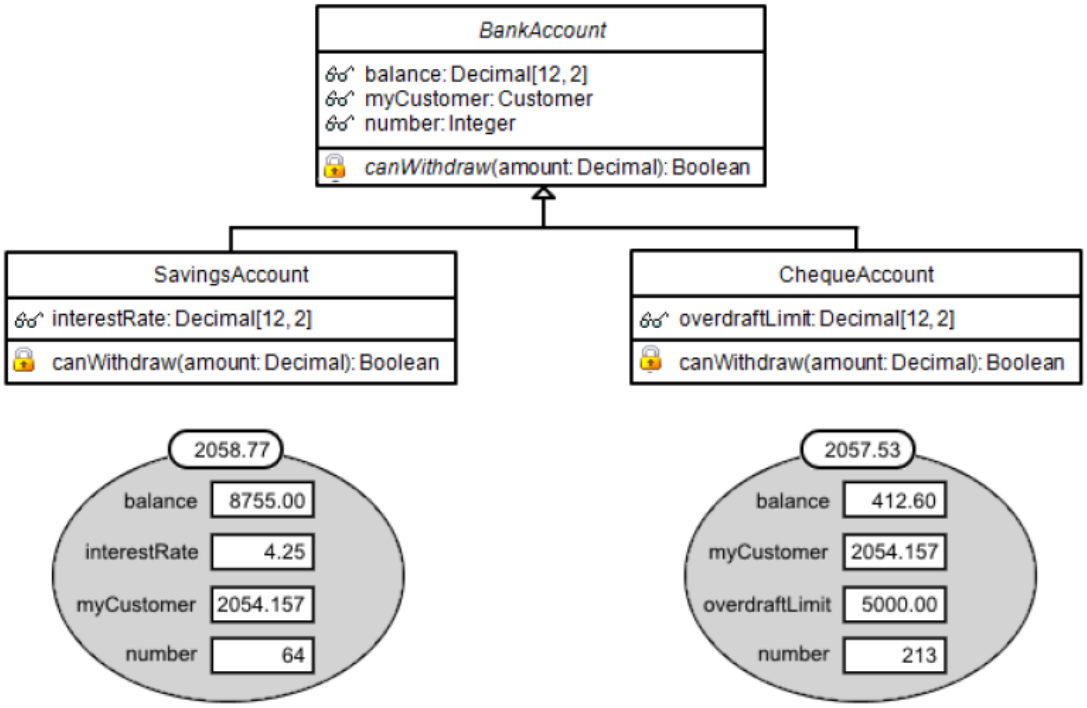
```
onCreate( pTA : BankAccountTA ) updating;

begin
    inheritMethod( pTA );

    self.number := app.myBank.nextAccountNum();

    self.myCustomer := pTA.myCustomer;
end;
```

4. Add a method called **setCommonProperties**.
5. Code the method as follows.

```
setCommonProperties( pTA : BankAccountTA ) updating, protected;

begin
    inheritMethod( pTA );

    self.balance := pTA.balance;
end;
```

Inheritance

Inheritance defines an *is a kind of* hierarchy between classes in which a subclass inherits properties and methods defined in one or more superclasses; for example, in the hierarchy of bank account classes, a **ChequeAccount** object *is a kind of* **BankAccount**. A superclass can be shared by one or more subclasses, but a subclass cannot have more than one superclass.



A subclass inherits all properties and all methods defined in classes above it in the hierarchy. A subclass can reimplement methods defined in a superclass to extend or replace superclass behavior.

Note When you reimplement a method, you can use **inheritMethod** to call the superclass implementation.

Polymorphism

Polymorphism means *many forms*. In the banking system, bank accounts come in many forms: cheque accounts, savings accounts, credit card accounts, and so on. A bank account handles a withdrawal request by calling the **canWithdraw** method, which also comes in many forms. Each **canWithdraw** implementation is specific to the type of bank account.

Using polymorphism, you can code a withdrawal from an Automated Teller Machine (ATM) in a simple way.

At run time, the code that is executed is as follows.

```
// Polymorphic coding
ba.withdraw(amount);
```

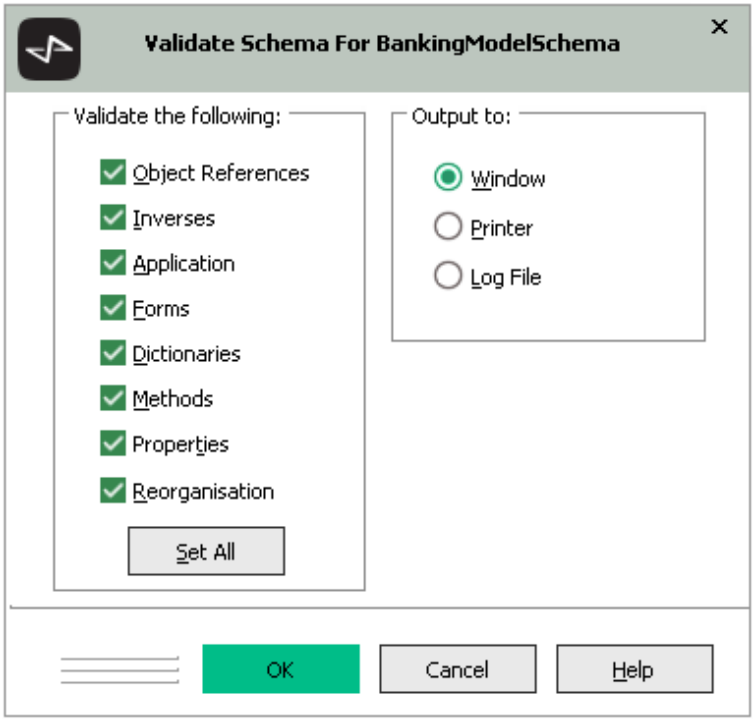
The **ba** variable is of generic type **BankAccount**. At run time, the ATM user selects a cheque account, a savings account, or some other type of bank account and then enters a value for the **amount** parameter.

The important point to notice is the absence of **if** instructions that check for a specific types of bank account. Without polymorphism, the code would be as follows.

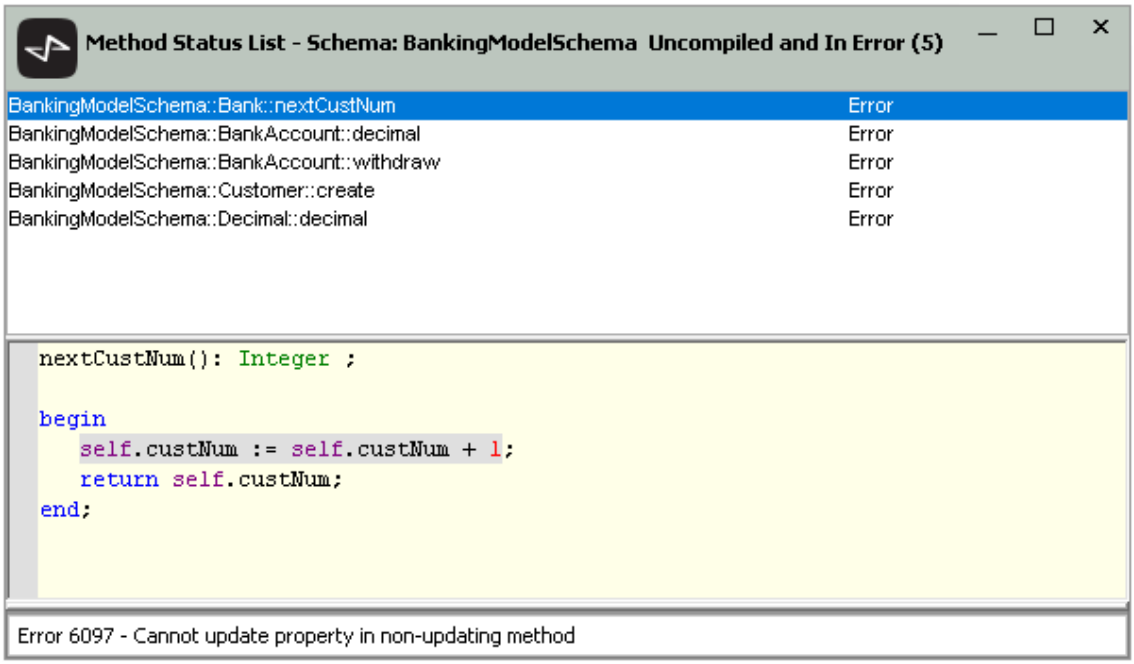
```
// Non-polymorphic coding
if ba.isKindOf(ChequeAccount) then
    // Code for a cheque account
elseif ba.isKindOf(SavingsAccount) then
    // Code for a savings account
endif;
```

Validating a Schema

You can validate many components of a schema, including checking for subclasses where an abstract method has not been implemented, by using the Schema menu **Validate** command.



If you want only to check for methods that are uncompiled and in error, use the Browse menu **Status List** command.

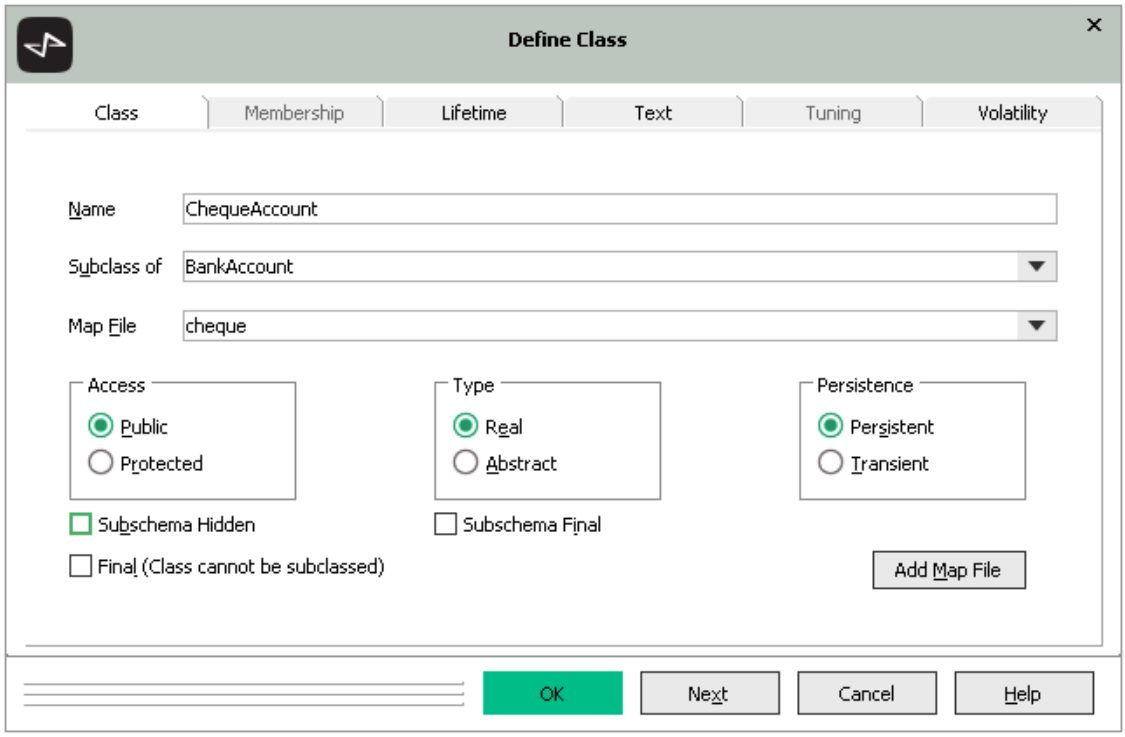


Exercise 8.5 - Adding a ChequeAccount Class

In this exercise, you will add a real class called **ChequeAccount** class, which is a subclass of **BankAccount**. In addition to the properties inherited from **BankAccount**, **ChequeAccount** has an additional **overdraftLimit** property. You will implement the **setCommonProperties** hook method called by the TAF framework to persist the data.

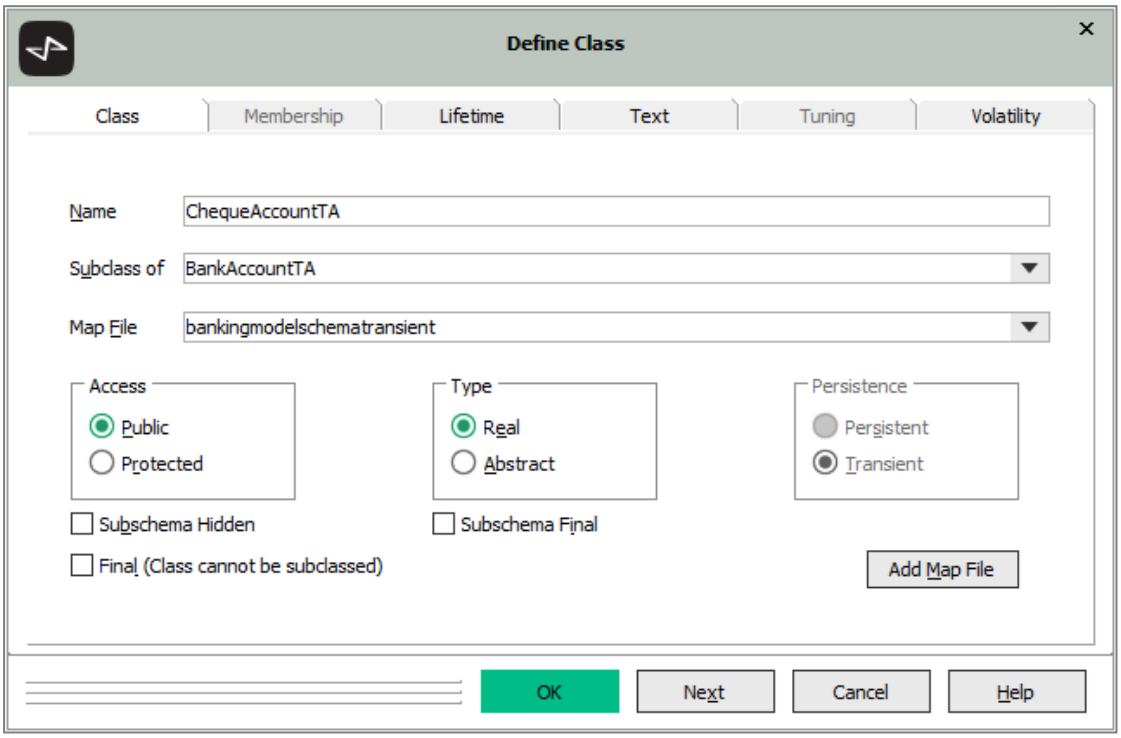
You will reimplement the **canWithdraw** method to allow withdrawals that would not cause the **balance** to exceed the overdraft facility.

1. Select the **BankAccount** class in the Class Browser.
2. Add a subclass to the **BankAccount** class by selecting the Classes menu **Add** command.
3. Enter **ChequeAccount** as the name of the class, select the **cheque** map file, and then click the **OK** button.



4. Select the View menu **Show Inherited** command, to see the properties and methods that are inherited.
5. Add a read-only **overdraftLimit** attribute of type **Decimal** with a length (precision) of **12** and a scale factor (number of decimal places) of **2**.
6. Select the **BankAccountTA** class in the Class Browser.
7. Add a subclass to the **BankAccountTA** class by selecting the Classes menu **Add** command.

8. Enter **ChequeAccountTA** as the name of the class, select the **bankingmodelschematransient** map file, and then click the **OK** button.



9. Add a public **overdraftLimit** attribute of type **Decimal**, length **12**, and scale factor **2**.

10. Select the **CheckAccount** class in the Class Browser.

11. Add the **setCommonProperties** method, by selecting the Methods menu **New Jade Method** command. Click **Yes** on any warning messsage boxes.

12. Code the method as follows.

```
setCommonProperties( pTA : ChequeAccountTA ) updating, protected;

begin
    inheritMethod( pTA );

    self.overdraftLimit := pTA.overdraftLimit;
end;
```

13. Add a **canWithdraw** method. A dialog warns that there is already a method of that name in a superclass. Click the **Yes** button, to continue.

14. Code the method as follows.

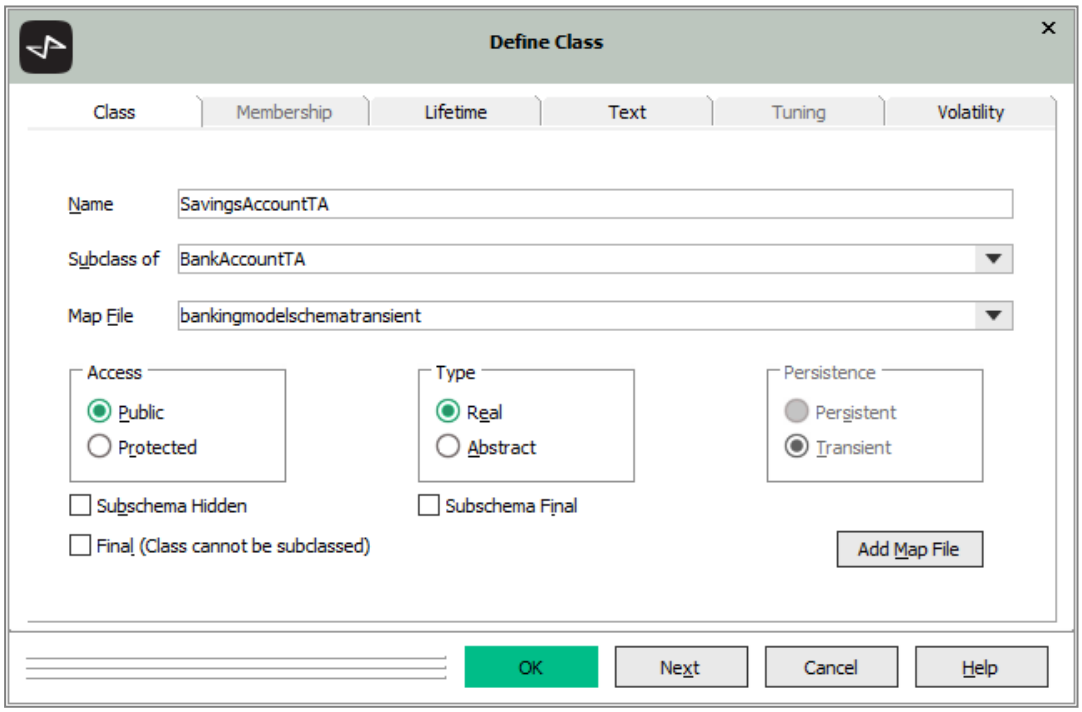
```
canWithdraw( pAmount : Decimal ) : Boolean;

begin
    if pAmount > self.balance + self.overdraftLimit then
        write "Insufficient funds in cheque account";
        return false;
    else
        return true;
    endif;
end;
```

15. Compile the method.

Exercise 8.6 - Adding a *SavingsAccountTA* Class

In this exercise, you will add the transaction agent class for persisting **SavingsAccount** class instances.

1. Select the **BankAccountTA** class in the Class Browser.
2. Add a subclass to the **BankAccountTA** class by selecting the Classes menu **Add** command.
3. Enter **SavingsAccountTA** as the name of the class, selecting the **bankingmodelschematransient** map file, and then click the **OK** button.



4. Add a public **interestRate** attribute of type **Decimal**, length **12**, and scale factor **2**.
5. Add a new method called **initialize**, by selecting the Methods menu **New Jade Method** command.

6. Code the method as follows.

```
initialize() updating;

begin
    inheritMethod();

    self.interestRate := null;
end;
```

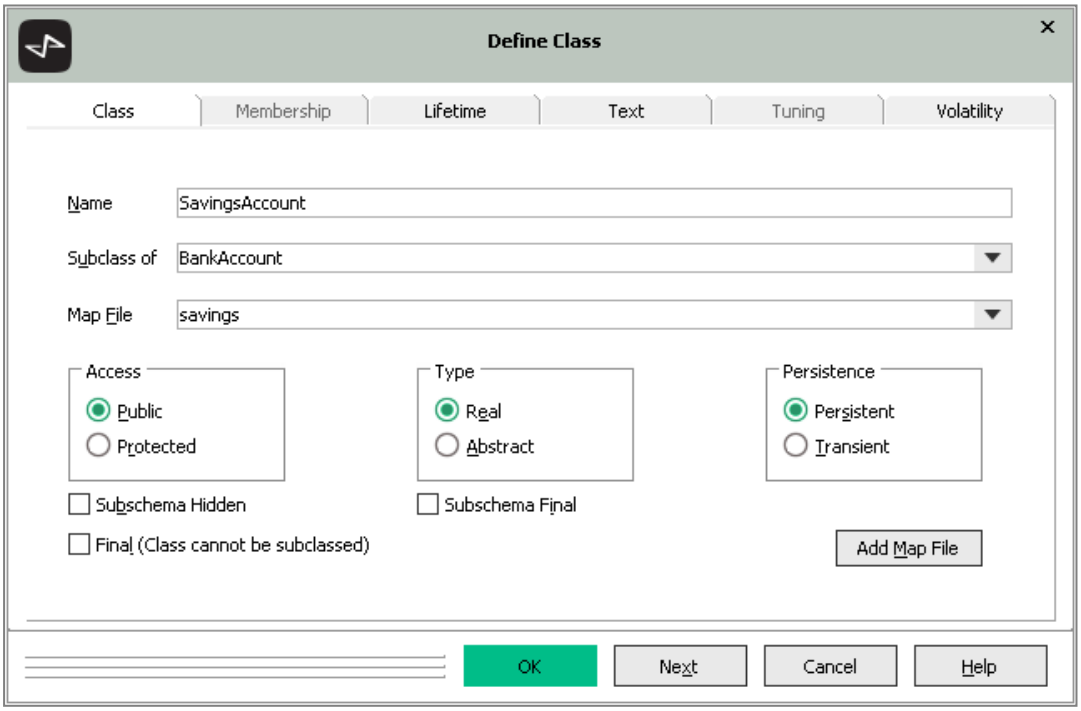
7. Compile the method.

Exercise 8.7 - Adding a *SavingsAccount* Class

In this exercise, you will add a real class called **SavingsAccount**, which is a subclass of **BankAccount**. In addition to the properties inherited from **BankAccount**, **SavingsAccount** has an additional **interestRate** property.

You will reimplement the **canWithdraw** method to allow withdrawals that would not cause the **balance** to become negative.

1. Select the **BankAccount** class in the Class Browser.
2. Add a subclass to the **BankAccount** class by selecting the Classes menu **Add** command.
3. Enter **SavingsAccount** as the name of the class, select the **savings** map file, and then click the **OK** button.



4. Add a read-only **interestRate** attribute of type **Decimal** with a length (precision) of **12** and a scale factor of **2**.
5. Add an updating method called **setCommonProperties**, by selecting the Methods menu **New Jade Method** command.

6. Code the method as follows.

```
setCommonProperties( pTA : SavingsAccountTA ) updating, protected;

begin
    inheritMethod( pTA );

    self.interestRate := pTA.interestRate;
end;
```

7. Add a **canWithdraw** method. A message box warns that there is already a method of that name in a superclass. Click the **Yes** button, to continue.

8. Code the method as follows.

```
canWithdraw( pAmount : Decimal ) : Boolean;

begin
    if pAmount > self.balance then
        write "Insufficient funds in savings account!";
        return false;
    else
        return true;
    endif;
end;
```

9. Compile the method.

Exercise 8.8 - Adding Additional Methods to Transaction Agent Classes

In this exercise, you will add additional methods to Transaction Agent classes.

1. Select the **BankAccountTA** class.

2. Add a method called **getModelObjectClass**.

3. Code the method as follows.

```
getModelObjectClass() : Class protected;

begin
    return BankAccount;
end;
```

4. Compile the method using F8.

5. Select the **ChequeAccountTA** class.

6. Add a method called **getModelObjectClass**.

7. Code the method as follows.

```
getModelObjectClass(): Class protected;

begin
    return ChequeAccount;
end;
```

8. Compile the method.
9. Select the **SavingsAccountTA** class.
10. Add a method called **getModelObjectClass**.
11. Code the method as follows.

```
getModelObjectClass(): Class protected;

begin
    return SavingsAccount;
end;
```

12. Compile the method.

Exercise 8.9 - Creating Bank Accounts with a JadeScript

In this exercise, you will add a **createBankAccounts** JadeScript method to create a cheque account and a savings account.

1. Select the **JadeScript** class in the Class Browser.
2. Add a method called **createBankAccounts**, by selecting the Methods menu **New Jade Method** command.

3. Code the method as follows.

```
createBankAccounts();

vars
    chequeAccountTA : ChequeAccountTA;
    savingsAccountTA : SavingsAccountTA;

begin
    app.initialize();

    beginTransaction;

    create chequeAccountTA transient;
    chequeAccountTA.balance := 0;
    chequeAccountTA.overdraftLimit := 500;
    if not chequeAccountTA.persistEntityInTransState( BMS_Full_update ) then
        write "Errors encountered creating cheque account";
        write chequeAccountTA.getFullErrorDetails();
        abortTransaction;
        return;
    endif;

    create savingsAccountTA transient;
    savingsAccountTA.balance := 100;
    savingsAccountTA.interestRate := 4.5;
    if not savingsAccountTA.persistEntityInTransState( BMS_Full_update ) then
        write "Errors encountered creating savings account";
        write savingsAccountTA.getFullErrorDetails();
        abortTransaction;
        return;
    endif;

    commitTransaction;

epilog
    delete chequeAccountTA;
    delete savingsAccountTA;
end;
```

4. Compile and execute the method.

5. Inspect the cheque account and savings account objects by selecting the **BankAccount** class, and then selecting the Classes menu **Inspect All Instances** command.

Exercise 8.10 - ATM Simulation

In this exercise, you will simulate a withdrawal from an ATM. We will first add some additional methods required by the TAF framework to our persistent **BankAccount** class and both subclasses. We will then write a JadeScript method to simulate the withdrawal.

1. Select the **BankAccount** class in the Class Browser.

2. Add a method called **getTAClass**.

3. Code the method as follows.

```
getTAClass() : Class;

begin
    return BankAccountTA;
end;
```

4. Compile the method.
5. Select the **ChequeAccount** class in the Class Browser.
6. Add a method called **getTAClass**.
7. Code the method as follows.

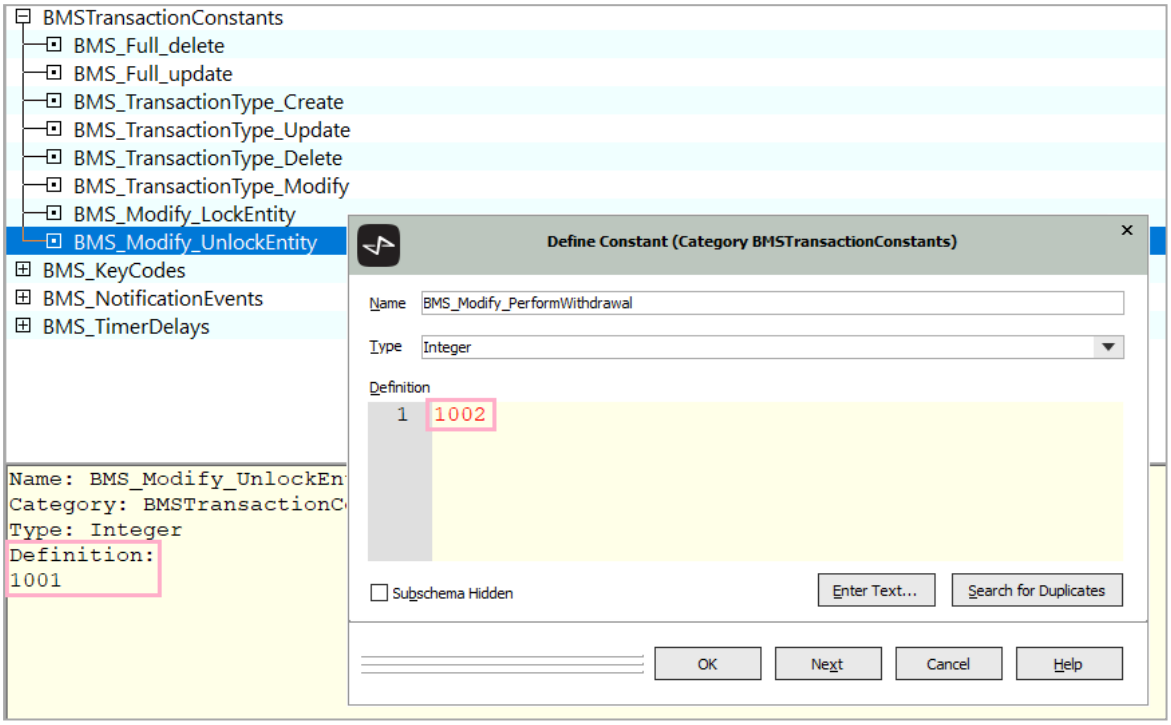
```
getTAClass(): Class;

begin
    return ChequeAccountTA;
end;
```

8. Compile the method.
9. Select the **SavingsAccount** class in the Class Browser.
10. Add a method called **getTAClass**.
11. Code the method as follows.

```
getTAClass(): Class;

begin
    return SavingsAccountTA;
end;
```

12. Compile the method.
13. Press Ctrl+G to open the Global Constants Browser.
14. Select the **BMSTransactionConstants** category.
15. Select the **Sorted by Value** command from the **Category** command in the Global Constants menu.
16. Select the last **BMS_Modify_*xxx*** entry so that you can see the current highest-numbered modification constant value.
17. Add a new global constant called **BMS_Modify_PerformWithdrawal** by selecting the **Add** command from the **Constant** command in the Global Constants menu.

Make it an **Integer** constant with a value that is +1 larger than the current highest-numbered modification constant value.



18. Select the **BankAccountTA** class in the Class Browser.
19. Add a new public attribute **transactionAmount**, type **Decimal**, length **12**, and scale factor **2**.
20. Add a new **doValidate** method.
21. Code the method as follows.

```
doValidate( pOperation : Integer ) : Boolean updating;

begin
    inheritMethod( pOperation );

    if self.modificationCode = BMS_Modify_PerformWithdrawal then
        if not self.getModelObject.canWithdraw( self.transactionAmount ) then
            self.addError(
                    "Insufficient funds to withdraw " & transactionAmount.currencyFormat() &
                    " as the current balance is " & self.getModelObject().balance.currencyFormat(),
                    null
                    );
        endif;
    endif;

    return self.hasNoErrors();
end;
```

22. Add a new method called **populateFromObject**.

23. Code the method as follows.

```
populateFromObject( pBankAccount: BankAccount ) updating;

begin
    inheritMethod( pBankAccount );

    self.balance := pBankAccount.balance;

    self.myCustomer := pBankAccount.myCustomer;
end;
```

24. Compile the method.
25. Select the **ChequeAccountTA** class in the Class Browser.
26. Add a new method called **populateFromObject**.
27. Code the method as follows.

```
populateFromObject( pChequeAccount : ChequeAccount ) updating;

begin
    inheritMethod( pChequeAccount );

    self.overdraftLimit := pChequeAccount.overdraftLimit;
end;
```

28. Compile the method.
29. Select the **SavingsAccountTA** class in the Class Browser.
30. Add a new method called **populateFromObject**.
31. Code the method as follows.

```
populateFromObject( pSavingsAccount : SavingsAccount ) updating;

begin
    inheritMethod( pSavingsAccount );

    self.interestRate := pSavingsAccount.interestRate;
end;
```

32. Compile the method.
33. Select the **BankAccount** class in the Class Browser.
34. Add a new method called **onModify**.

35. Code the method as follows.

```
onModify( pTA : BankAccountTA ) updating;

begin
    if pTA.modificationCode = BMS_Modify_PerformWithdrawal then
        self.balance -= pTA.transactionAmount;
    endif;
end;
```

36. Compile the method.

37. Select the **JadeScript** class in the Class Browser.

38. Add a method called **simulateATM**.

39. Code the method as follows.

```
simulateATM();

vars
    accountType : String;
    bankAccount : BankAccount;
    bankAccountTA : BankAccountTA;
    amount : Decimal[ 12, 2 ];

begin
    // Select account type
    write "Enter 'cheque' or 'savings'";
    read accountType;

    if accountType = "cheque" then
        bankAccount := ChequeAccount.firstInstance();
    elseif accountType = "savings" then
        bankAccount := SavingsAccount.firstInstance();
    else
        write "Invalid account type";
        return;
    endif;

    write "Enter amount to withdraw";
    read amount;

    write "Starting balance of account = " & bankAccount.balance.String;

    create bankAccountTA as bankAccount.getTAClass() transient;
    bankAccountTA.populateFromObject( bankAccount );
    bankAccountTA.transactionAmount := amount;
    if not bankAccountTA.persistEntity( BMS_Modify_PerformWithdrawal ) then
        write "The withdrawal failed with the following errors";
        write bankAccountTA.getFullErrorDetails();
        return;
    endif;

    write "New balance of account = " & bankAccount.balance.String;

epilog
    delete bankAccountTA;
end;
```

40. Run the JadeScript method and then check that the withdrawal limits are being enforced.

Module 9 Collections

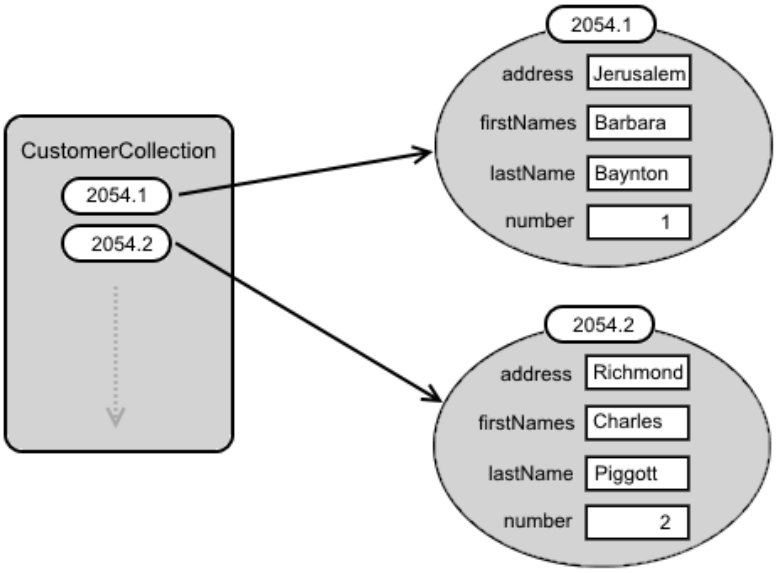
This module contains the following topics.

Introduction

A collection is an object that stores:

- Primitive types (for example, an **IntegerArray** contains a series of integer values)
- References to other objects

Note It does not contain the objects themselves; just references to them.



Types of Collection

The three types of collection are:

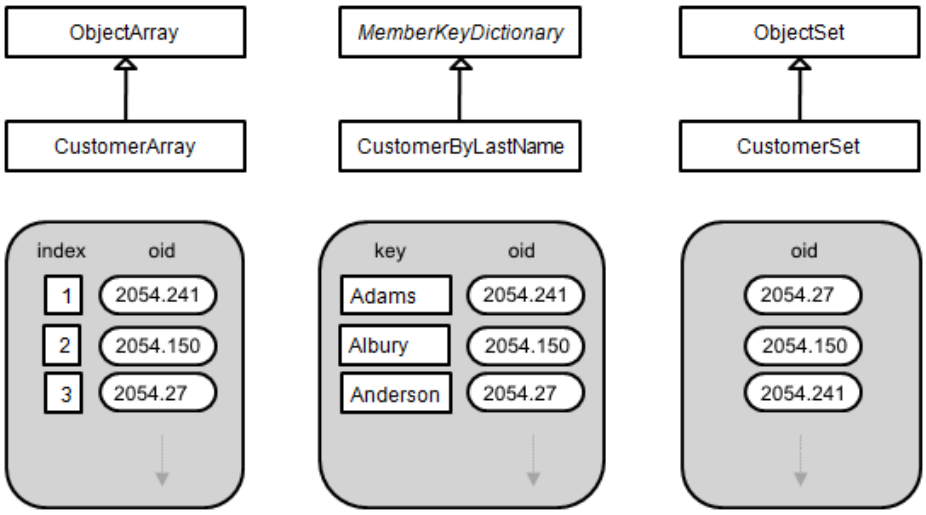
- **Array**, which is a collection of objects or primitive values, ordered by index number. An array can hold the same object or primitive value more than once.
- **Dictionary**, which is a collection of objects ordered by keys that you specify.

  The three types of dictionary are:

  - **MemberKeyDictionary**, whose keys are properties of the member objects
  - **ExtKeyDictionary**, whose keys are specified manually when objects are added
  - **DynaDictionary**, which is a dictionary defined at run time
- **Set**, which is a collection of objects conceptually unordered (in practice, ordered by OID).

Adding a Collection Class

Collection classes are added as subclasses of collection classes in **RootSchema**.



The new subclass inherits the methods of the superclass.

In addition to naming the collection, you must specify the membership class (the class that supplies objects to the collection), and for a dictionary, you must specify the keys.

Collection Methods

The following methods are defined for the abstract **Collection** class in **RootSchema**. Methods are reimplemented in the different **Collection** subclasses.

Method	Example
size	`// Number of entries in the collection` `size := coll.size();`
first	`// First entry in the collection` `cust := coll.first();`
last	`// Last entry in the collection` `cust := coll.last();`

copy	`// Entries from one collection (coll1) copied to another (coll2)`
	`// Entries must meet membership criteria of target collection`
	`coll1.copy(coll2);`

clear	`// Objects are removed from collection, but objects not deleted`
	`// An empty collection remains`
	`coll.clear();`

purge	`// Objects are removed from collection, and objects are deleted`
	`// An empty collection remains`
	`coll.purge();`

Method	Example
add	`// Object added to end array or correct place in set or dictionary` `coll.add(cust);`

tryAdd	`// Object added to end array or correct place in set or dictionary`
	`// UNLESS that object already exists in the collection`
	`coll.tryadd(cust);`

remove	`// First reference to cust removed from collection`
	`// Exception raised if cust not in collection`
	`coll.remove(cust);`

tryRemove	`// First reference to cust removed from collection`
	`// Returns false if cust not in collection`
	`coll.tryRemove(cust);`

includes	`// Checks whether cust is already in collection`
	`if not coll.includes(cust) then`
	`  coll.add(cust);`
	`endif;`

createIterator	`// Iterator created for collection`
	`// Iterator can move forwards or backwards through collection`
	`iter := coll.createIterator();`

Dictionaries

Dictionaries store objects in the order specified by the keys; for example, the customers in a **CustomerByLastNameDict** collection are ordered alphabetically by last name.

You can retrieve an object from a dictionary by using the **getAtKey** method. In the following example, **dict** is a **CustomerByLastNameDict** collection containing the customers from the **Customers.txt** file.

```
cust := dict.getAtKey("Baynton");  // Retrieves customer with key value "Baynton"
```

You can use the equivalent square brackets notation.

```
cust := dict["Baynton"];           // Equivalent square bracket notation
```

Arrays

Arrays store objects in index order, and you can access an object using its index. In the following examples, **array** is a **CustomerArray** collection containing the customers from the **Customers.txt** file.

```
cust := array[207];// Retrieves the 207th customer from the array
```

```
array[1000] := cust;// Puts cust into the array at position 1000
```

In the second example, if the array contained fewer than 1,000 entries before the instruction is executed, it is expanded with null entries up to that size.

Methods are available for inserting and removing objects into an array. When these methods are executed, the other entries in the array are moved up or down automatically.

You can use array index values to move through an array, but it is more efficient to use an iterator. Indexing on large arrays is slow, and degrades with size.

Exercise 9.1 - Adding a Customer Dictionary

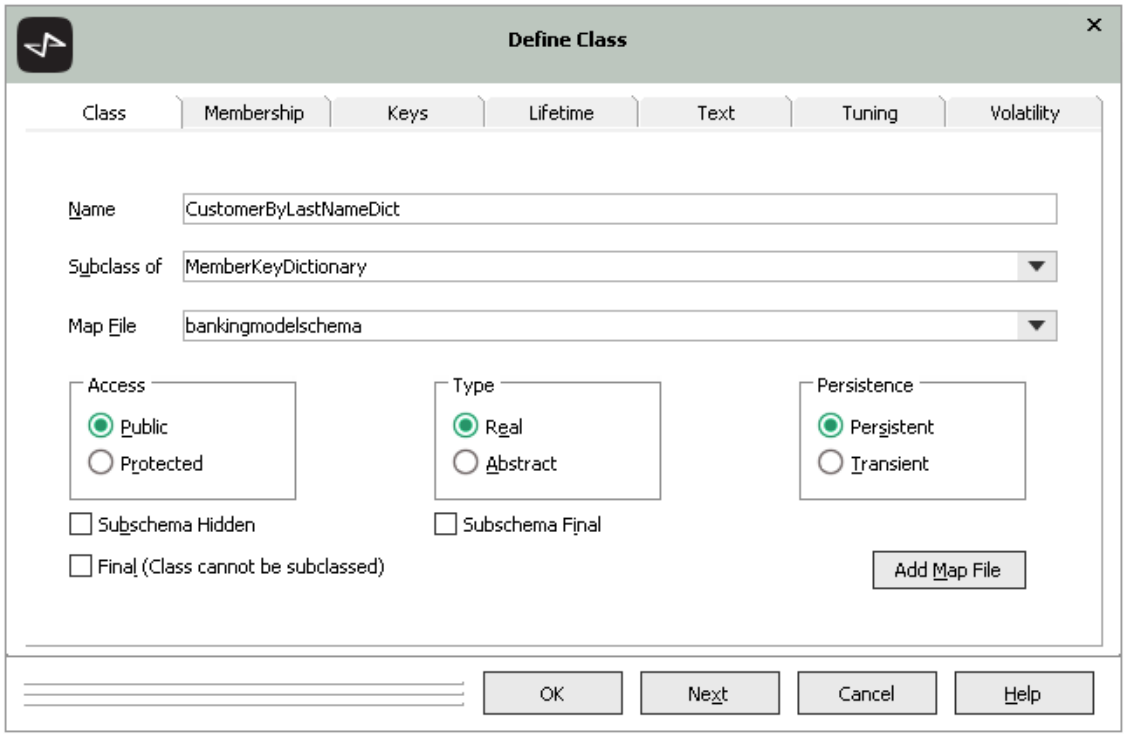
In this exercise, you will add a **CustomerByLastNameDict** dictionary.

1. Find the **MemberKeyDictionary** class.

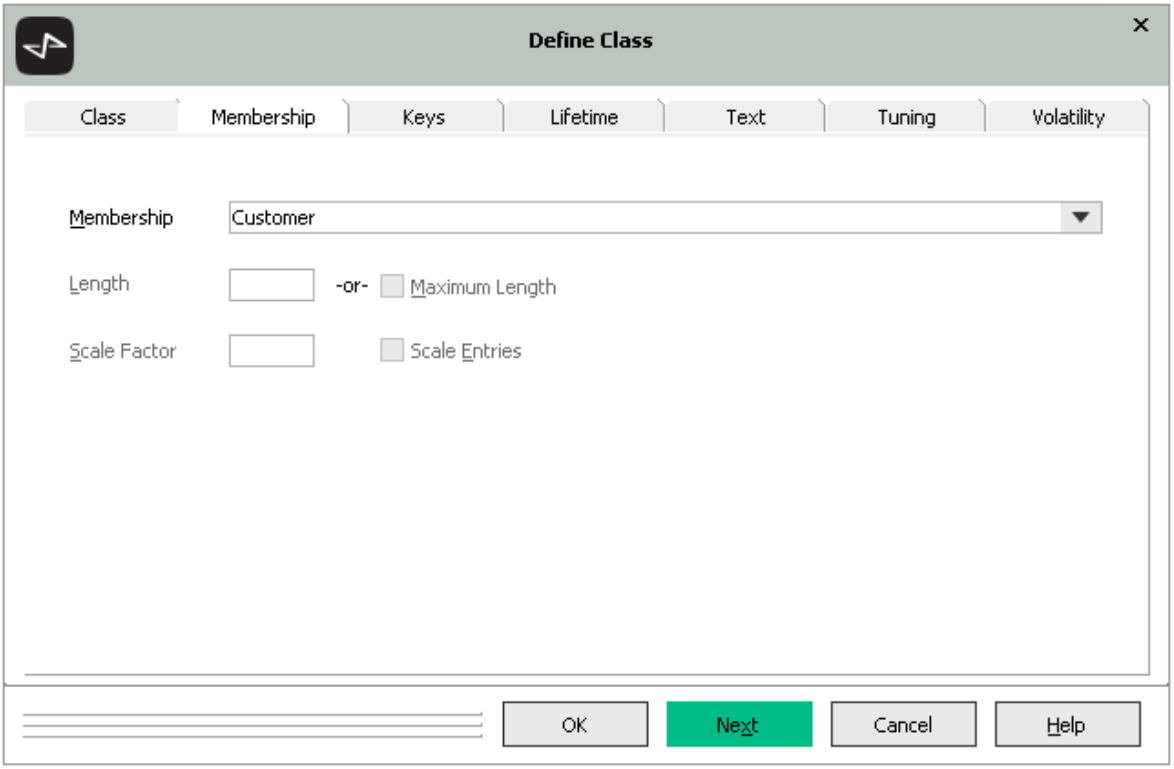
   Tip When you use the Find Type dialog, which is opened with the F4 keyboard shortcut, you can enter the initials rather than the full name of a type; for example, **MKD** will find the **MemberKeyDictionary** class.

2. Add a subclass to the **MemberKeyDictionary** class by selecting the Classes menu **Add** command.

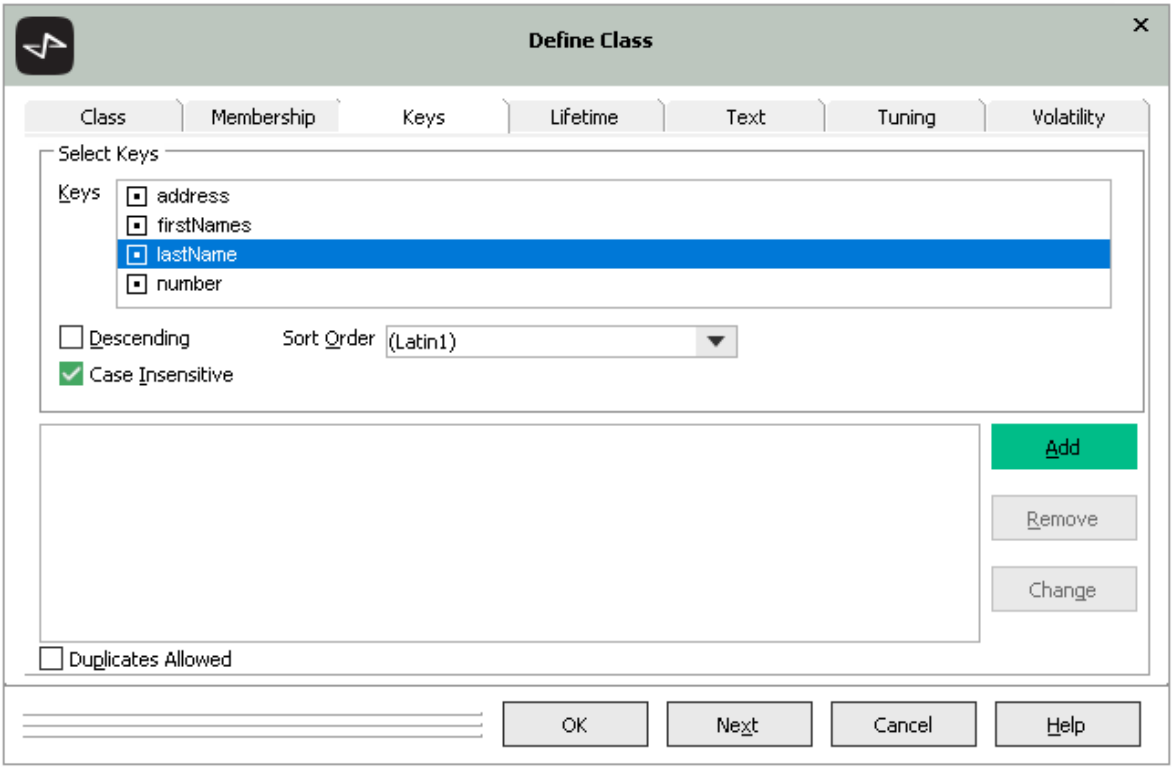
3. On the **Class** sheet, enter **CustomerByLastNameDict** as the name of the class, and then select the **Membership** sheet.



4. On the **Membership** sheet, select **Customer** as the **Membership** class, and then select the **Keys** sheet.



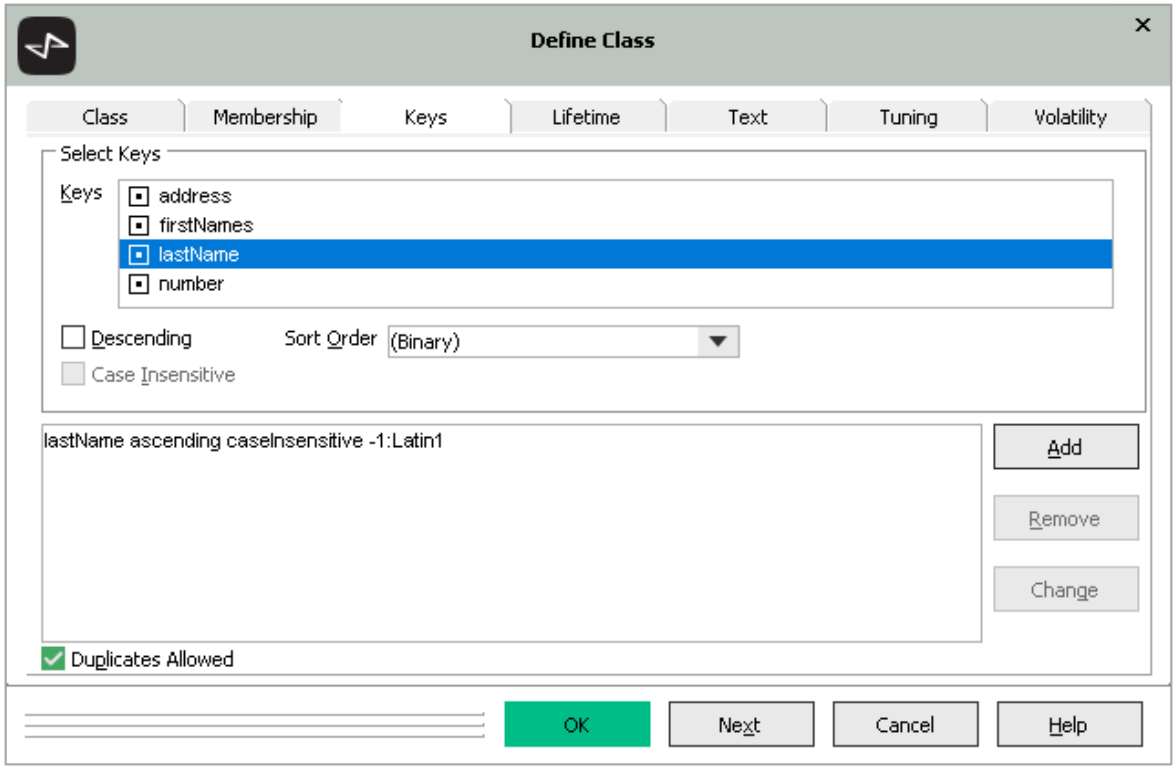
5. On the **Keys** sheet, select **lastName** as the key, select **Latin1** as the sort order, check the **Case Insensitive** check box, and then click the **Add** button.



Tips **Latin1** is a standard ISO ordering sequence suitable for many alphabets.

Case-insensitive ordering enables customer searches without entering uppercase and lowercase exactly.

6. Check the **Duplicates Allowed** check box and then click the **OK** button.



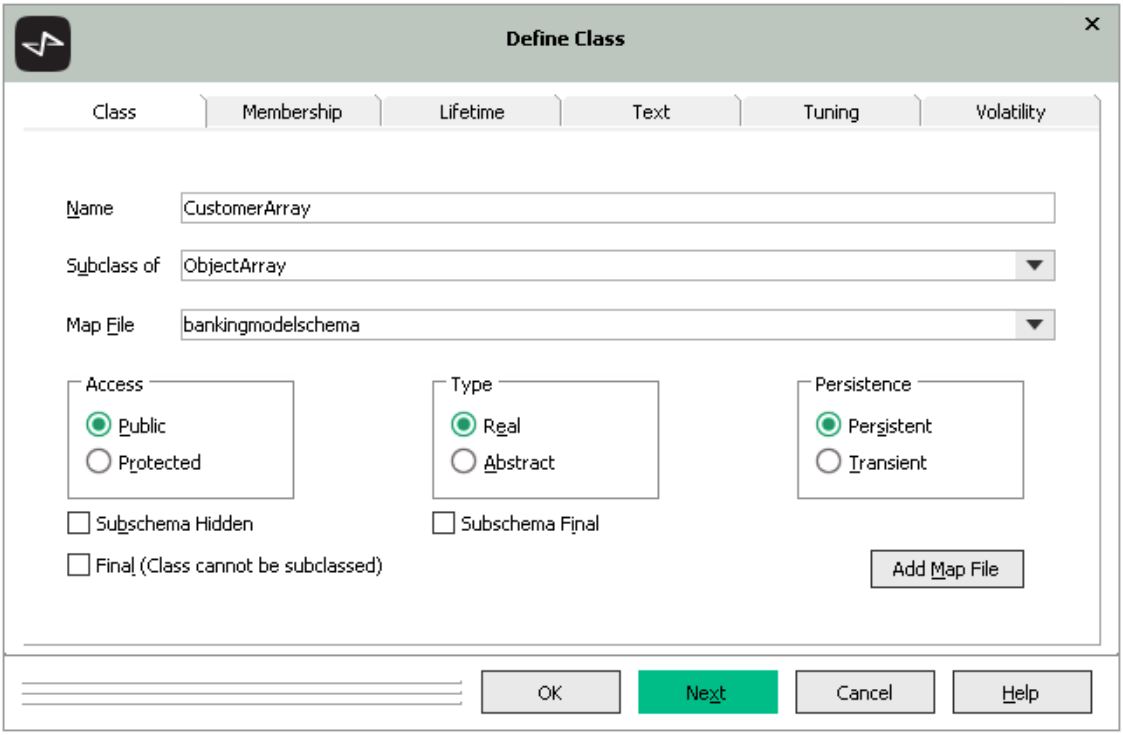
Tip Check the **Duplicates Allowed** check box if the selected keys are likely not to be unique.

Exercise 9.2 - Adding a Customer Array

In this exercise, you will add a **CustomerArray** class.

1. Find the **ObjectArray** class.
2. Add a subclass to the **ObjectArray** class by selecting the Classes menu **Add** command.

3. On the **Class** sheet, enter **CustomerArray** as the name of the class, and then select the **Membership** sheet.



4. On the **Membership** sheet, select **Customer** as the **Membership** class, and then click the **OK** button.

Exercise 9.3 - Removing Test Objects

In this exercise, you will enhance the **removeTestData** method to remove all of the test data that you have created.

1. Select the **JadeScript** class in the Class Browser.
2. Change the **removeTestData** method, as follows.

```
removeTestData();

begin
    beginTransaction;
    Bank.instances.purge();
    ChequeAccount.instances.purge();
    Customer.instances.purge();
    CustomerArray.instances.purge();
    CustomerByLastNameDict.instances.purge();
    SavingsAccount.instances.purge();
    commitTransaction;
end;
```

3. Execute the method.

Exercise 9.4 - Adding Some More Required TAF Methods

In this exercise, you will add some more required methods to Transaction Agent Framework classes.

1. Select the **BankAccountTA** class in the Class Browser.
2. Add a new method called **getModelObject**.
3. Code the method as follows.

```
getModelObject() : BankAccount;

begin
    return inheritMethod().BankAccount;
end;
```

4. Compile the method using F8.
5. Select the **ChequeAccountTA** class in the Class Browser.
6. Add a new method called **getModelObject**.
7. Code the method as follows.

```
getModelObject(): ChequeAccount;

begin
    return inheritMethod().ChequeAccount;
end;
```

8. Compile the method using F8.
9. Select the **SavingsAccountTA** class in the Class Browser.
10. Add a new method called **getModelObject**.
11. Code the method as follows.

```
getModelObject(): SavingsAccount;

begin
    return inheritMethod().SavingsAccount;
end;
```

12. Compile the method using F8.
13. Select the **CustomerTA** class in the Class Browser.
14. Add a new method called **getModelObject**.

15. Code the method as follows.

```
getModelObject() : Customer;

begin
    return inheritMethod().Customer;
end;
```

16. Compile the method using F8.

Exercise 9.5 - Populating a Collection

In this exercise, you will use the data in the **Customers.txt** file to create hundreds of customers and add the customers to a collection.

1. Change the **createCustomersFromFile** JadeScript method as follows.

```
createCustomersFromFile();

vars
    fileOpenDialog : CMDFileOpen;
    inputFile : File;
    nextLineInFile : String;
    customerTA : CustomerTA;
    dict : CustomerByLastNameDict;

begin
    app.initialize();

    create fileOpenDialog transient;
    if fileOpenDialog.open() <> 0 then
        // Exit early, as the user did not select a file
        return;
    endif;

    create customerTA transient;
    create inputFile transient;

    // inputFile.fileName := "c:\JadeCourse\Files\Customers.txt";
    inputFile.fileName := fileOpenDialog.fileName;
    inputFile.mode := File.Mode_Input;
    inputFile.open();

    beginTransaction;
    create dict persistent;
    while not inputFile.endOfFile() do
        nextLineInFile := inputFile.readLine();
        customerTA.initialize();
        customerTA.address := nextLineInFile[ 41 : end ];
        customerTA.firstNames := nextLineInFile[ 16 : 25 ];
        customerTA.lastName := nextLineInFile[ 1 : 15 ];
        customerTA.persistEntityInTransState( BMS_Full_update );
        dict.add( customerTA.getModelObject() );
    endwhile;
    commitTransaction;

    inputFile.close();

epilog
    delete fileOpenDialog;
    delete inputFile;
    delete customerTA;
end;
```

2. Execute the method and then inspect the instance of **CustomerByLastNameDict** that is created.

In this method:

- A persistent instance of **CustomerByLastNameDict** is created.
- The **add** method is used to add each customer to the collection.

foreach with Collections

The **foreach** instruction provides a simple way to iterate any type of collection; that is, process all of the objects in the collection.

```
foreach cust in coll do
    write cust.lastName;
endforeach;
```

The objects are processed in the order in which they are encountered in the collection, unless you add the **reversed** option to work through the objects backwards, starting at the end of the collection.

```
foreach cust in coll reversed do
    write cust.lastName;
endforeach;
```

As you will learn in the module on locking later in this course, the **foreach** instruction places a shared lock on the collection for the duration of the iteration. The shared lock prevents other processes from adding or removing objects from the collection. The purpose of the lock is to iterate the latest edition of the collection without it being changed. However, if you do not want the collection locked, you can use the **discreteLock** option.

```
foreach cust in coll discreteLock do
    write cust.lastName;
endforeach;
```

The **where** clause enables you to be selective about which objects in the collection are processed. In the following example, only the customers from **Richmond** are displayed.

```
foreach cust in coll where cust.address = "Richmond" do
    write cust.lastName;
endforeach;
```

The **foreach** instruction is optimized for dictionaries, with a single key if there is a simple condition based on that key. In the following example, the iteration starts with the first customer with a last name of **Jones**, if there is one.

```
foreach cust in dict where cust.lastName >= "Jones" do
    write cust.lastName;
endforeach;
```

Iterators and Collections

An iterator is an object that can retrieve the next or previous object in a collection. You create an instance of the **Iterator** class and associate it with a collection before the iteration starts.

Note You should delete the iterator when it is no longer needed.

The **createIterator** method of a collection creates an iterator of the correct type and associates it with a collection.

The **next** or **back** methods traverse the collection in a forwards or backwards direction. The methods return **true** if they find the next (or previous) object in the collection, and place a reference to that object in the method's output parameter. When the iterator reaches the end (or the beginning) of the collection, the methods return **false**.

```
iter := coll.createIterator();
while iter.next(cust) do
    write cust.lastName;
endwhile;
delete iter;
```

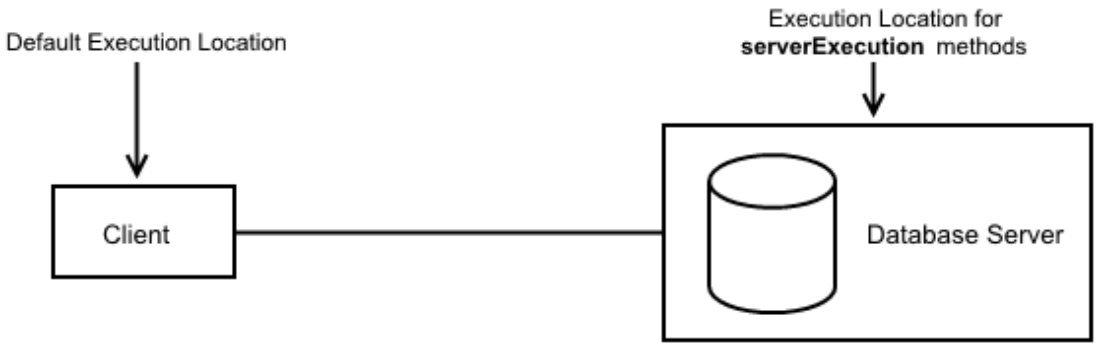
For a dictionary, you can set the start position for iteration by using one of the **startKey** family of methods.

```
iter := coll.createIterator();
coll.startKeyGeq("Jones", iter);
while iter.next(cust) do
    write cust.lastName;
endwhile;
delete iter;
```

An iterator takes a *snapshot* of a collection; that is, it reads a batch of entries from the collection. When an iterator performs its first **next** or **back** call, or when it has exhausted its current entries, it sends a message to the collection to retrieve the next *snapshot*. At this point, a shared lock is acquired on the collection for the time it takes to fetch the next set of entries.

Execution Location

The majority of application code is executed in the client nodes. However, there are situations where it makes sense to switch the execution location of a method to the database server; for example, a method working with a large collection of objects.



You can switch the execution location to the database server by adding the **serverExecution** option to the signature of the method.

```
calledMethod01(parameters): returnType serverExecution;
```

If the **serverExecution** method calls another method, that method will also execute on the database server unless it has the **clientExecution** method option.

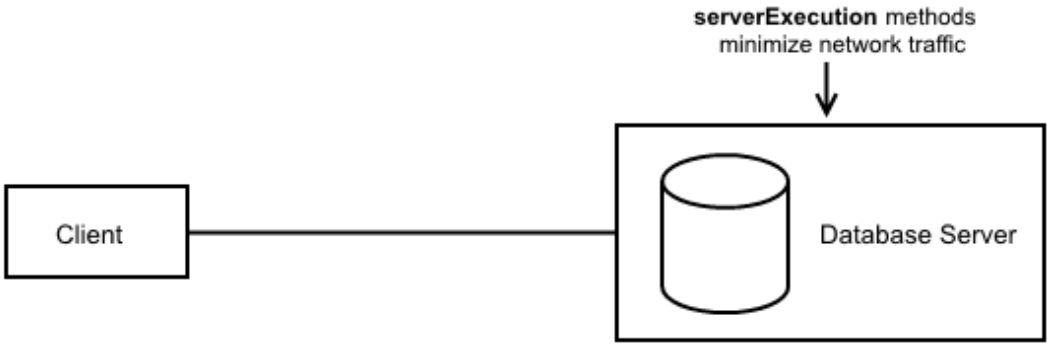
```
calledMethod02(parameters): returnType clientExecution;
```

When a **serverExecution** or **clientExecution** method returns (that is, it completes execution), the calling method resumes executing in the node where it started.

Note You cannot debug server execution methods.

A good case for using a **serverExecution** method would be a method that needs to filter a large collection of objects to produce a smaller collection of objects to be processed.

The filtering could be done on the database server, with the subsequent processing being done on the client.



Note When you execute methods in single user mode, the **serverExecution** and **clientExecution** options have no effect.

Exercise 9.6 - Deleting the *J* Customers

In this exercise, you will use a **foreach** instruction to delete the customers whose last name begins with the letter **J** and report the number of customers deleted. You will use the collection you created in a previous exercise.

Notes Jade methods usually use a camel case naming convention, where each "word" in the name begins with a capital letter except for the first. This is only a convention, and the following method gives an example of an alternative naming convention, snake case, where each "word" in the name is separated by an underscore.

AutoComplete functionality works better with camel case names than with snake case names. For example, if you had a method called **theBestMethodEver**, you could type **tBME** into an editor and it would AutoComplete to it. This is not possible with snake case unless you also uppercase each word.

1. Create a **JadeScript** method called **delete_j_customers**, and code it as follows.

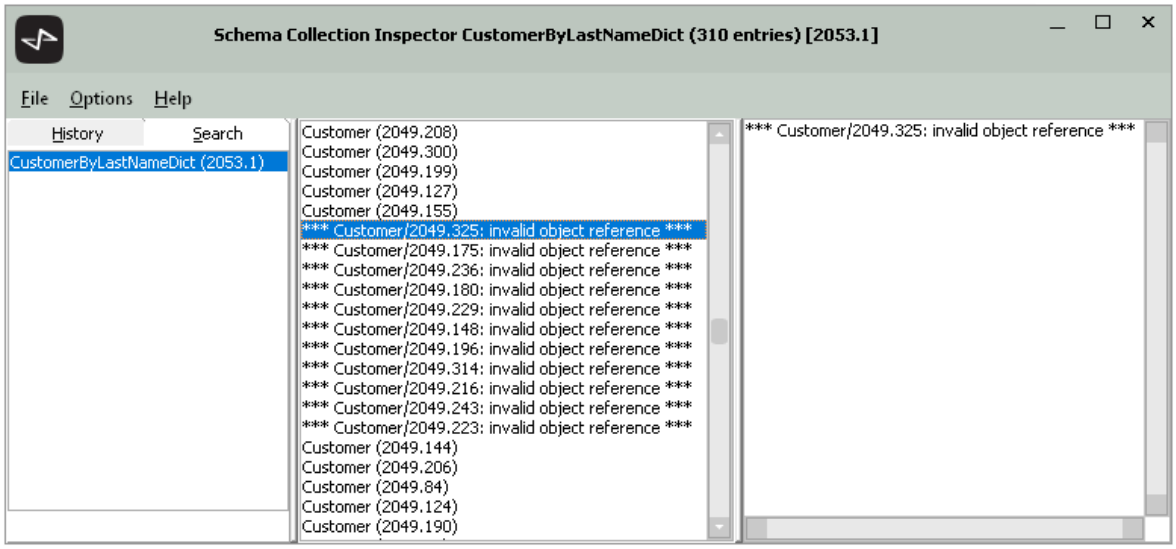
```
delete_j_customers();

vars
    dict : CustomerByLastNameDict;
    cust : Customer;
    i : Integer;

begin
    dict := CustomerByLastNameDict.firstInstance();

    beginTransaction;
    foreach cust in dict where cust.lastName[ 1 ] >= "J" do
        if cust.lastName[ 1 ] >= "K" then
            break;
        endif;
        delete cust;
        i += 1;
    endforeach;
    commitTransaction;
    write i.String & " customers deleted";
end;
```

   In this method:

   - The **firstInstance** method is used to identify the **CustomerByLastNameDict** collection to be iterated.
   - The **where** clause is used to optimize the iteration by starting with the first **J** customer in the collection.
   - The **break** instruction is used to exit from the loop after processing the **J** customers.
   - A counter variable is incremented inside the **foreach** loop.
   - The **delete** instruction is used to delete an object.

2. Execute the method.
3. Inspect the **CustomerByLastNameDict** dictionary.

If you scroll down to the customers whose name should begin with the letter **J**, the inspector window shows a number of *invalid object references*. Can you explain why this has happened?



Note In a later module, you will learn how to avoid having invalid object references in a collection.

Exercise 9.7 - Filtering a Collection

In this exercise, you will create a JadeScript method to filter the **CustomerByLastNameDict** collection. The method executes on the database server and returns a much smaller transient instance of **CustomerArray** for use by the client. The condition for inclusion in the array is that the customer exists and lives in **Richmond**.

1. Select the **JadeScript** class in the Class Browser.
2. Create a method called **filter_Richmond_customers**, as follows.

```
filter_Richmond_customers( pFilteredCustomers : CustomerArray input ) serverExecution;

vars
    customersByLastName : CustomerByLastNameDict;
    customer : Customer;

begin
    customersByLastName := CustomerByLastNameDict.firstInstance();

    foreach customer in customersByLastName where app.isValidObject( customer ) and customer.address = "Richmond" do
        pFilteredCustomers.add( customer );
    endforeach;
end;
```

3. Create a method called **getFilteredCustomers**, as follows.

```
getFilteredCustomers();

vars
    filteredCustomers : CustomerArray;

begin
    create filteredCustomers transient;

    self.filter_Richmond_customers( filteredCustomers );

    write CustomerByLastNameDict.firstInstance().size();

    write filteredCustomers.size();

epilog
    delete filteredCustomers;
end;
```

4. Execute the method.

In the **filter_Richmond_customers** method:

- The **firstInstance** method is used to identify the **CustomerByLastNameDict** collection to be iterated.
- The **where** clause filters the collection by processing only customers who live in **Richmond**.
- The **isValidObject** method of the **Application** class is used to test whether the customer exists. (Remember that there are a number of invalid object references in the collection.)

In the **getFilteredCustomers** method:

- The transient **CustomerArray** object is created. This empty collection is passed to the **filter_Richmond_customers** method for filling.
- The **size** method demonstrates the reduced subset of objects that are to be processed on the client.
- The transient **CustomerArray** object is deleted in the epilog.

Tip It is important to delete transient objects when you have finished with them. To make this easy to remember, a good rule of thumb is that any transient object should be deleted in the same method in which it is created. This is why we pass it as an input parameter to **filter_Richmond_customers** rather than creating it in **filter_Richmond_customers** and returning it as the return value.

Module 10 Relationships

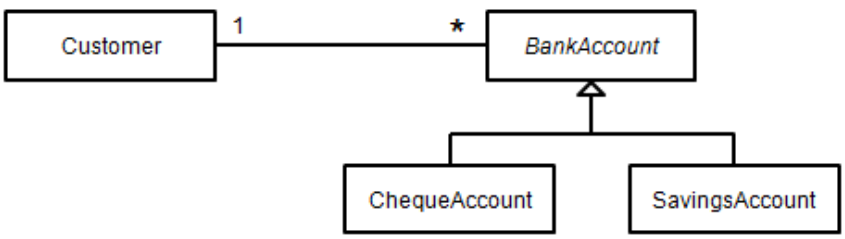
This module contains the following topics.

Introduction

Object-oriented analysis for the banking system uncovers a one-to-many relationship between the **Customer** and **BankAccount** classes.

- *One customer has many bank accounts*. The one-to-many relationship is the most common type.

  The accounts can be cheque accounts, savings accounts, or other types that are added to the hierarchy later.

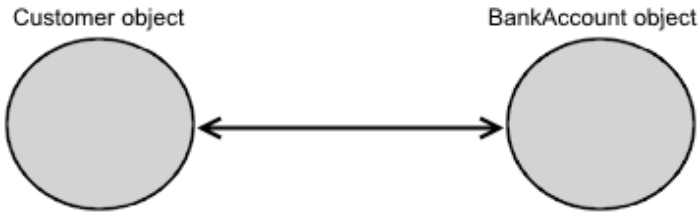


Relationships between classes are implemented using references. References enable you to:

- Navigate from one object to an associated object
- Send a message to an associated object (that is, call a method on the object)

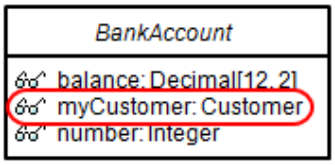
You have already used a reference to navigate from the **app** object to the **Bank** root object.

The one-to-many relationship enables navigation from a customer object to a bank account owned by the customer, and in the other direction.



myCustomer Reference

In an earlier module, you added a **myCustomer** reference to the owner of the bank account in the **BankAccount** class.

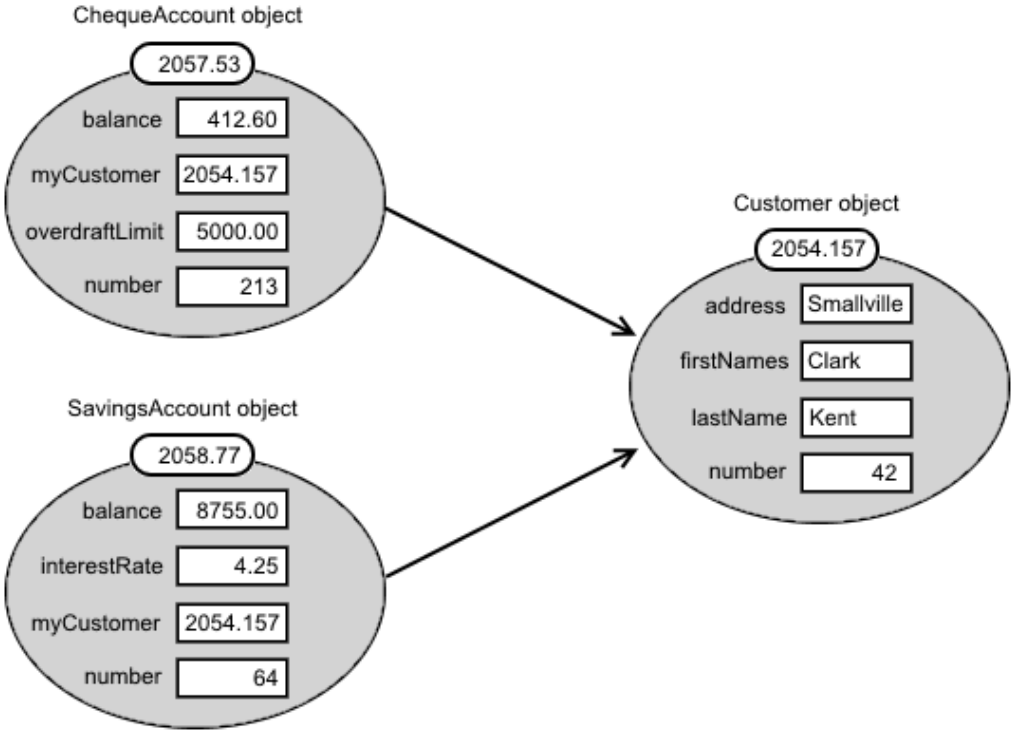


By convention, a reference name starting with **my** is a reference to a single object. In this case, the **BankAccount** object references the **Customer** object who owns the bank account. When a customer is created, the **myCustomer** reference is null.

The **create** method is used to set the initial balance, the overdraft facility, and to associate the bank account with its owner, as follows.

```
create(bal, od: Decimal; cust: Customer) updating;

begin
    self.balance := bal;
    self.overdraftLimit := od;
    self.myCustomer := cust;
end;
```

The following diagram shows two bank account objects that have the same **myCustomer** reference, and therefore belong to the same customer.

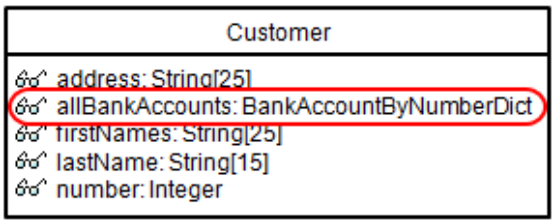


The **myCustomer** reference enables you to navigate from a bank account to the customer who owns the bank account.

In the following sections, you will add an *inverse reference* so that you can navigate from a customer to his or her bank accounts. This will be implemented by a customer having a collection that can contain any number of bank accounts. Consequently, the first step is to define a **BankAccount** collection class.

Exclusive Collections

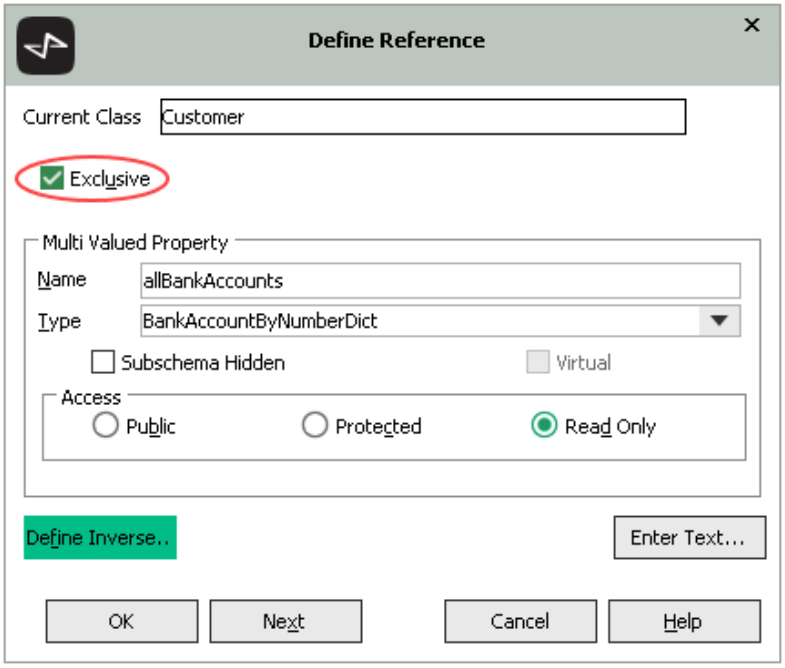
An exclusive collection is one that belongs exclusively to a parent object. Conceptually, the exclusive collection is created when the parent object is created, and deleted when the parent object is deleted. A customer can have any number of bank accounts of different types. This can be implemented by a **Customer** object having an exclusive **BankAccountByNumberDict** collection called **allBankAccounts**. The name **allBankAccounts** should be interpreted as all of the bank accounts owned by the customer; not all of the bank accounts in the system.



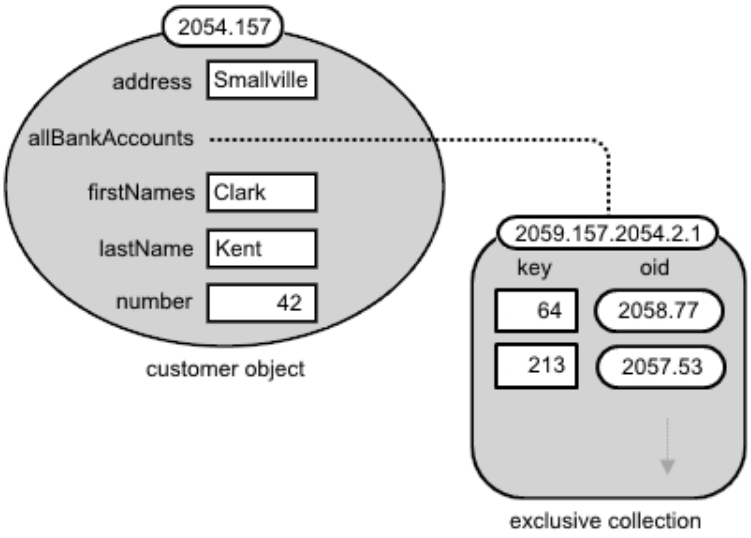
The naming convention used in this course is as follows.

- Start the name of a reference to a single object with **my**
- Start the name of a reference to a collection of objects with **all**

When you add the collection reference, the **Exclusive** check box is checked by default.



An exclusive collection is a subobject (that is, a separate object). No space is allocated in the parent **Customer** object.

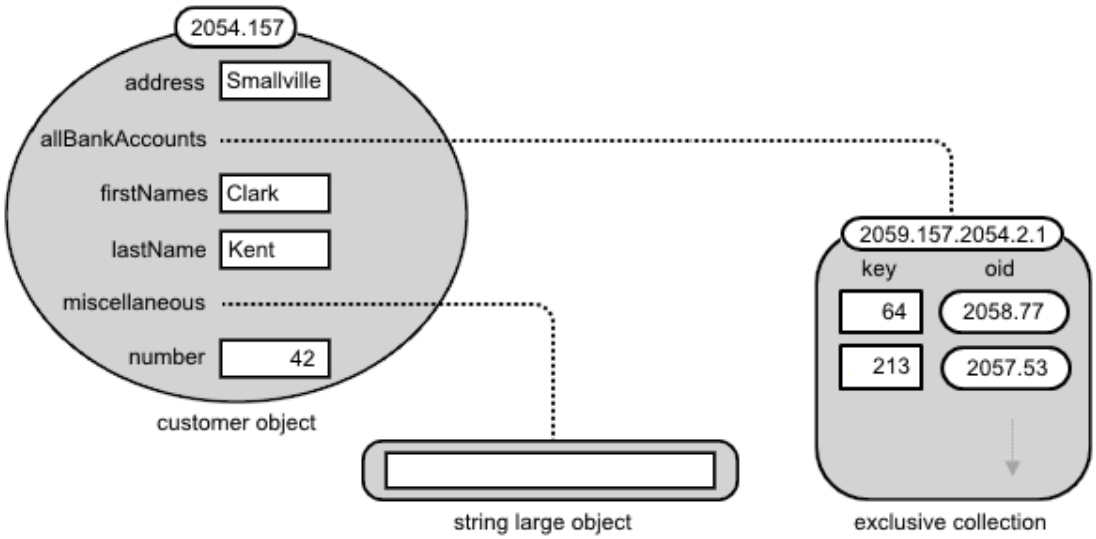


Other Subobjects

When you define a string attribute with a length with fewer than 540 characters, the attribute is embedded in the object; that is, space is allocated in the object to store the attribute value.

If the length is greater than this, the attribute is stored in a subobject, often referred to as a *string large object* (SLOB). Similarly, a binary attribute with a length greater than 540 bytes is a *binary large object* (BLOB). For example, you could add a string attribute called **miscellaneous** to the **Customer** class and specify that the length as *maximum length*, which means the largest integer value.

The following diagram shows a **Customer** object with its subobjects.

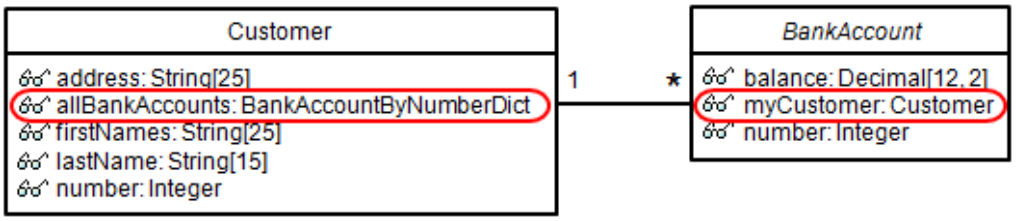


Although you can think of subobjects being created at the same time as the parent object, in reality they are not created until the first time they are used. In addition, subobjects are not fetched from the database unless they are being accessed in code.

Another type of subobject is the dynamic property cluster, which is used to store dynamic properties. When a dynamic property is added at run time, a database reorganization can be avoided, because the property is stored in a subobject rather than the parent object.

Inverse References

The one-to-many relationship between a customer and the bank accounts owned by the customer will be implemented by the **myCustomer** reference in the **BankAccount** class and the **allBankAccounts** reference in the **Customer** class.



If a bank account is created and its **myCustomer** reference is set to customer Mary Smith, the **Customer** object for Mary Smith *must* contain the bank account in its **allBankAccounts** collection. If this is not the case, something is wrong. This consistency requirement is similar to the referential integrity requirement for tables in a relational database.

You can enforce consistency in the relationship between **Customer** and **BankAccount** classes, by making the references involved *inverse references*.

myCustomer is the inverse of **allBankAccounts**, and **allBankAccounts** is the inverse of **myCustomer**. The benefits of inverse references are:

- You write code for an object at one end of the relationship only.
- Automatically the object (or objects) at the other end of the relationship are maintained in a consistent way. You

do not have to write this code.

- Not only do you write less code, but you avoid errors.

The following examples show the single instruction that you would write and the set of instructions that are effectively carried out as part of automatic inverse maintenance.

- A cheque account object is created and associated with a customer.

```
// instruction coded (manually)
account.myCustomer := cust;
```

```
// code executed (automatic maintenance)
account.myCustomer := cust;
cust.allBankAccounts.add(account);
```

- The cheque account object is associated with a new customer.

```
// instruction coded (manually)
account.myCustomer := newcust;
```

```
// code executed (automatic maintenance)
cust.allBankAccounts.remove(account);
account.myCustomer := newcust;
newcust.allBankAccounts.add(account);
```

- The cheque account object is deleted.

```
// instruction coded (manually)
delete account;
```

```
// code executed (automatic maintenance)
newcust.allBankAccounts.remove(account);
delete account;
```

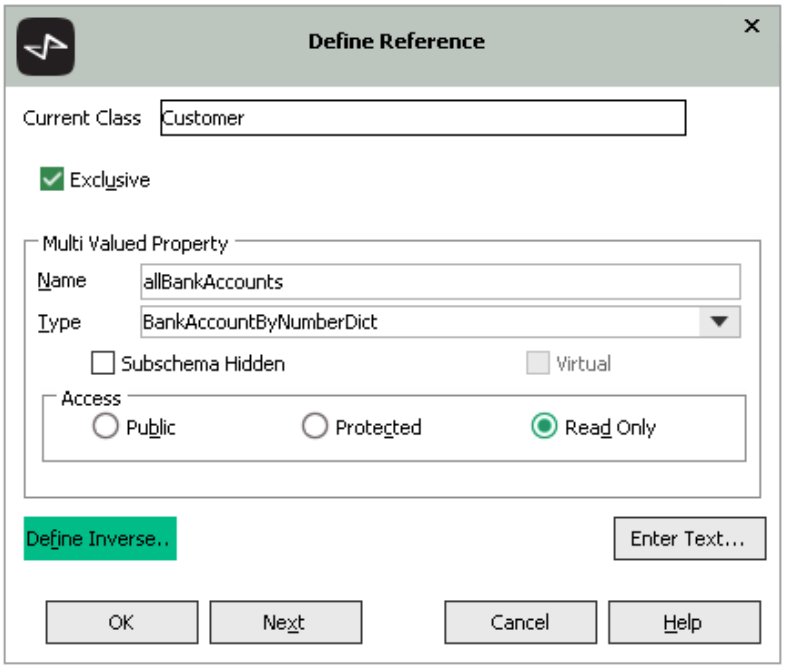
Note Deletions no longer result in collections with *invalid object references*, as they did before.

Adding Both Inverse References

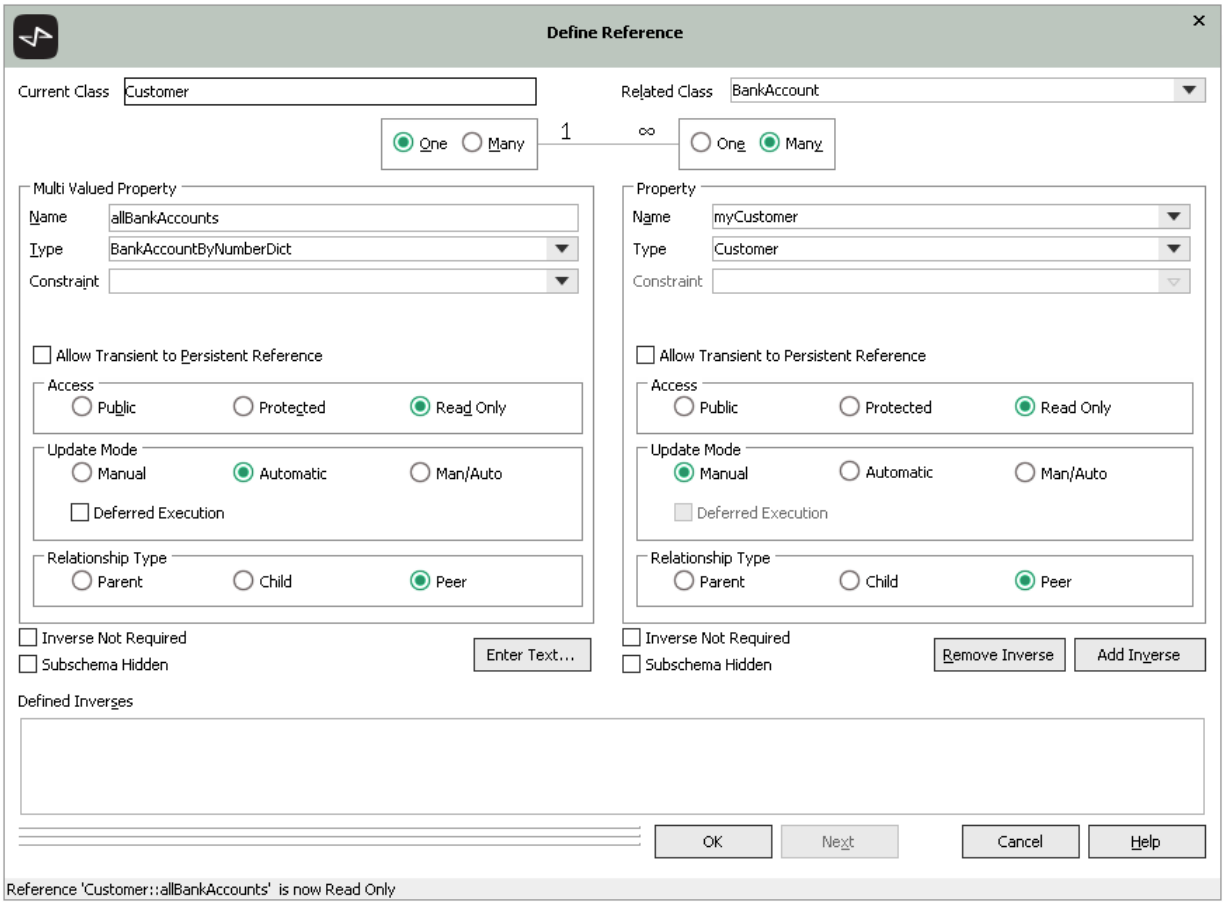
The one-to-many relationship between the **Customer** and **BankAccount** classes has been defined in the following three separate stages.

1. **myCustomer** reference is added to the **BankAccount** class.
2. **allBankAccounts** reference is added to the **Customer** class.
3. **myCustomer** and **allBankAccounts** references are set as inverse references.

The three stages are usually carried out at the same time, by clicking the **Define Inverse** button on the Define Reference dialog when you define the first reference.

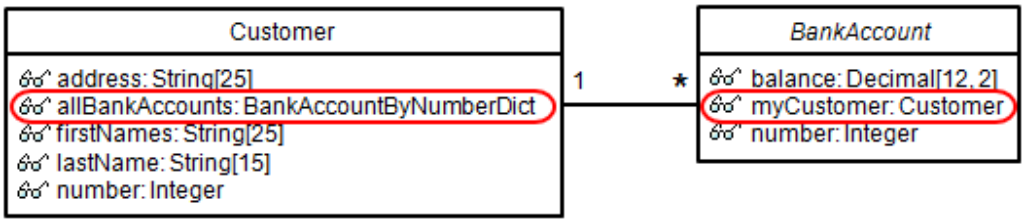


When the **Define Inverse** button is clicked, the dialog expands to show the related **BankAccount** class next to the **Customer** class. This enables you to add both inverse references at the same time.



Advice on Defining Inverses

It is helpful to draw the UML class diagram for the relationship (for example, with pen and paper) before attempting to enter information into the Define Reference dialog.



Automatic and Manual Updating

These options specify whether a reference is maintained manually (that is, in application code) or automatically as part of inverse maintenance.

- If the update mode of **myCustomer** is **Manual**, **allBankAccounts** is **Automatic**.

```
account.myCustomer := cust;          // Allowed
cust.allBankAccounts.add(cust);      // Not allowed (does not compile)
```

- If the update mode of **myCustomer** is **Automatic**, **allBankAccounts** is **Manual**.

```
account.myCustomer := cust;          // Not allowed (does not compile)
cust.allBankAccounts.add(cust);      // Allowed
```

- Alternatively, both update modes could be **Man/Auto**.

```
account.myCustomer := cust;          // Allowed
cust.allBankAccounts.add(cust);      // Allowed
```

Peer-to-Peer and Parent-Child Relationships

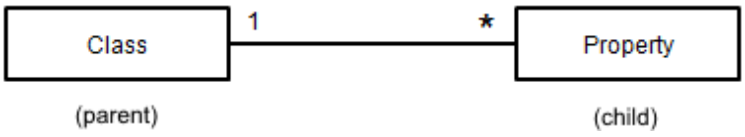
Peer-to-peer and parent-child relationships specify whether deleting one object causes related objects to be deleted.

Deleting a *parent* object causes the automatic deletion of the related *child* objects. However, the reverse is not the case. There is no automatic deletion when a *child* or a *peer* object is deleted.

If the relationship type of **myCustomer** is set to:

- **Parent**, **allBankAccounts** is **Child**
- **Child**, **allBankAccounts** is **Parent**
- **Peer**, **allBankAccounts** is **Peer**

Automatic deleting is useful for a *whole-part* aggregation relationship, where the *part* objects have meaning only as part of the whole *object*. The following example involves Jade meta data.



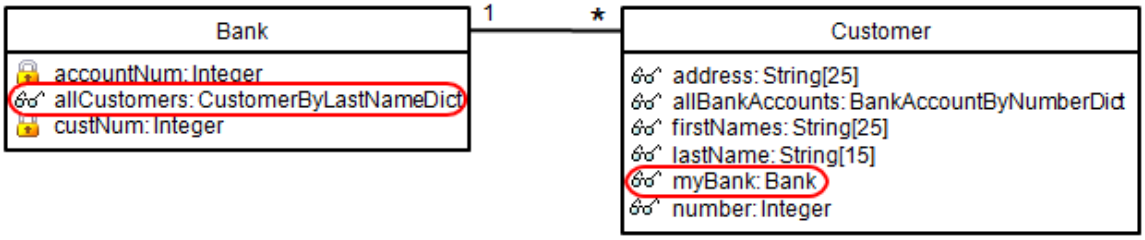
The **Customer** class object is the parent of the **address**, **firstNames**, and **lastName** property objects. If you were to remove the **Customer** class, the associated property and method objects would be deleted automatically.

Root Object Collections

One of the functions of the root object is to hold comprehensive collections (usually dictionaries) of instances of important classes in the system; for example, all of the customers, all of the bank accounts, and so on. You can use the root object collections in an application to display data in tables, and to navigate to any object in the system.

Inverse references are used to maintain the collections and to avoid invalid object references.

The first relationship to implement is one bank (the root object) that has many customers, as follows.



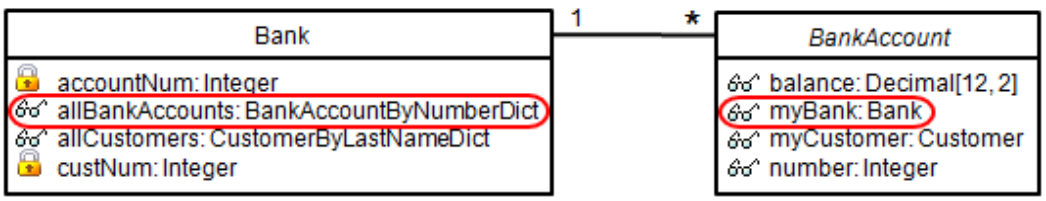
After defining the inverse references, a coding change is required to ensure that the **myBank** reference is set for a new customer. This can be done in the **onCreate** method in the **Customer** class, as follows.

```
onCreate( pTA : CustomerTA ) updating;

begin
    inheritMethod( pTA );

    self.number := app.myBank.nextCustNum();

    self.myBank := app.myBank;
end;
```

Note There is a general rule to set references after setting attributes. In the **onCreate** method, setting the **myBank** reference at the start of the method would be inefficient, because it triggers inverse maintenance, which in this case adds the customer to the **Bank** root object's **allCustomers** dictionary.

At the start of the method, the **lastName** property has not been set until we call **inheritMethod**, so the customer would be added to the dictionary with a null key. When the **lastName** property is subsequently set, additional dictionary maintenance is required.

The next relationship is one bank (the root object) that has many bank accounts, as follows.



After defining the inverse references, a coding change is required to ensure that the **myBank** reference is set for a new bank account.

This can be done in the **onCreate** methods in the **ChequeAccount** and **SavingsAccount** classes, as follows.

```
onCreate( pTA : ChequeAccountTA ) updating;

begin
    inheritMethod( pTA );

    self.myBank := app.myBank;
end;
```

```
onCreate( pTA : SavingsAccountTA ) updating;

begin
    inheritMethod( pTA );

    self.myBank := app.myBank;
end;
```

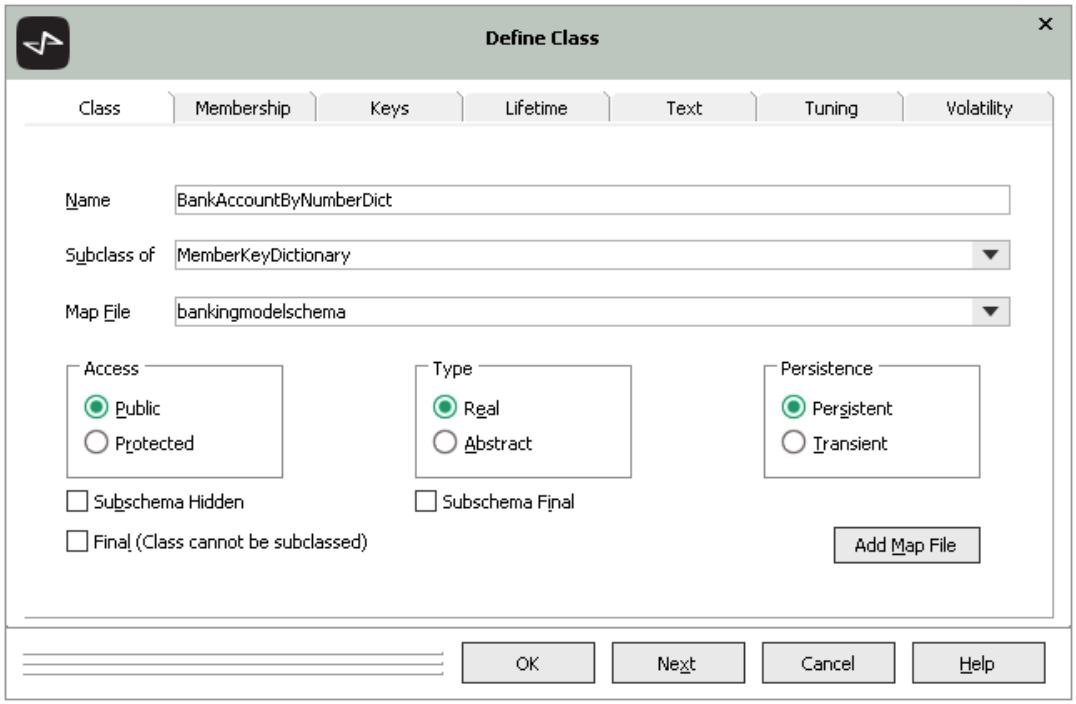
Note We want to have a **myBank** reference on the **Customer**, **ChequeAccount**, and **SavingsAccount** classes. Rather than adding this reference on each of these classes and setting the reference in the **onCreate** method on each of these classes, we will add this property once at the **BMSModel** level so that it is inherited by all of its subclasses.

Exercise 10.1 - Adding a BankAccount Dictionary

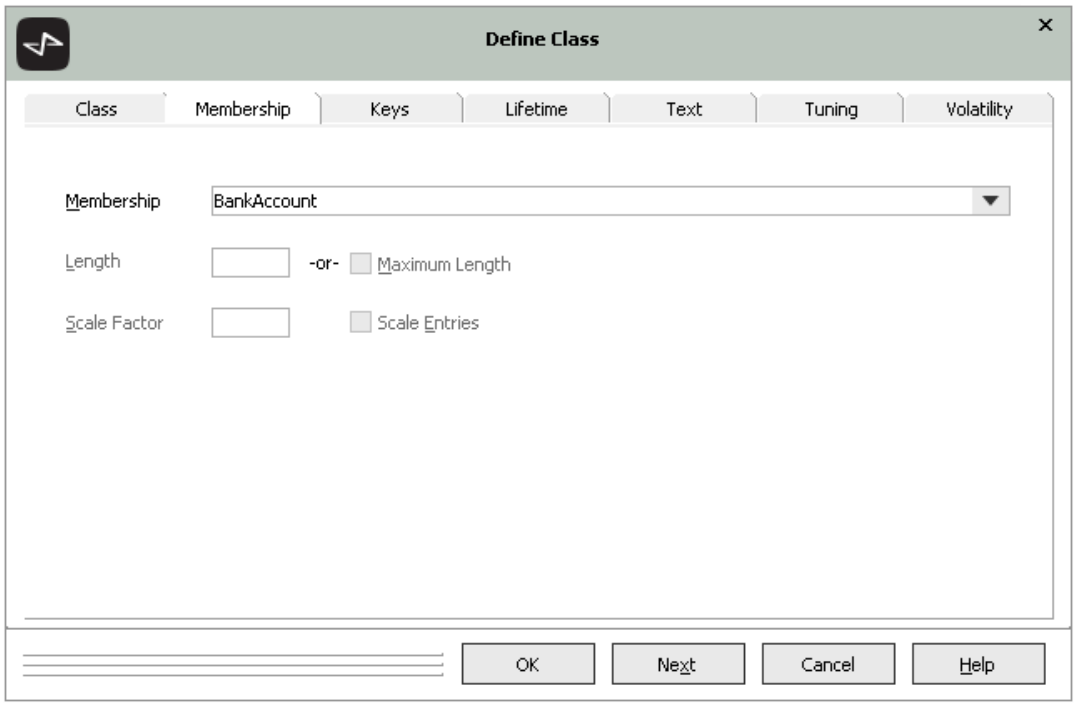
In this exercise, you will add a **BankAccountByNumberDict** dictionary. The instructions are similar to those for adding the **CustomerByLastNameDict** dictionary, except that the key property for **BankAccountByNumberDict** is guaranteed to be unique, so there is no need to allow duplicates.

1. Find the **MemberKeyDictionary** class.
2. Add a subclass to the **MemberKeyDictionary** class by selecting the Classes menu **Add** command.

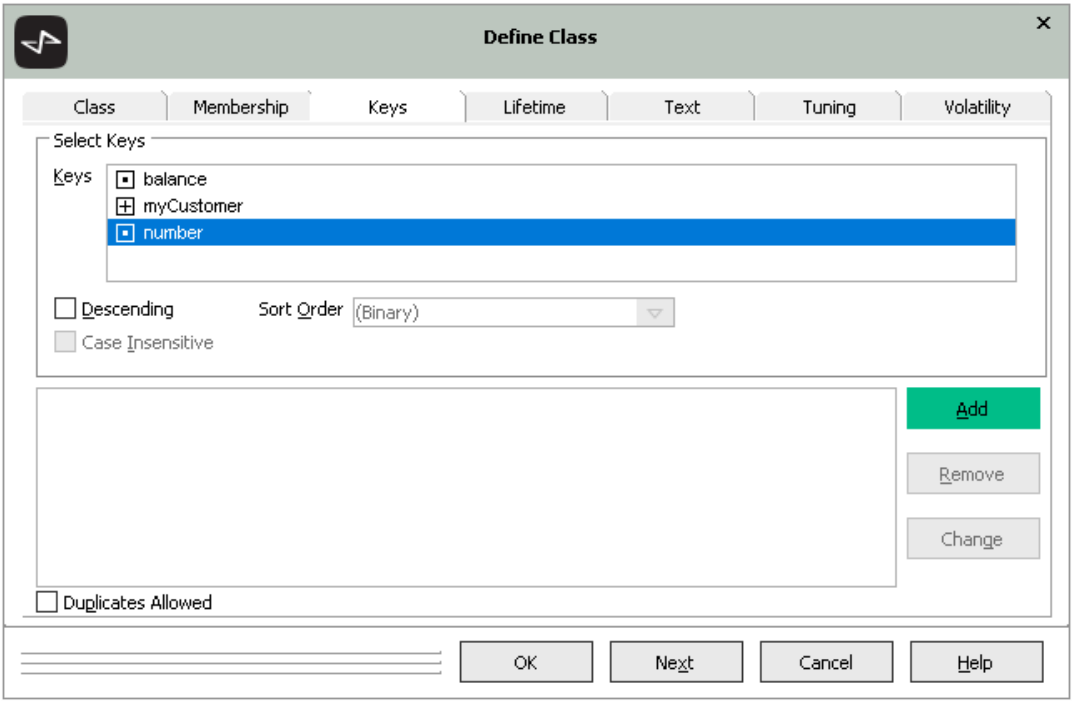
3. On the **Class** sheet, enter **BankAccountByNumberDict** as the name of the class, and then select the **Membership** sheet.



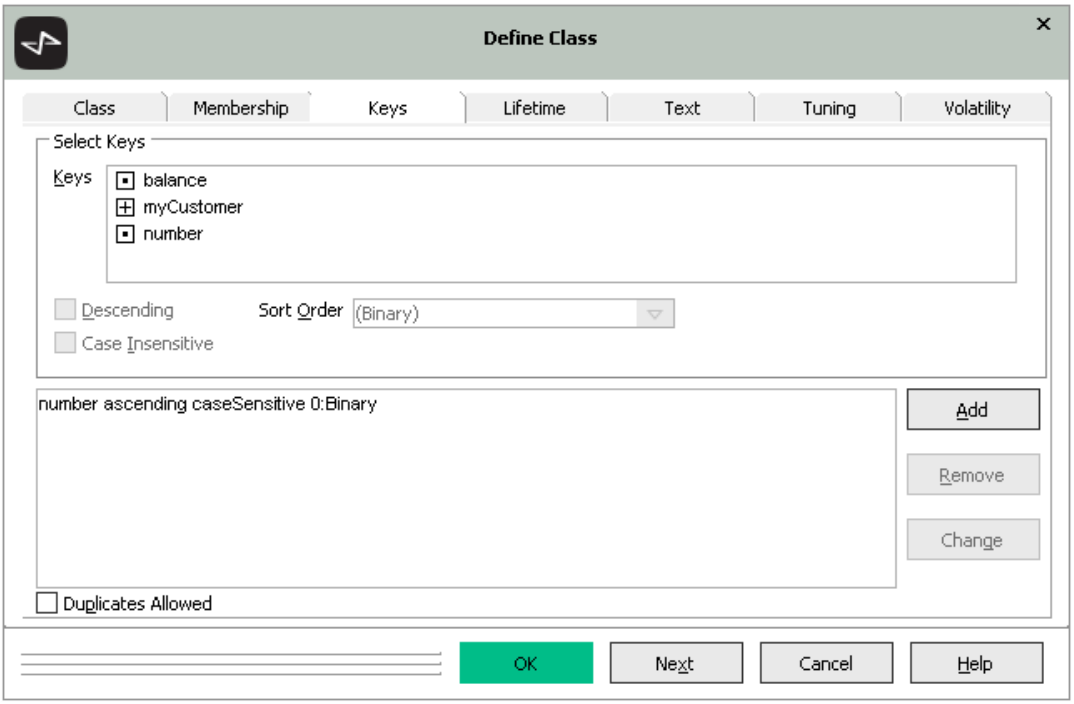
4. On the **Membership** sheet, select **BankAccount** as the **Membership** class and then select the **Keys** sheet.



5. On the **Keys** sheet, select **number** as the key and then click the **Add** button.



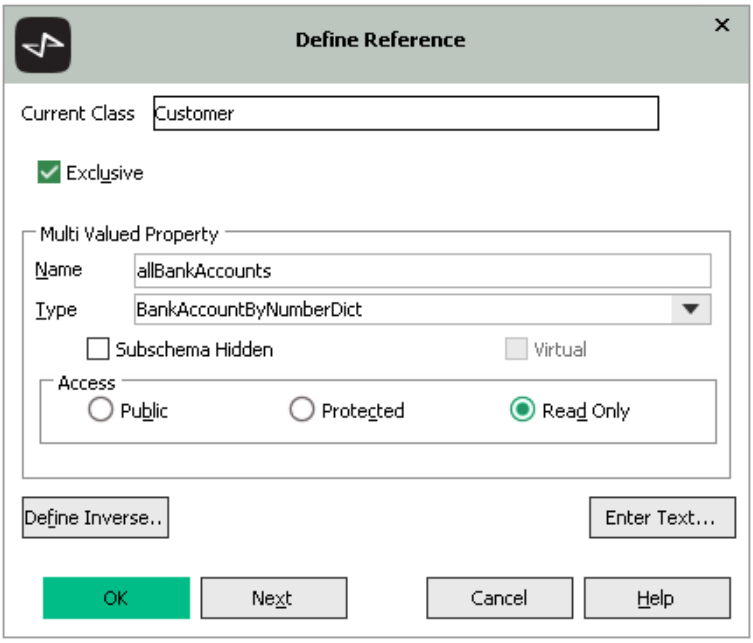
6. Click the **OK** button.



Exercise 10.2 - Adding an Exclusive Collection

In this exercise, you will add an **allBankAccounts** reference property.

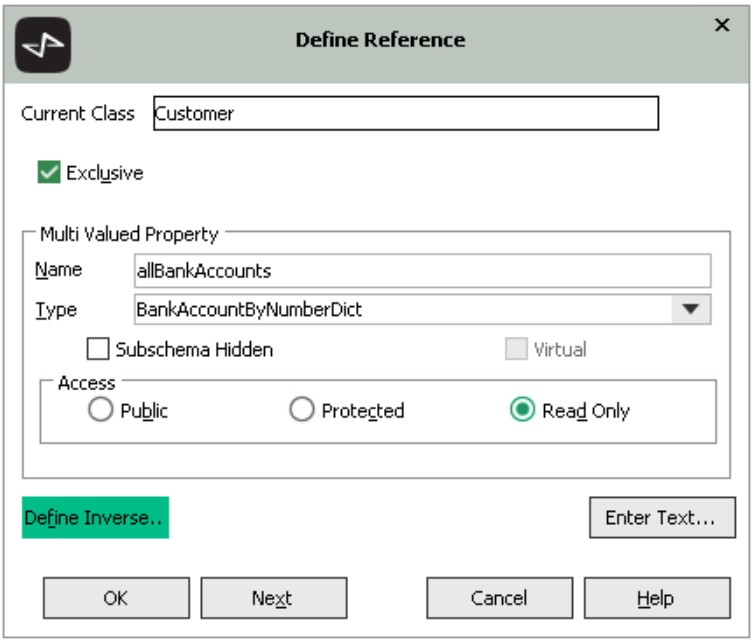
1. Select the **Customer** class.
2. Add a reference property by selecting the Properties menu **Add Reference** command.
3. Enter **allBankAccounts** as the name, make the reference read-only, and then click the **OK** button. (You may be warned that a reorganization is required when adding this reference.)



Exercise 10.3 - Adding Inverse References

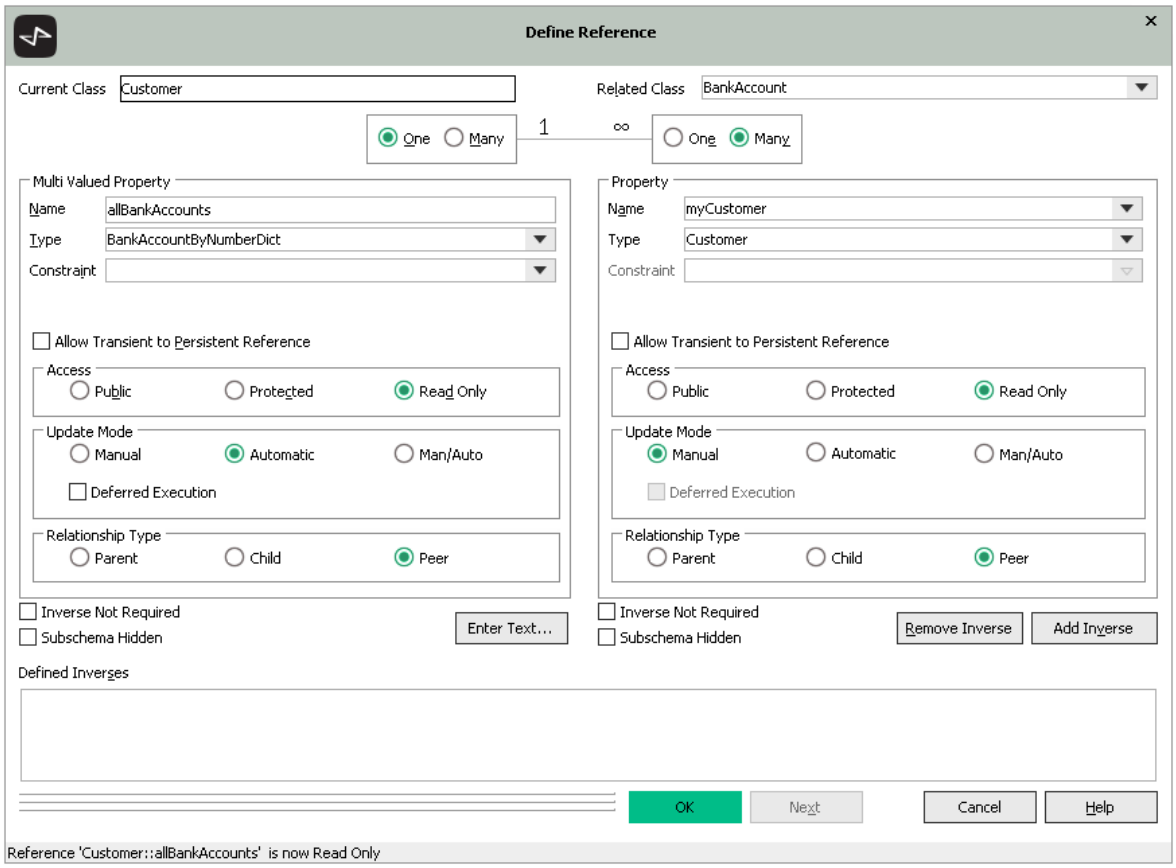
In this exercise, you will associate the **allBankAccounts** reference in the **Customer** class and the **myCustomer** reference in the **BankAccount** class as inverses.

1. Select the **allBankAccounts** reference in the **Customer** class.
2. Select the Properties menu **Change** command.



3. Click the **Define Inverse** button.

4. In the **BankAccount** class, select the **myCustomer** reference and then click the **OK** button.



5. This change will require a reorganization. Click the **Schema Needs Reorg** toolbar button and on the Classes Needing Reorg dialog, click the **Reorg** button.

Exercise 10.4 - Adding Root Object Collections

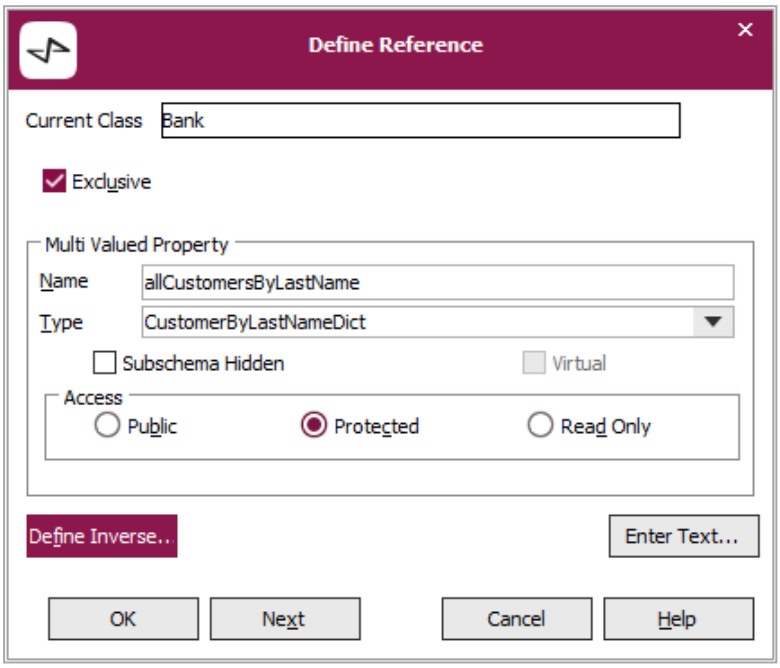
In this exercise, you will add the root object collections of **Customer** and **BankAccount** objects. You will also change the **onCreate** method on the **BMSModel** class so that new instances are automatically added to these collections.

1. Select the **BMSModel** class in the Class Browser.
2. Add a read-only reference property called **myBank** of type **Bank**.
3. Add a method called **onCreate**.
4. Code the method as follows.

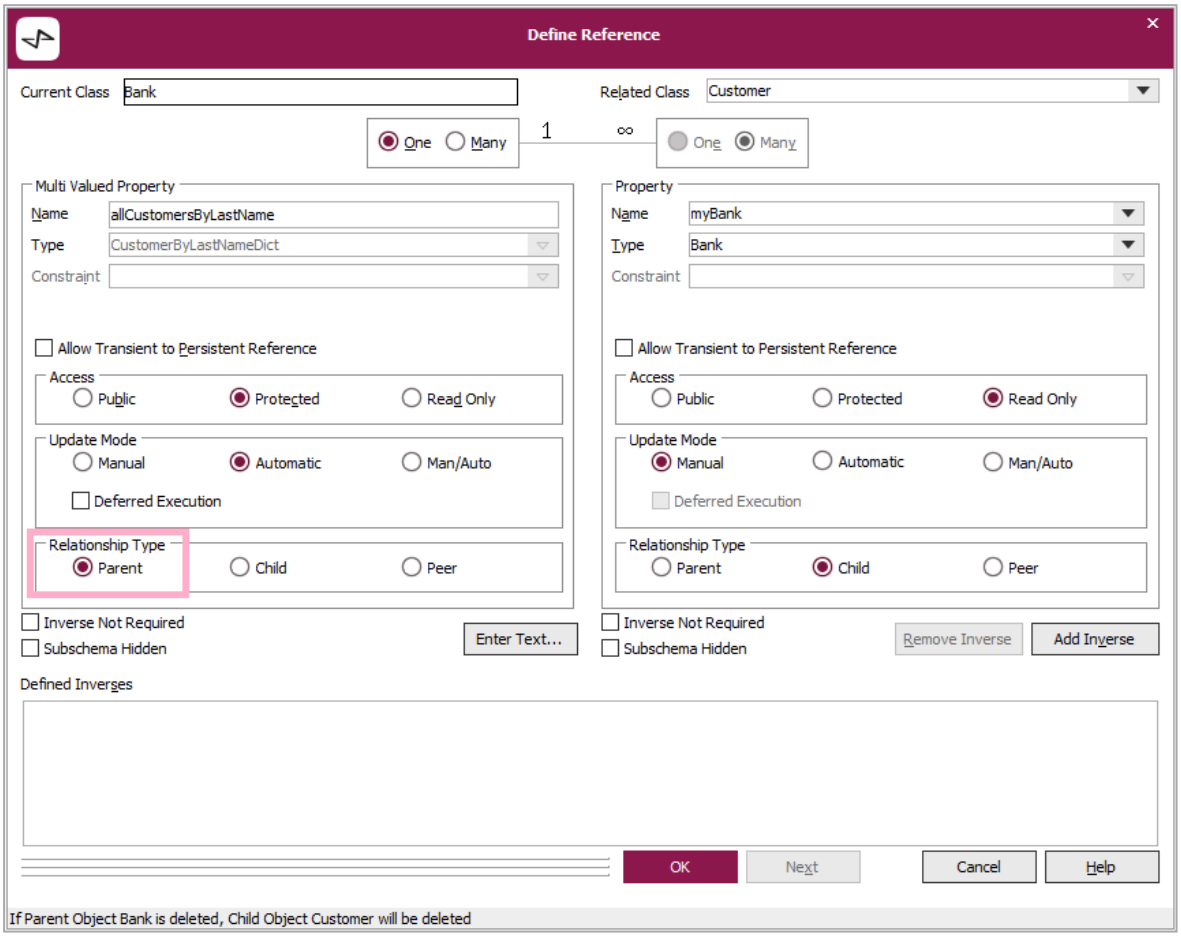
```
onCreate( pTA : BMSModelTA ) updating;

begin
    inheritMethod( pTA );

    self.myBank := app.myBank;
end;
```

5. Compile the method using F8.
6. Select the **Bank** class.

7. Add a reference called **allCustomersByLastName** of type **CustomerByLastNameDict** class, and then click the **Define Inverse** button.

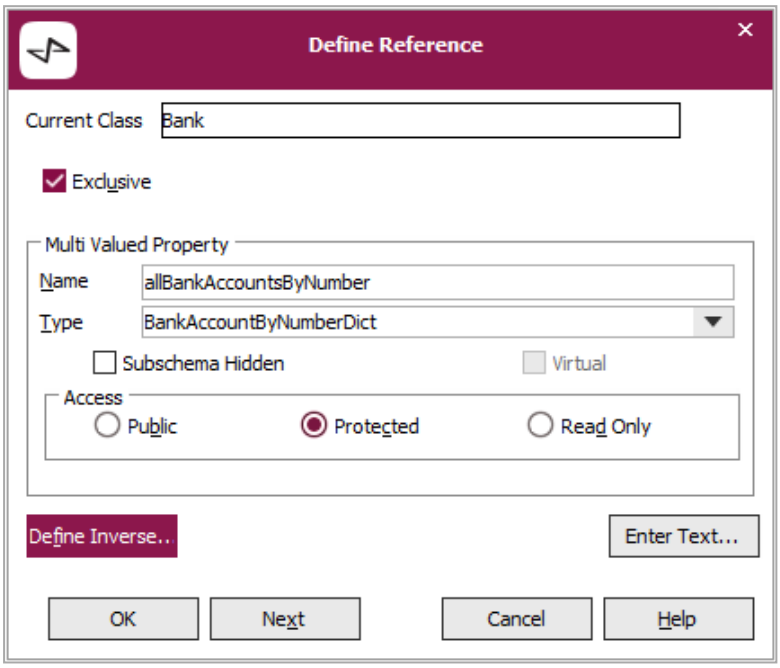


8. In the **Customer** class, select the existing **myBank** property in the **Name** drop-down list box, set the **Bank** side as the parent, and then click the **OK** button.

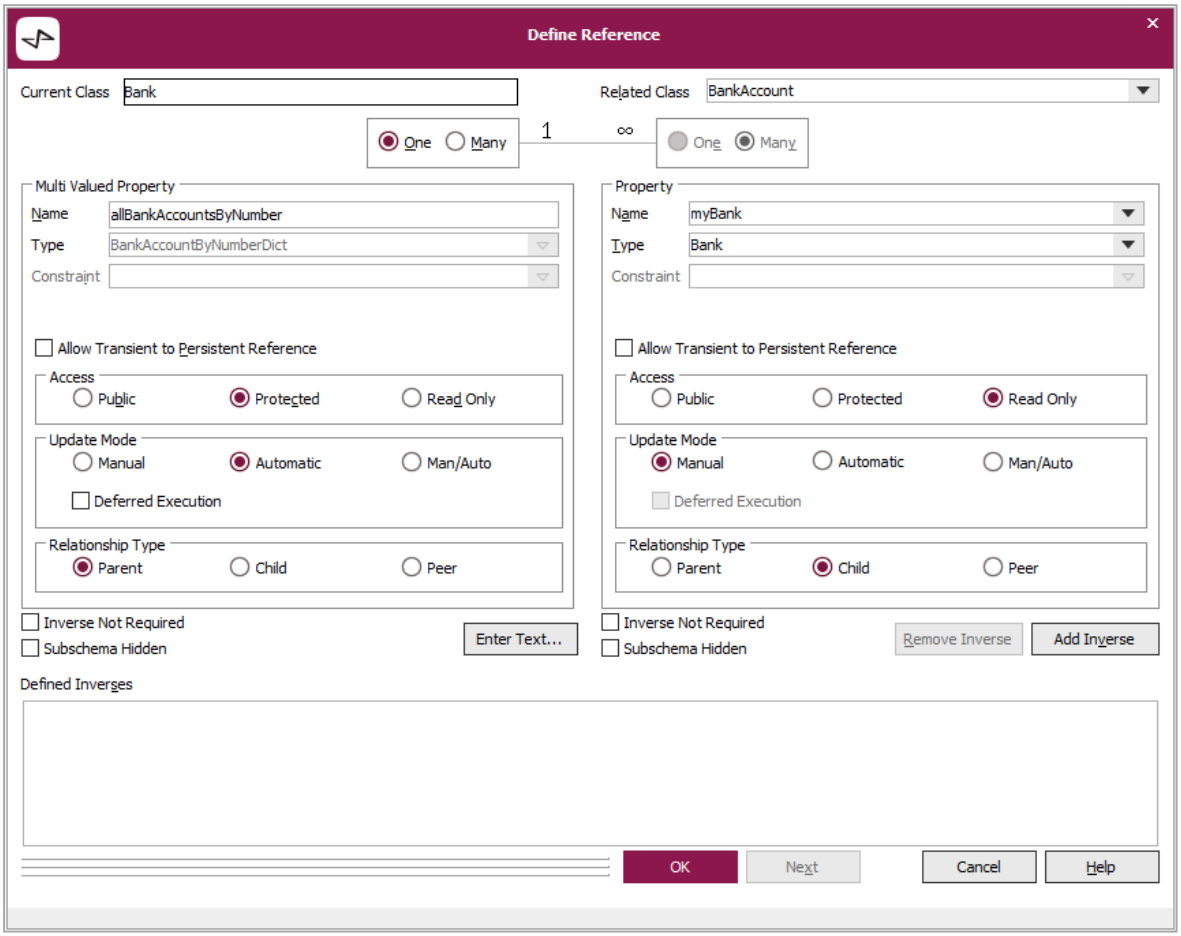


9. You are then prompted that the schema has been versioned. Perform a reorganization now.

10. Add a reference called **allBankAccountsByNumber** of type **BankAccountByNumberDict** class and then click the **Define Inverse** button.



11. In the **BankAccount** class, select the existing **myBank** property in the **Name** drop-down list and then click the **OK** button.



12. You are then prompted that the schema has been versioned. Perform a reorganization now.

13. Navigate to the **JadeScript** class and execute the **removeTestData** method.

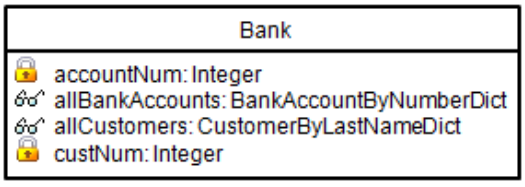
14. Execute the **createCustomersFromFile** and **createBankAccounts** methods. This will reload the test data, this time with the **myBank** reference set.

Extra Challenge: How might you establish this inverse relationship without deleting and reloading the test data?

Exercise 10.5 - Multiple Inverses

At this stage, the **Bank** root object has two collections, as follows.

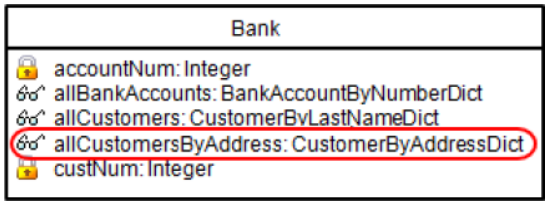
- A collection of bank accounts ordered by number
- A collection of customers ordered by last name



In the following two challenges, you can add further collections to the root object that could prove useful in the banking system applications.

Challenge #1

Add a reference called **allCustomersByAddress**, containing customer references but ordered by address, which is the inverse of **myBank** in the **Customer** class. You will need a new **CustomerByAddressDict** member key dictionary.

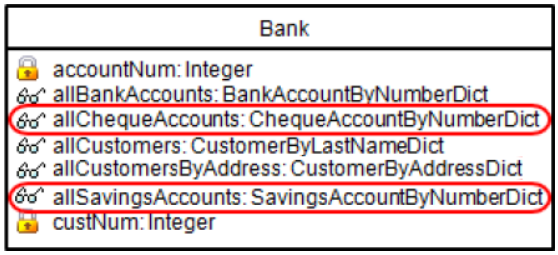


When the **myBank** reference is set for a new customer, the customer is added to the **allCustomers** collection and the **allCustomersByAddress** collection.

Challenge #2

Add a reference called **allChequeAccounts**, containing references to cheque accounts ordered by number, which is the inverse of **myBank** in the **BankAccount** class. You will need a new **ChequeAccountByNumberDict** member key dictionary.

Add a reference called **allSavingsAccounts**, containing references to savings accounts ordered by number, which is the inverse of **myBank** in the **BankAccount** class. You will need a new **SavingsAccountByNumberDict** member key dictionary.



When the **myBank** reference is set for a new bank account, the bank account is added to the **allBankAccounts** collection.

Depending on its type, the bank account is also added to the **allChequeAccounts** collection or the **allSavingsAccounts** collection.

Conditions

You can define a condition on a class by selecting the Methods menu **New Condition** command.

A condition is a declarative method that returns a **Boolean** result. You cannot use local variables and you are restricted to:

- Properties of the **self** object
- Other conditions on the class

- **if** and **return** instructions

The following condition could be added to the **BankAccount** class.

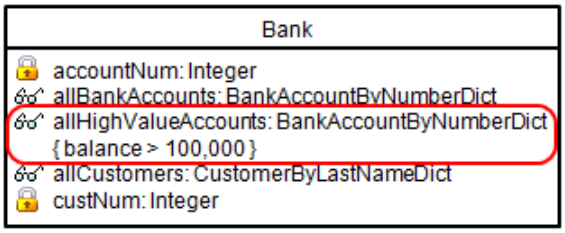
```
highValue(): Boolean condition;

begin
    return self.balance >= 100000;
end;
```

A condition method is indicated by the checkmark symbol (✓) displayed at the left of the method name in the Methods List of the Class Browser.

Constraint on Collection Maintenance

For a collection that is the automatically maintained end of the relationship, you can specify a constraint that determines whether an object should be added to or removed from the collection as part of the inverse maintenance. For example, the **Bank** root object could have an **allHighValueAccounts** collection of accounts with balances greater than $100,000.

This collection for bank accounts with no condition on the balance is in addition to the **allBankAccounts** collection.

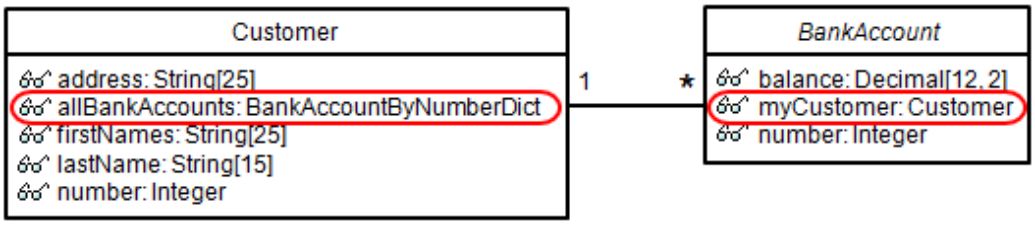


When an account is created, depending on the initial balance, inverse maintenance adds it to the **allHighValueAccounts** collection. Subsequently, as the balance changes through deposits and withdrawals, the bank account will be removed automatically from or added to the collection, depending on whether the condition is met.

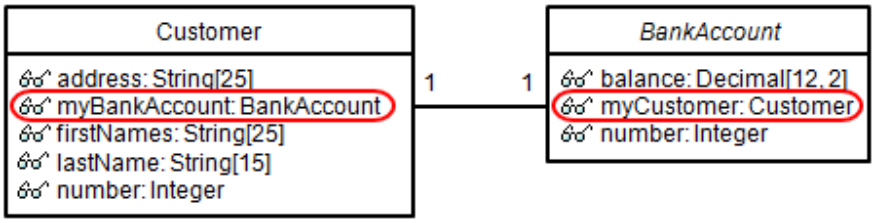
Cardinality

Cardinality is the number of objects at the ends of a relationship. A one-to-many relationship, which is the type you have defined in this module, has a **my** reference at one end and an **all** reference at the other. One collection is required.

One customer has many bank accounts.

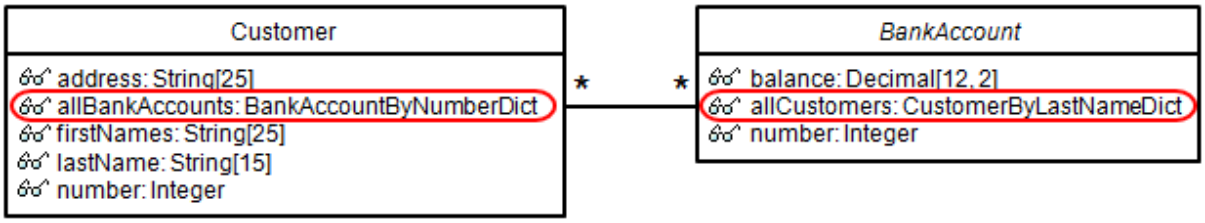


A one-to-one relationship has **my** references at both ends. No collections are required.



Note Restricting a customer to a single bank account is not realistic.

A many-to-many relationship has **all** references at both ends. Two collections are required.



Note A bank account owned by two or more customers is a *joint* account.

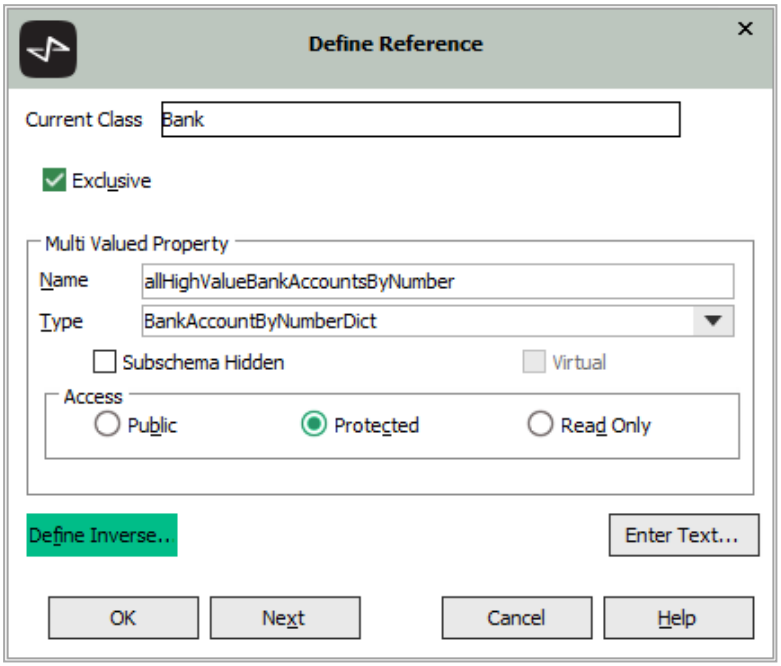
Exercise 10.6 - Adding an allHighValueAccounts Collection

In this exercise, you will add a **highValue** condition to the **BankAccount** class, and then add an **allHighValueAccounts** collection to the **Bank** class. To demonstrate that the inverse maintenance works as expected, you will write a **testHighValue** JadeScript method.

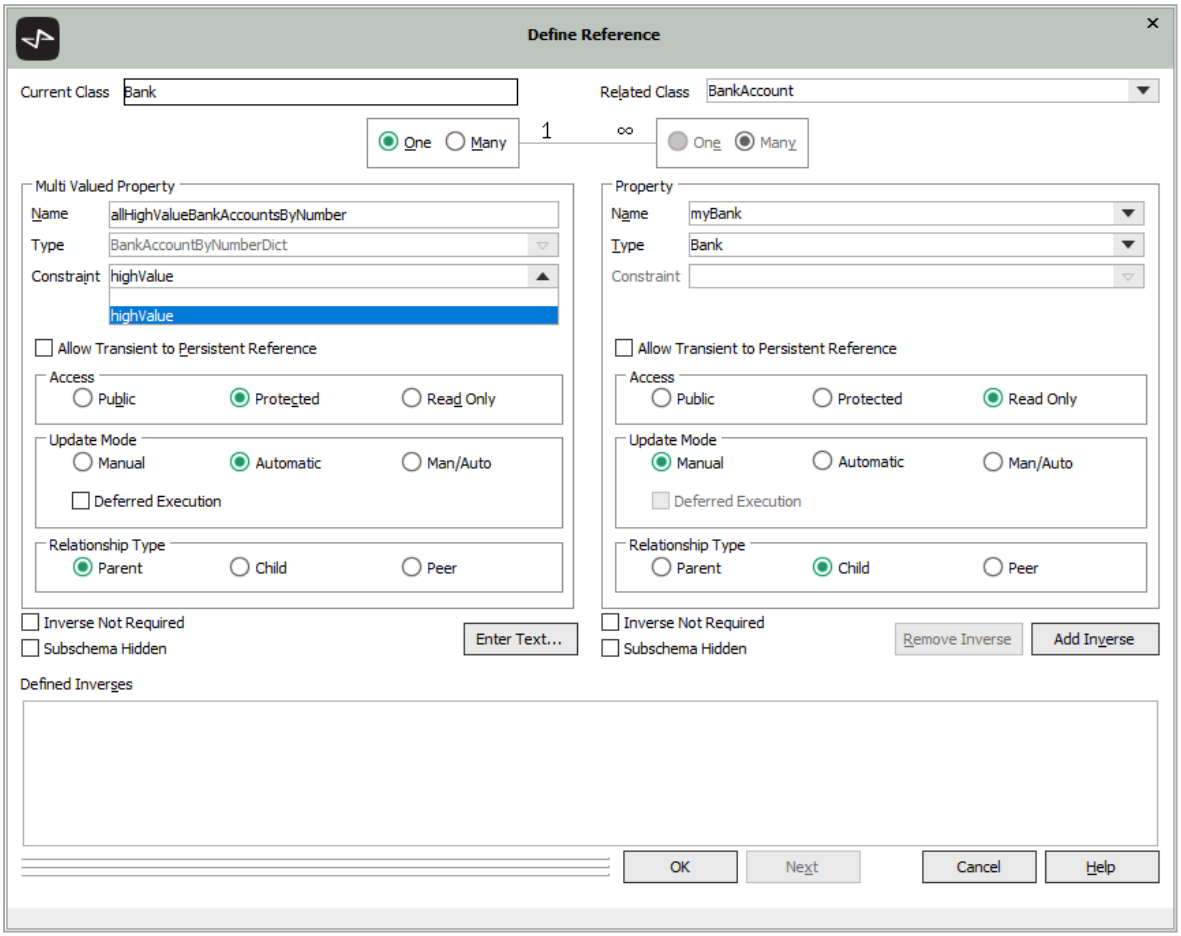
1. Select the **BankAcount** class.
2. Add a condition called **highValue**, by selecting the Methods menu **New Condition** command.
3. Code the method as follows.

```
highValue() : Boolean condition;

begin
    return self.balance >= 1000000;
end;
```

4. Add a reference called **allHighValueBankAccountsByNumber** of type **BankAccountByNumberDict** to the **Bank** class and then click the **Define Inverse** button.



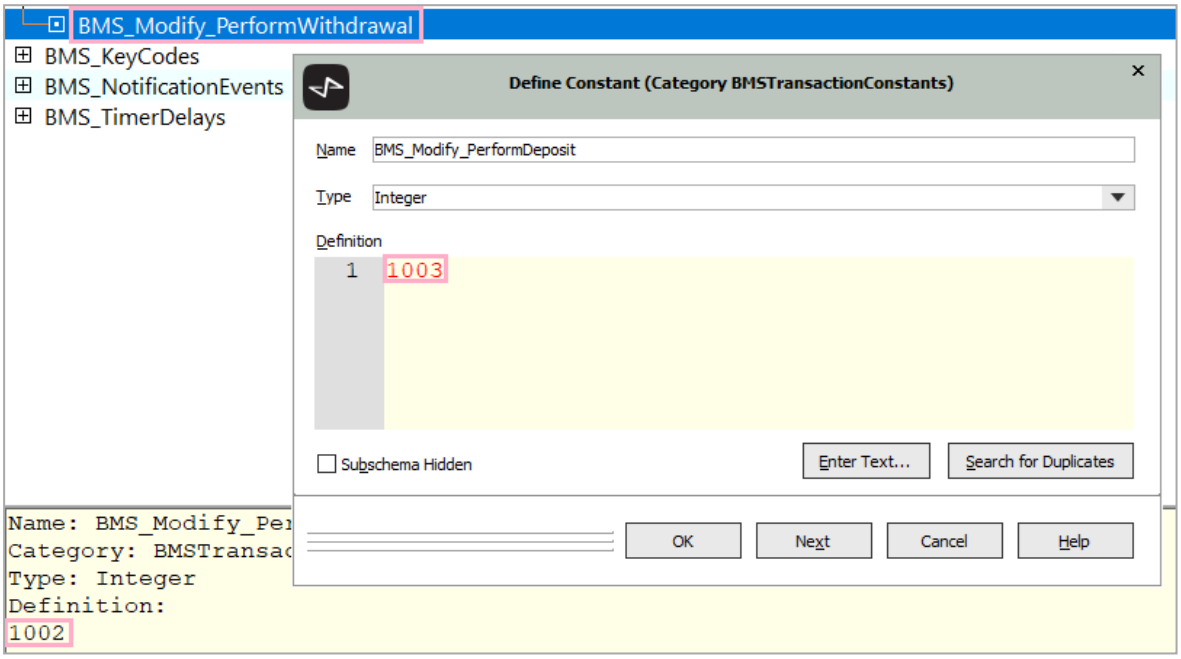
5. Select **highValue** in the **Constraint** combo box and **myBank** as the inverse reference, as shown in the following image.



You are then prompted that the schema has been versioned. Perform a reorganization now.

6. Press Ctrl+G to open the Global Constants Browser.
7. Select the **BMSTransactionConstants** category.
8. Select the Global Constants menu **Category Sorted by Value** command.
9. Select the last **BMS_Modify_*xxx*** entry so that you can see the current highest-numbered modification constant value.

10. Add a new global constant called **BMS_Modify_PerformDeposit** by selecting the **Constant** command in the Global Constants menu and then the **Add** command in the submenu. Make the new global constant an Integer constant with a value that is +1 larger than the current highest-numbered modification constant value.



11. Select the **BankAccount** class in the Class Browser.

12. Select the existing **onModify** method.

13. Add additional code as follows.

```
onModify( pTA : BankAccountTA ) updating;

begin
    if pTA.modificationCode = BMS_Modify_PerformWithdrawal then
        self.balance -= pTA.transactionAmount;

    elseif pTA.modificationCode = BMS_Modify_PerformDeposit then
        self.balance += pTA.transactionAmount;

    endif;
end;
```

14. Compile the method using F8.

15. Add a JadeScript method called **testHighValue** that creates a cheque account with a zero balance, uses the **deposit** method to put the bank account into the **allHighValueAccounts** collection, and then uses the **withdraw** method to remove it from the collection.

```
testHighValue();

vars
    chequeAccountTA : ChequeAccountTA;

begin
    app.initialize();

    create chequeAccountTA transient;

    chequeAccountTA.initialize();
    chequeAccountTA.overdraftLimit := null;
    chequeAccountTA.balance := null;
    chequeAccountTA.persistEntity( BMS_Full_update );
    write app.myBank.allHighValueBankAccountsByNumber.size();       // Outputs 0

    chequeAccountTA.transactionAmount := 1000000;
    chequeAccountTA.persistEntity( BMS_Modify_PerformDeposit );
    write app.myBank.allHighValueBankAccountsByNumber.size();       // Outputs 1

    chequeAccountTA.transactionAmount := 1;
    chequeAccountTA.persistEntity( BMS_Modify_PerformWithdrawal );
    write app.myBank.allHighValueBankAccountsByNumber.size();       // Outputs 0

epilog
    delete chequeAccountTA;
end;
```

16. Execute the JadeScript method.

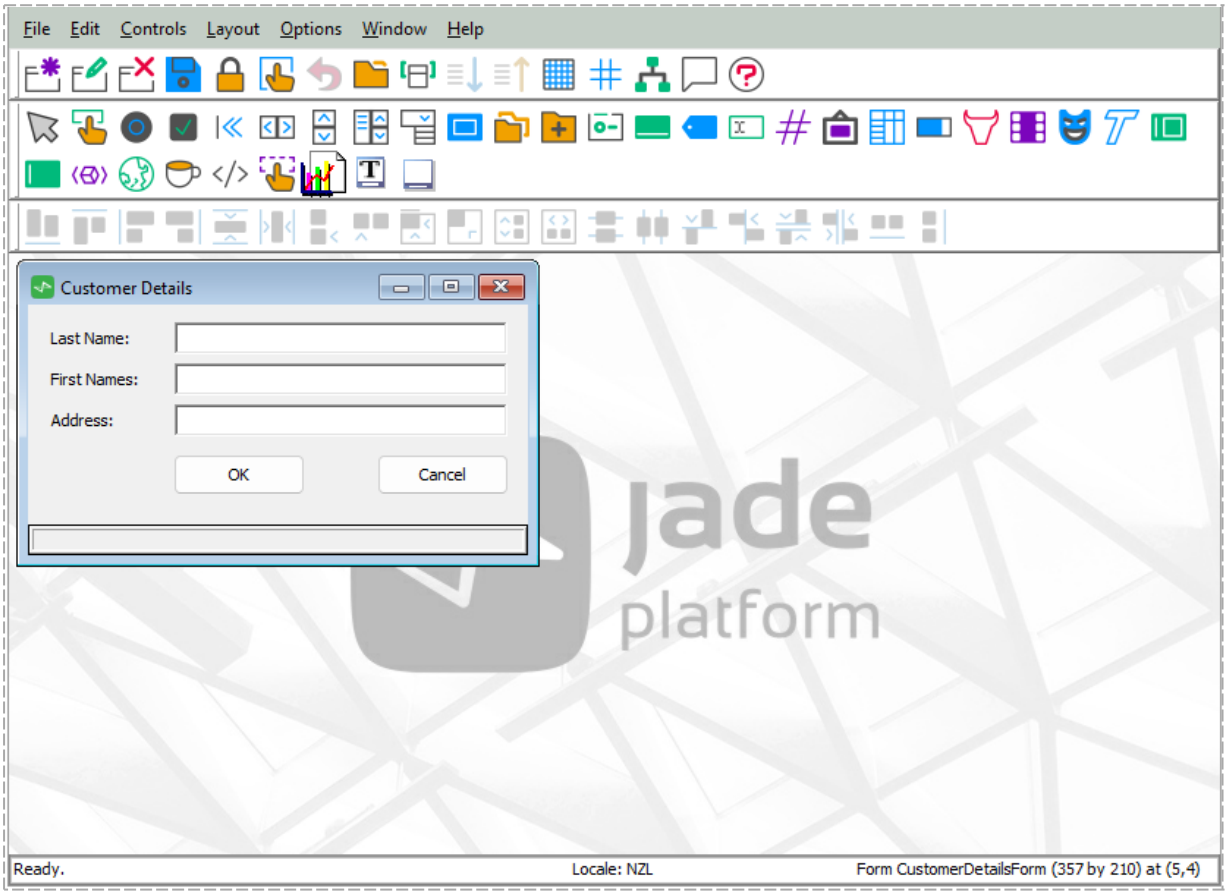
Module 11 Forms

This module contains the following topics.

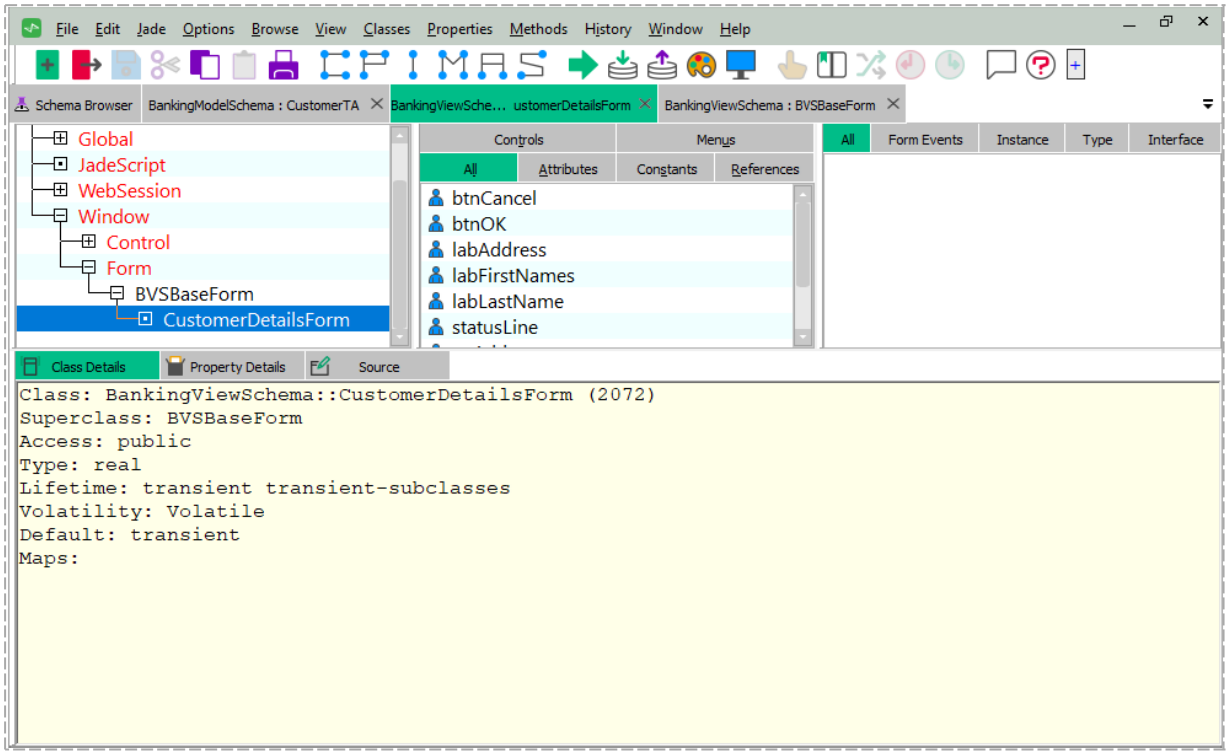
Introduction

The **BankingModelSchema** implements the model for the system. All classes for which persistent objects are created are defined in this schema.

You can open the separate Painter application by selecting the File menu **Painter** command in the Jade Platform development environment, or by clicking the paintbrush icon from the development environment toolbar. After creating a form and adding controls in the **BankingViewSchema**, save the form by selecting the File menu **Save Form** command.



The Class Browser displays a class corresponding to the form you designed in the Painter.



You add functionality to the form by writing code in this class.

You can select a runtime skin that is used to display any form that you are painting, by selecting the **Select Skin** command from the File menu. The Select or Cancel a Skin form is then displayed, to enable you to select the runtime skin in the **Choose Skin** combo box.

If you have not loaded any runtime skins into your Jade system, the default value of **<None>** is the only value available in this combo box.

Tip The **examples\skins** subfolder of the Jade Platform install files contains runtime skins that you can load. For details about loading the **SampleSkins.ddx** file, see the **readme.txt** file in that subfolder.

When you select a runtime skin, the **Control Examples** pane on the form displays an example of controls (and menu and menu items, if selected for display) using that skin.

When you are happy with the controls and menu on the painted form displayed in that skin, click the **Apply** button. That skin is then applied to any forms being painted. If a skin is selected, the JADE Painter caption reads Jade Painter : *schema-name::form-name* - using skin '*skin-name*' - [*caption-of-form*]; for example:

```
Jade Painter : DemoSchema::Company - using skin 'Windows Broadbean' - [Company]
```

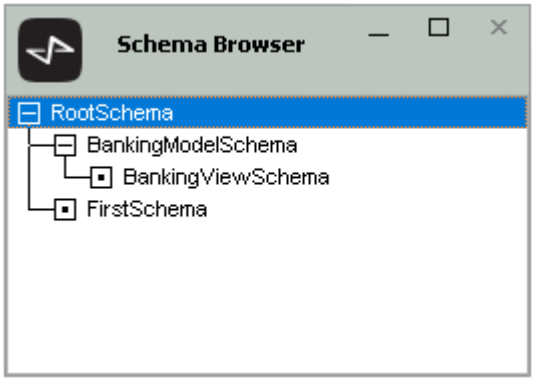
In addition, any subsequent forms opened in the JADE Painter are displayed using the selected runtime skin. The selected skin is saved in your user preferences when you close the JADE Painter and restored when you re-open the Painter.

View Schema

The **BankingViewSchema** implements the views or applications that run over the model. The entire user interface (forms) is implemented in this schema. Jade uses subschemas to separate the model from the views, allowing for a cleaner, more well-defined design and implementation. It also means that separate development teams can more easily work on separate parts of the system, but still within the same single Jade Platform environment.

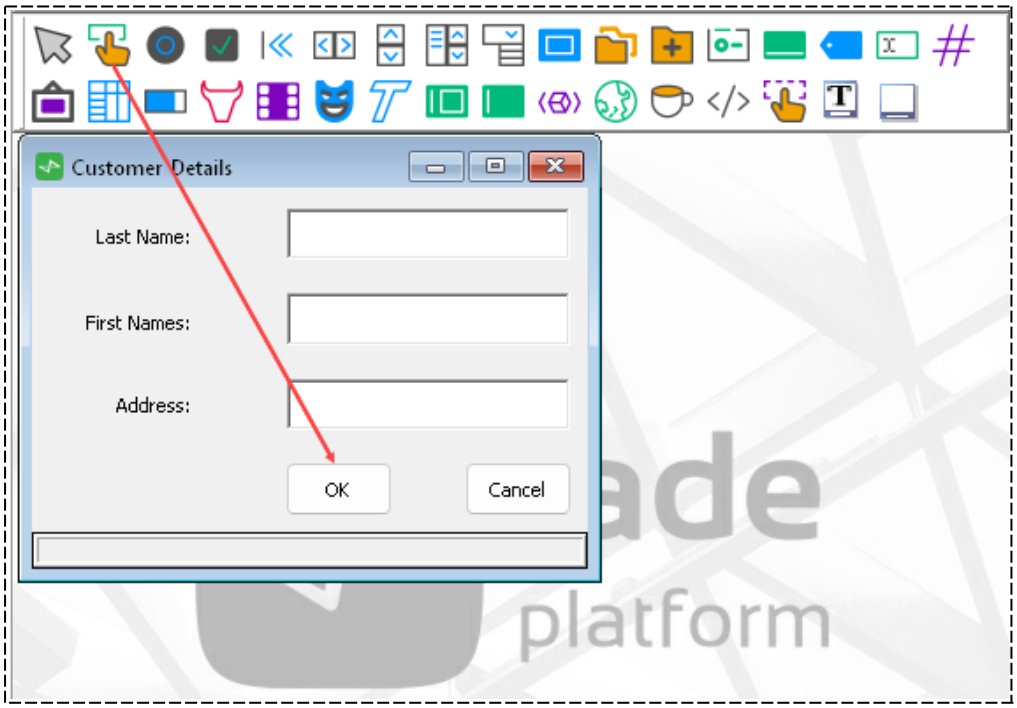
Separating the views from the model by packaging them in their own schemas prevents the model schema from becoming cluttered with user interface implementation, and means that the model schema can support many different views. It also makes it easier to identify the services provided by the model.

Create forms in a subschema (the **BankingViewSchema**, in this course).



Painter

To add a control to a form, click on the control in the **Tools** palette and then click on the form. Alternatively, use the Ctrl+Insert shortcut keys to display a text-based list of the controls that are available to be added.

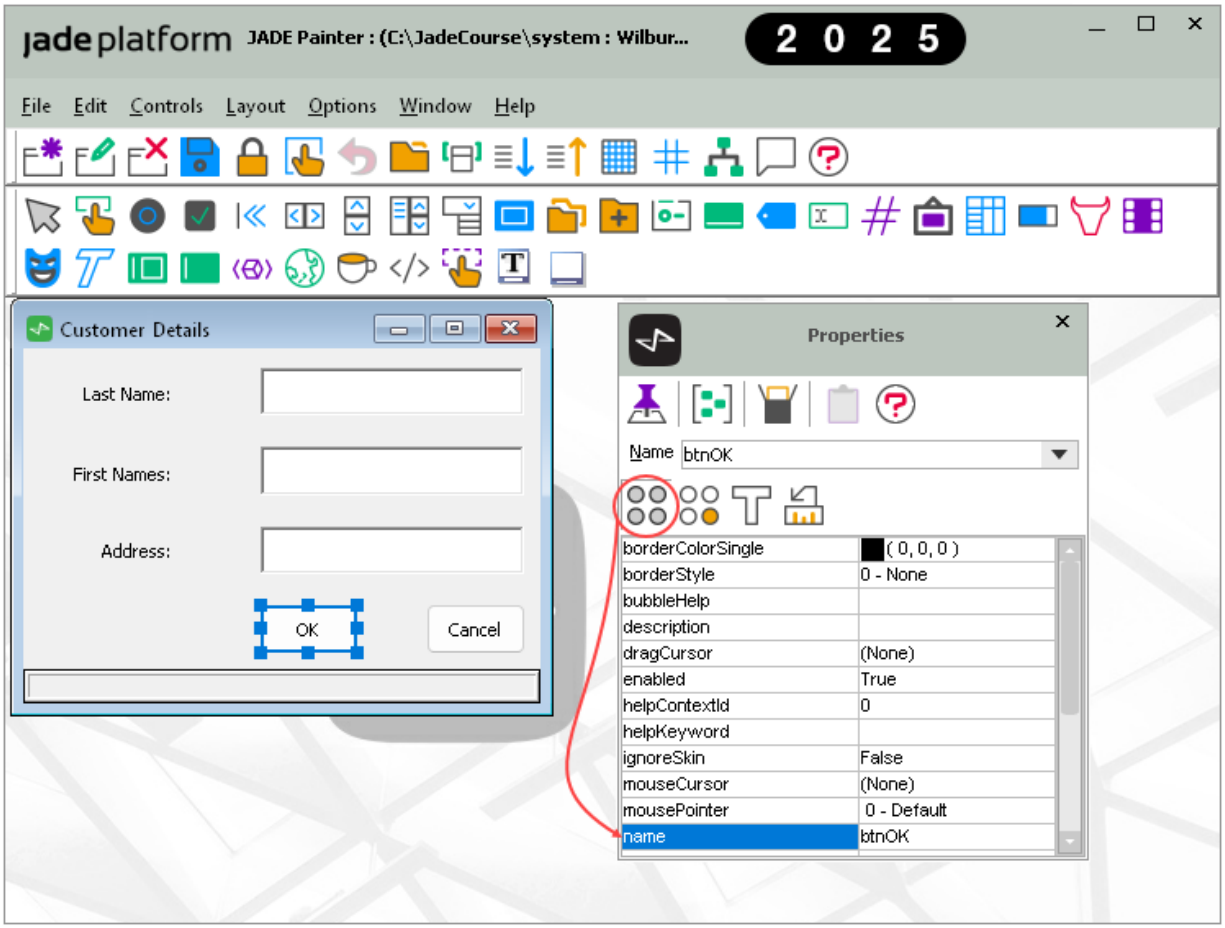


To change the properties of a control, double-click on the control to open the Properties dialog, which groups properties into the following categories.

- Common
- Specific

- Font and Color
- Size and Position

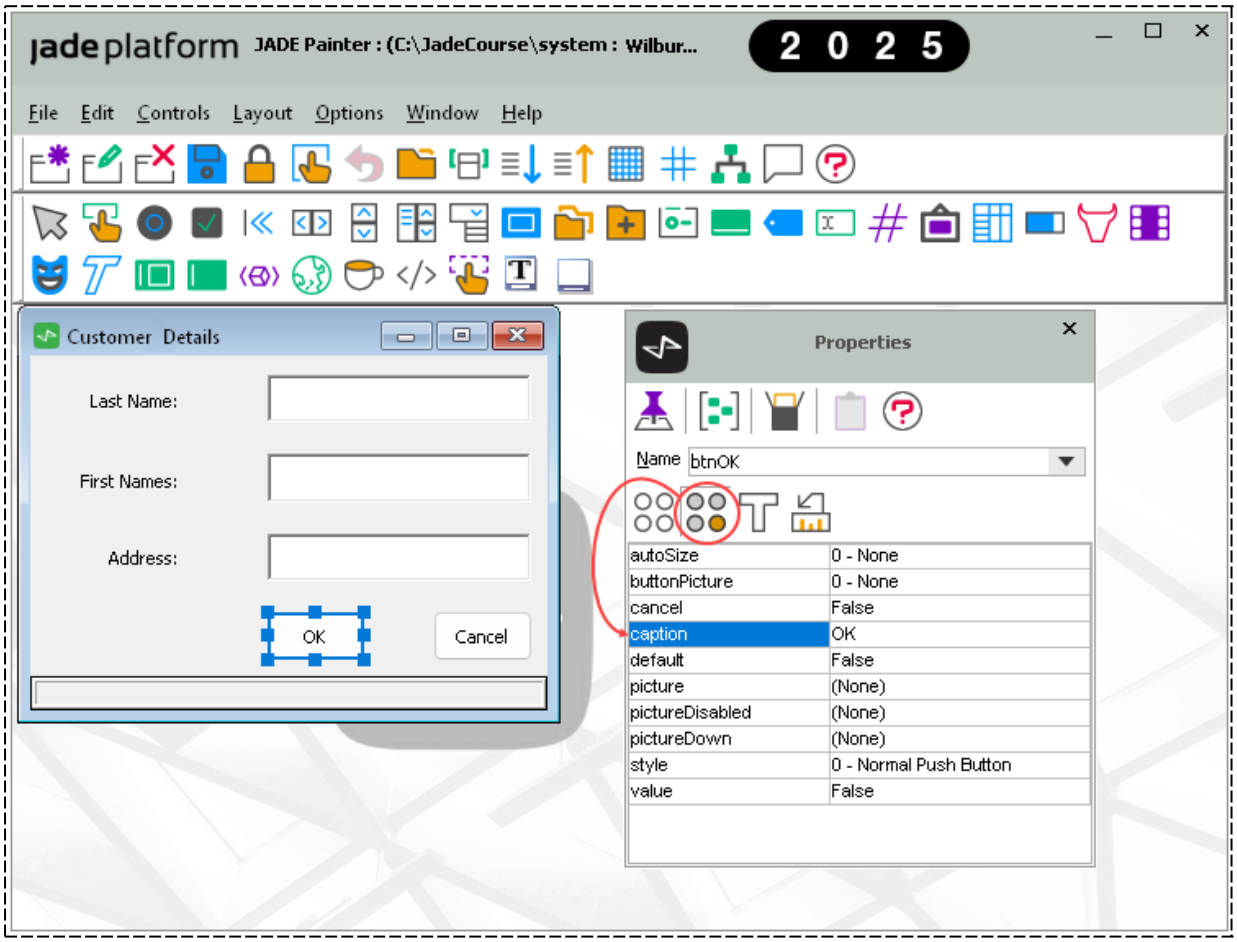
The **name** property is in the **Common** group of properties. The **Common** properties are those that every type of control has; for example, every control has a name. You use the **name** property when referring to the control in your code. You should change the default names **button1**, **button2**, and so on, to something more meaningful to a developer.



Tips Click the **Stay on top of Painter** icon at the left of the Properties dialog toolbar, to keep the Properties dialog positioned on top of the Painter. The icon then changes shape and is highlighted.

You can display a hierarchical list of all controls painted on the currently active form; for example, if you want to inspect the controls painted on a complex form. Activate the form by selecting the **Show Control Hierarchy Dialog** command from the Window menu of the Painter or by pressing F5 when the Painter has focus. Click the **Stay on top of Painter** icon at the top left of the dialog or select the **Control Hierarchy on Top** command from the Options menu to keep the Hierarchy for Form dialog on top of the Painter. Conversely, repeating these actions toggles the pinning of the dialog on top of the Painter and the check status of the menu command.

The **caption** property is in the **Specific** group of properties, because not all controls have captions. If all controls had captions, it would be in the **Common** group. The caption is the text seen by application users. You should change it to something more meaningful to an application user.



There is another toolbar with icons to help with alignment and sizing, displayed except when you select the **Hide Alignment/Size Palette** command from the Options menu.

Forms

Your form is a subclass of the **Form** class from **RootSchema**, which has inbuilt Windows functionality. The inherited **show** method loads and displays the form, and the **unloadForm** method closes it.

In the following JadeScript method, the **CustomerDetails** form is displayed for five seconds, and then closed.

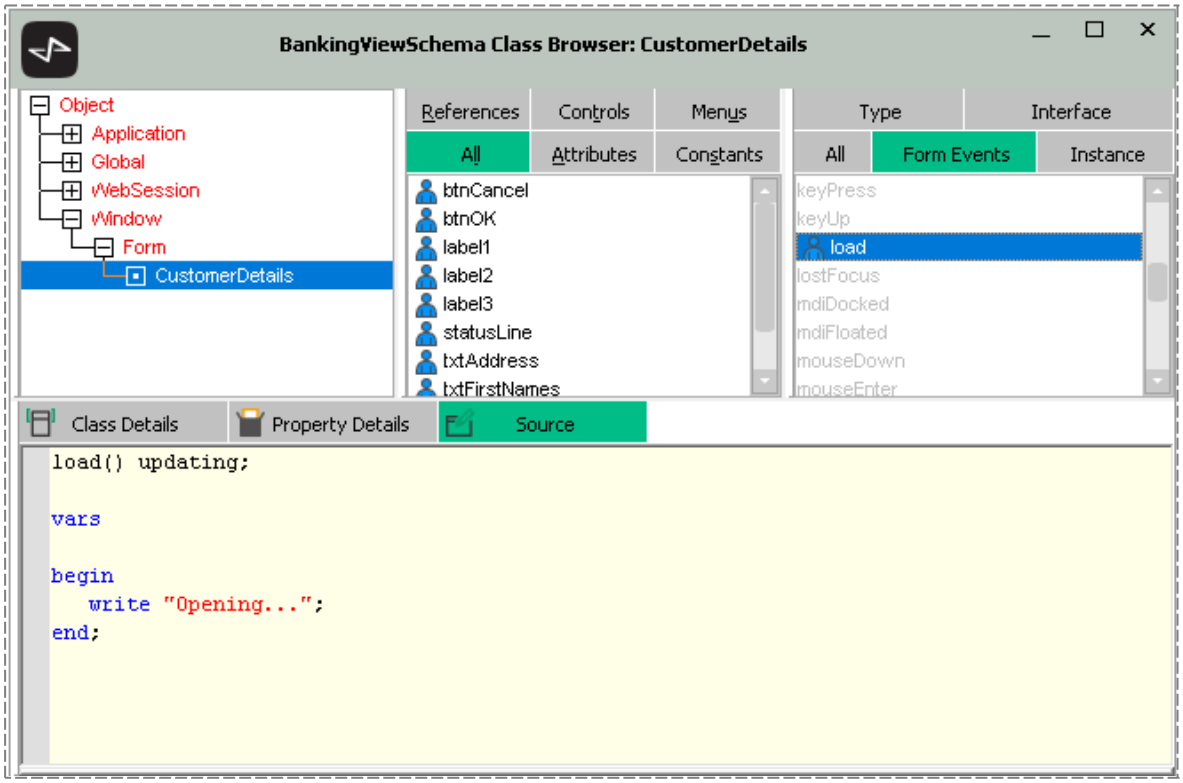
```
vars
    form: CustomerDetails;
begin
    create form transient;
    form.show();
    // Wait five seconds
    app.doWindowEvents(5000);
    form.unloadForm();
end;
```

Note The **unloadForm** method deletes the transient form object and the associated control objects.

The event method associated with the **show** method is called **load**. It enables text to be entered into text boxes and collections to be loaded into tables and list boxes. The event method associated with the **unloadForm** method is called **unload**.

Note Event methods are invoked when the associated event happens; for example, a button is clicked or a form is closed. They are not usually invoked directly with a method call from code.

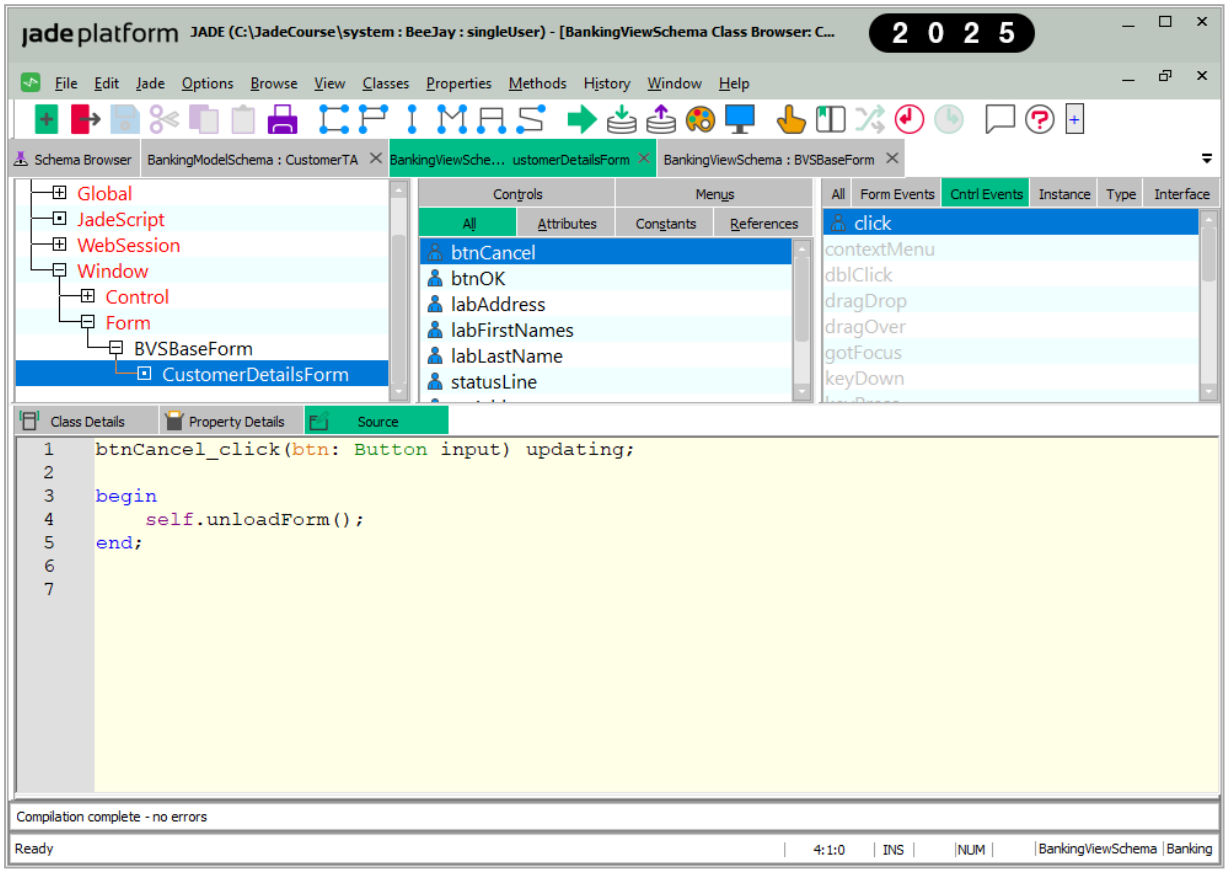
To code one of these event methods, select **<form>** in the central window (that is, the Properties List) and then select the appropriate event method from Methods List on the right.



Buttons

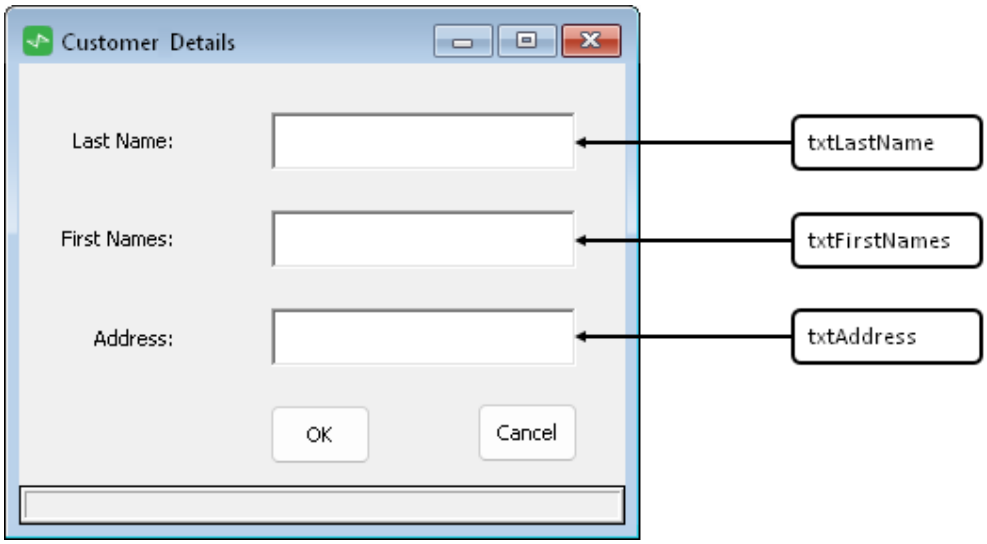
In a GUI application, most of the functionality is triggered when the application user clicks buttons on forms. To code a button **click** event method, select the button control in the central Properties List and then select the **click** event method from the Methods List on the right.

Write code in the editor pane and then compile the method.



Text Boxes

Text boxes enable an application user to enter text, which is stored in the text box's **text** attribute. The following diagram shows a form with **txtLastName**, **txtFirstNames**, and **txtAddress** text boxes.



You could add a **clearTextBoxes** method to the form to clear text from the text boxes and position the cursor in the **txtLastNames** text box.

```
clearTextBoxes();

begin
    self.txtLastName.text := "";
    self.txtFirstNames.text := "";
    self.txtAddress.text := "";
    self.txtLastName.setFocus();
end;
```

You could add an **isDataValid** method to the form to return **true** if data has been entered in all of the text boxes. If one of the text boxes is empty, a message is displayed in the status line and the method returns **false**.

```
isDataValid(): Boolean protected;

begin
    if self.txtLastName.text = "" then
        self.txtLastName.setFocus();
        self.statusLine.caption := "Please enter a last name";
        return false;
    elseif self.txtFirstNames.text = "" then
        self.txtFirstNames.setFocus();
        self.statusLine.caption := "Please enter first names";
        return false;
    elseif self.txtAddress.text = "" then
        self.txtAddress.setFocus();
        self.statusLine.caption := "Please enter an address";
        return false;
    endif;
    return true;
end;
```

You could add a **createCustomer** method to the form to create a **Customer** object from the data entered in the text boxes.

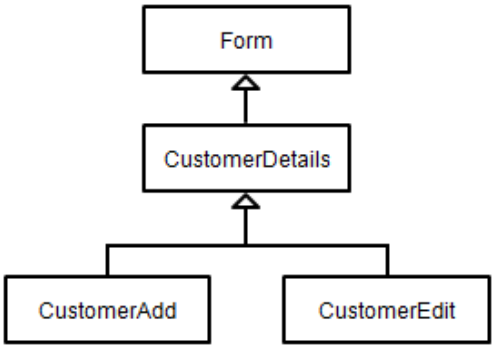
```
createCustomer() protected;

vars
    cust : Customer;
    address : String;
    firstNames : String;
    lastName : String;
begin
    address := self.txtAddress.text;
    firstNames := self.txtFirstNames.text;
    lastName := self.txtLastName.text;

    beginTransaction;
    cust := create Customer(address, firstNames, lastName) persistent;
    commitTransaction;
end;
```

Subforms

The **CustomerDetails** form has text boxes for displaying the attributes of a **Customer** object. Two situations in which you would use a form like this are when:

- Adding a new customer
- Editing an existing customer (possibly selected from a list box or table)

Instead of using the same form in both situations, which would inevitably involve more-complex code with **if** instructions, create two subforms.



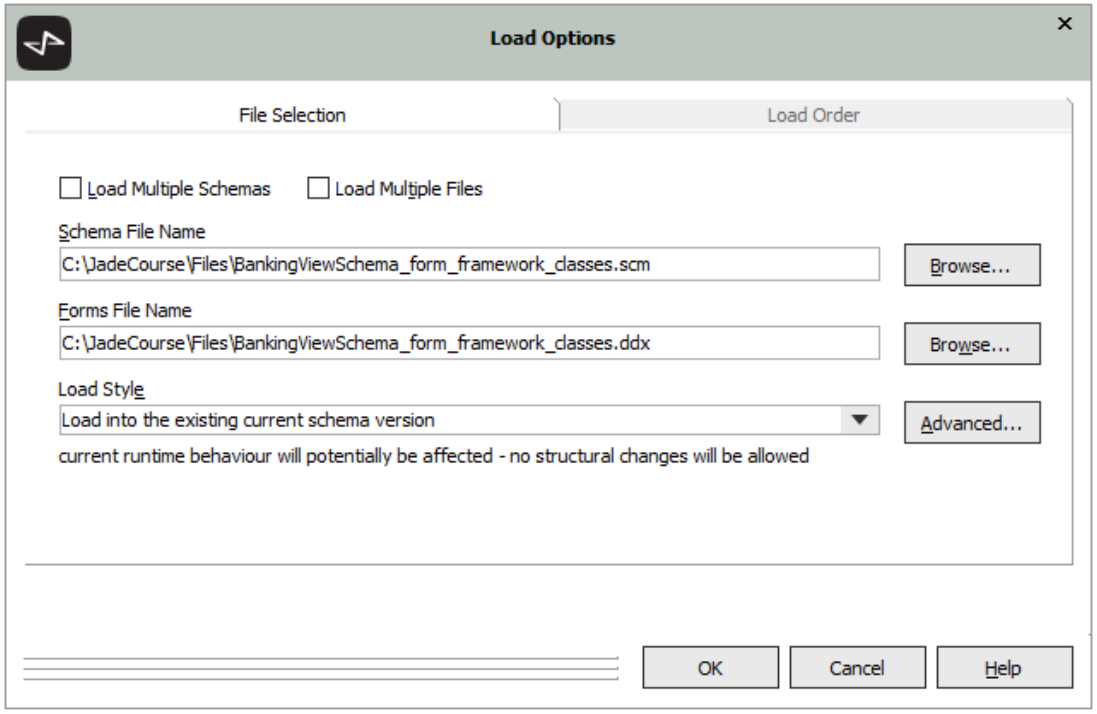
The **CustomerAdd** and **CustomerEdit** forms inherit controls, properties, and methods from **CustomerDetails**. In addition, the **CustomerEdit** class will have a **myCustomer** reference that is set to the **Customer** object to be edited.

Note Although you cannot make a form class abstract, the **CustomerDetails** form will be treated as an abstract class; that is, it will not be instantiated.

Exercise 11.1 - Adding the BankingViewSchema

In this exercise, you will create the **BankingViewSchema**, in which you will create forms and applications for the banking system.

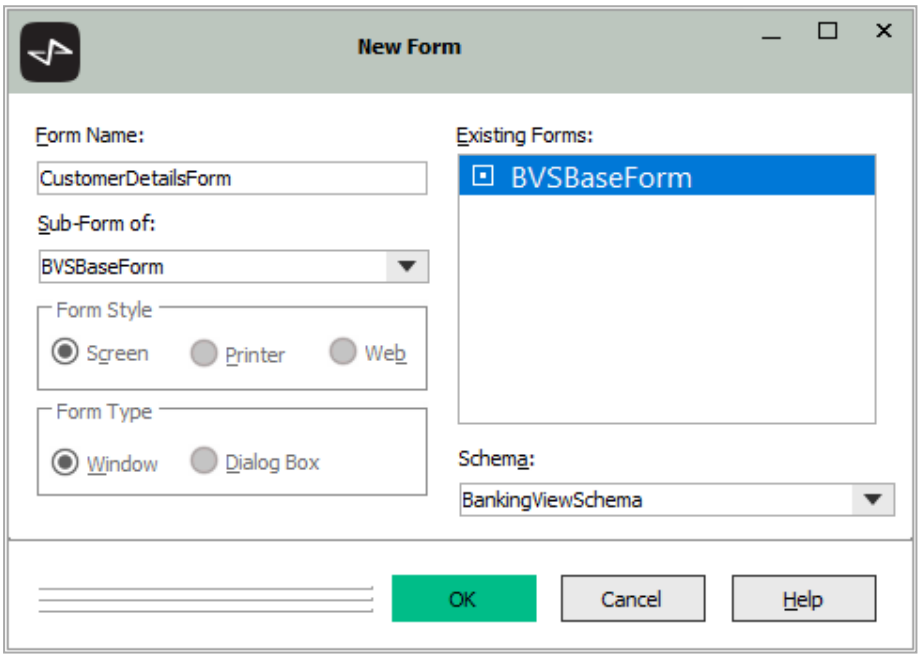
1. Select the **BankingModelSchema** in the Schema Browser.
2. Select the Schema menu **Add** command.
3. Enter **BankingViewSchema** as the name and then click the **OK** button.
4. Select the **BankingViewSchema** in the schema browser, if it is not already selected.
5. Select the Schema menu **Load** command.
6. Click the **Browse** button next to the **Schema File Name** text box and then select the **BankingViewSchema_form_framework_classes.scm** file that is included with the course files.
7. Click the **Browse** button next to the **Forms File Name** text box and then select the **BankingViewSchema_form_framework_classes.ddx** file that is included with the course files.
8. For the **Load Style**, select **Load into the existing current schema version** from the drop-down list.
9. Click the **OK** button to complete the loading of the form framework classes.



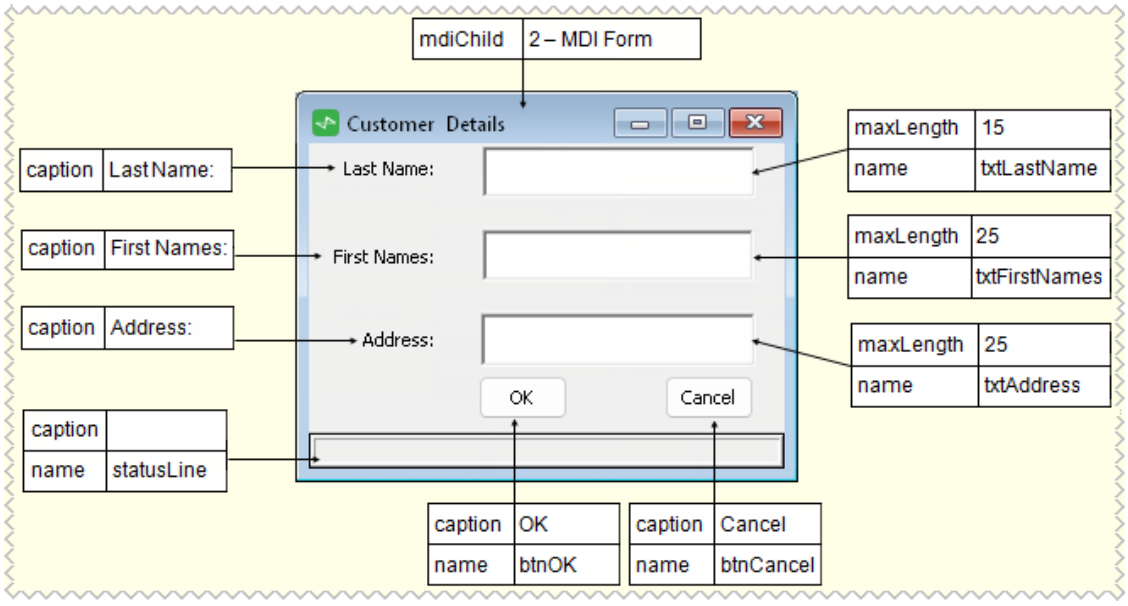
Exercise 11.2 - Adding a CustomerDetailsForm Form

In this exercise, you will create a new form called **CustomerDetailsForm** in the **BankingViewSchema**.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **CustomerDetailsForm** as the name of the form and then select **BVSBaseForm** in the **Sub-Form of** drop-down list.



3. Paint the form, as shown in the following diagram. To set the **mdiChild** property of the form, double-click on an empty part of the form (that is, an area that does not contain an element). The **mdiChild** property is located on the **Specific** sheet of the Properties dialog.



4. Save the form.

Exercise 11.3 - Adding a JadeScript Method to Run a Form

In this exercise, you will add a JadeScript method to display the **CustomerDetails** form.

Note You can run a form from within Painter by selecting the File menu **Run Form** command. However, by using a JadeScript method, you can run the **initialize** method from the **Application** class to set a reference to the root object.

1. Add a JadeScript method called **runForm** in the **BankingViewSchema**.
2. Code the method as follows.

```
runForm();

vars
    customerDetailsForm : CustomerDetailsForm;

begin
    app.initialize();

    create customerDetailsForm transient;

    customerDetailsForm.show();

    // Wait five seconds then close
    app.doWindowEvents( 5000 );

    customerDetailsForm.unloadForm();
end;
```

3. Execute the JadeScript method.

Exercise 11.4 - Coding the CustomerDetailsForm Form

In this exercise, you will code the following method in the **CustomerDetailsForm** form to add an event method to close the form when the **btnCancel** button is clicked.

You will call the protected methods from event methods.

1. In the **CustomerDetailsForm** form, select the **btnCancel** button and then select the **click** event.
2. Code the **click** method as follows.

```
btnCancel_click(btn: Button input) updating;

begin
    self.unloadForm();
end;
```

3. Add a new method called **displayObject** and code the method as follows.

```
displayObject( pCustomer : Customer ) updating, protected;

begin
    inheritMethod( pCustomer );

    if pCustomer = null then
        // Adding new customer
        self.txtLastName.text := null;
        self.txtFirstNames.text := null;
        self.txtAddress.text := null;
    else
        // Editing existing customer
        self.txtLastName.text := pCustomer.lastName;
        self.txtFirstNames.text := pCustomer.firstNames;
        self.txtAddress.text := pCustomer.address;
    endif;
end;
```

4. Add a new method called **getCurrentObject** and code the method as follows.

```
getCurrentObject() : Customer;

begin
    return inheritMethod().Customer;
end;
```

5. Add a new method called **getTA** and code the method as follows.

```
getTA() : CustomerTA;

begin
    return inheritMethod().CustomerTA;
end;
```

6. Add a new method called **getTAClass** and code the method as follows.

```
getTAClass() : Class protected;

begin
    return CustomerTA;
end;
```

7. Add a new method called **populateTAFromForm** and code the method as follows.

```
populateTAFromForm( pTA : CustomerTA input ) : Boolean protected;

begin
    if not inheritMethod( pTA ) then
        return false;
    endif;

    pTA.lastName := self.txtLastName.text;
    pTA.firstNames := self.txtFirstNames.text;
    pTA.address := self.txtAddress.text;

    return true;
end;
```

8. Select the **btnOK** button, and then select the **click** event and code the method as follows.

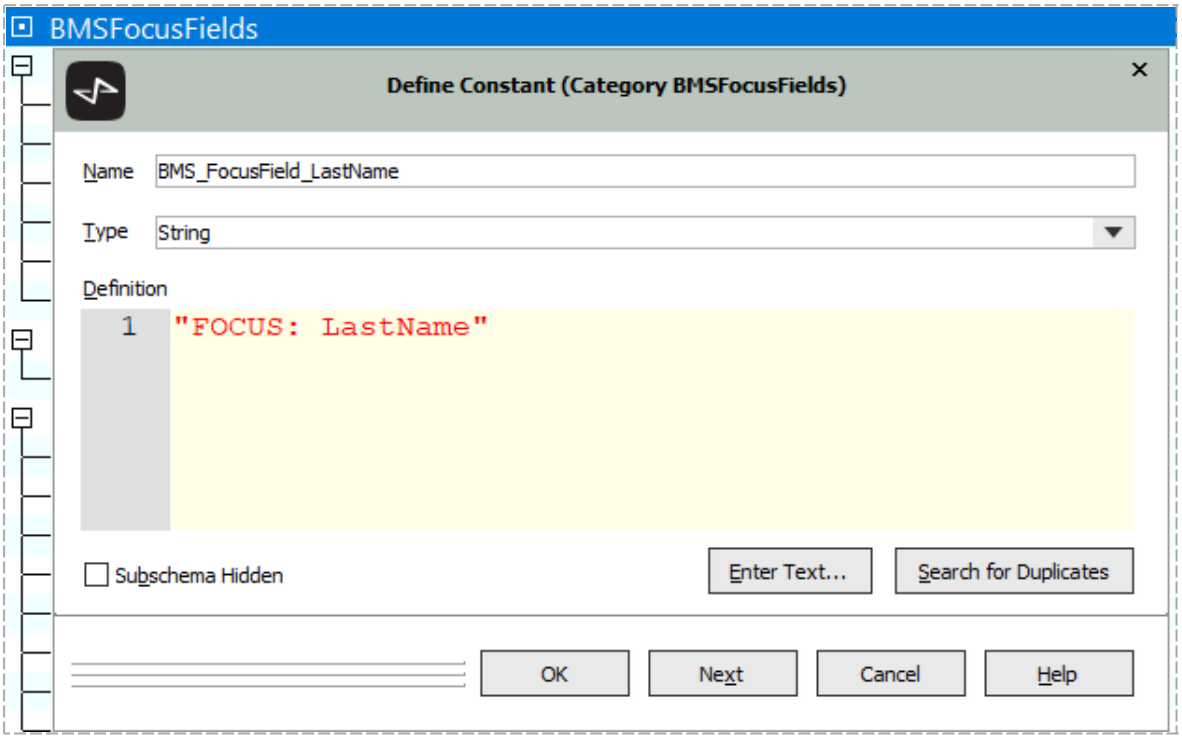
```
btnOK_click(btn: Button input) updating;

vars
    isAddingCustomer : Boolean;

begin
    isAddingCustomer := self.getCurrentObject() = null;

    if not self.doSave() then
        return;
    endif;

    if isAddingCustomer then
        self.statusLine.caption := "Customer added successfully";
        self.displayObject( null );     // Clear the fields ready for the next customer to be added
    else
        self.statusLine.caption := "Customer updated successfully";
    endif;
end;
```

Exercise 11.5 - Implementing Validation Rules for Customer

In this exercise, you will implement validation rules for the **Customer** object.

1. Press Ctrl+G to open the Global Constants Browser.
2. Add a new category called **BMSFocusFields**, using the **Add Category** command from the Category submenu in the Global Constants menu.

3. Add a new global constant called **BMS_FocusField_LastName** of type **String** with a definition of **"FOCUS: LastName"**, using the **Add** command from the Constant submenu in the Global Constants menu.



4. Add a new global constant called **BMS_FocusField_FirstNames** of type **String** with a definition of **"FOCUS: FirstNames"**.

5. Add a new global constant called **BMS_FocusField_Address** of type **String** with a definition of **"FOCUS: Address"**.

6. Select the **CustomerTA** class in a Class Browser opened in the context of the **BankingModelSchema**.

7. Add a new method called **doValidate**, as follows.

```
doValidate( pOperation : Integer ) : Boolean updating;

begin
    inheritMethod( pOperation );

    // We don't want to do this validation for BMS_TransactionType_Delete or BMS_TransactionType_Modify
    if pOperation.isOneOfTheseValues( BMS_TransactionType_Create, BMS_TransactionType_Update ) then
        if self.lastName = null then
            self.addError( "You need to provide a last name", BMS_FocusField_LastName );
        endif;

        if self.firstNames = null then
            self.addError( "You need to provide first names", BMS_FocusField_FirstNames );
        endif;

        if self.address = null then
            self.addError( "You need to provide an address", BMS_FocusField_Address );
        endif;
    endif;

    return self.hasNoErrors();
end;
```

8. Compile the method using F8.

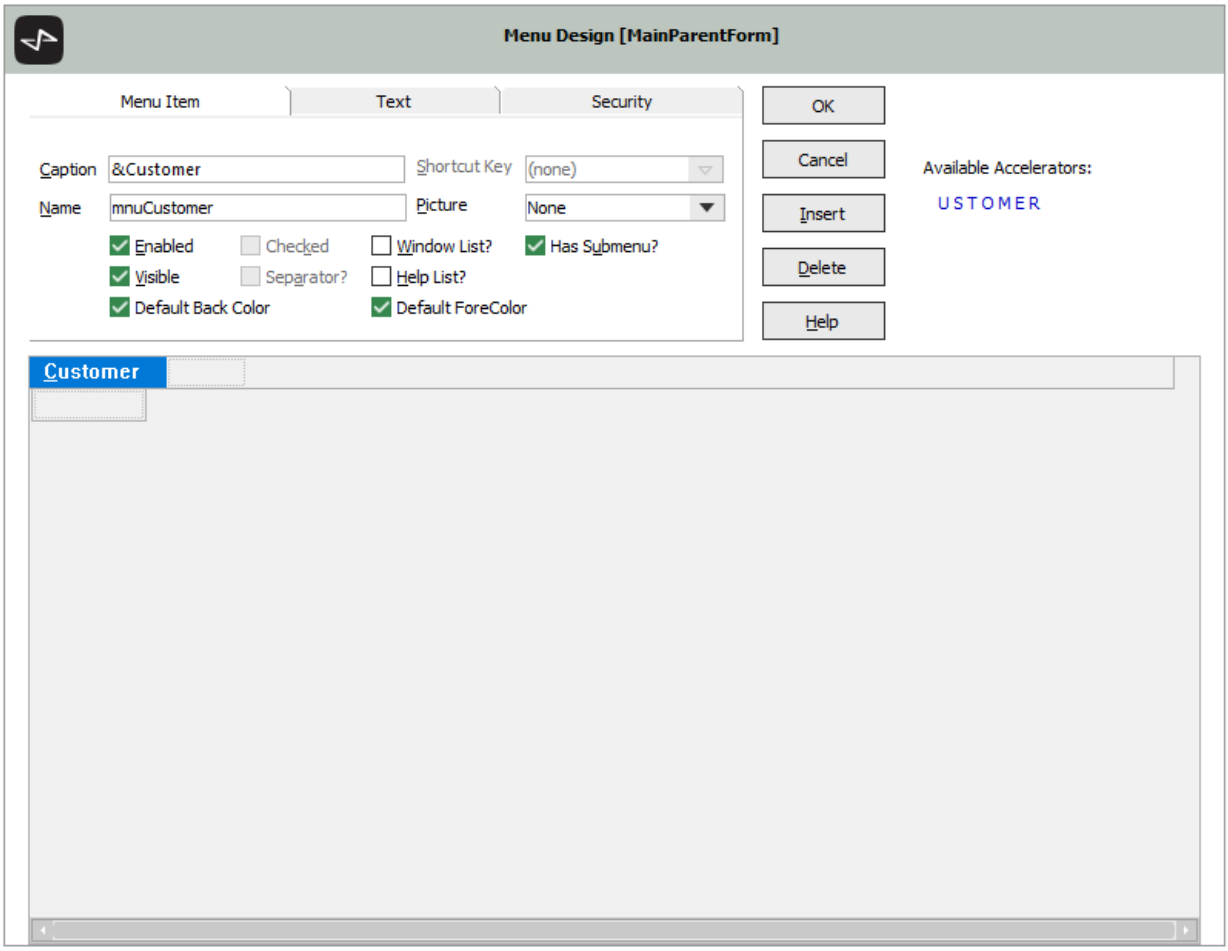
9. Open the JadeScript **runForm** method and comment out the instructions for automatically closing the form.

```
runForm();

vars
    customerDetailsForm : CustomerDetailsForm;

begin
    app.initialize();

    create customerDetailsForm transient;

    customerDetailsForm.show();

    // Wait five seconds then close
    //app.doWindowEvents( 5000 );

    //customerDetailsForm.unloadForm();
end;
```

10. Execute the JadeScript **runForm** method and test that you can add a customer.

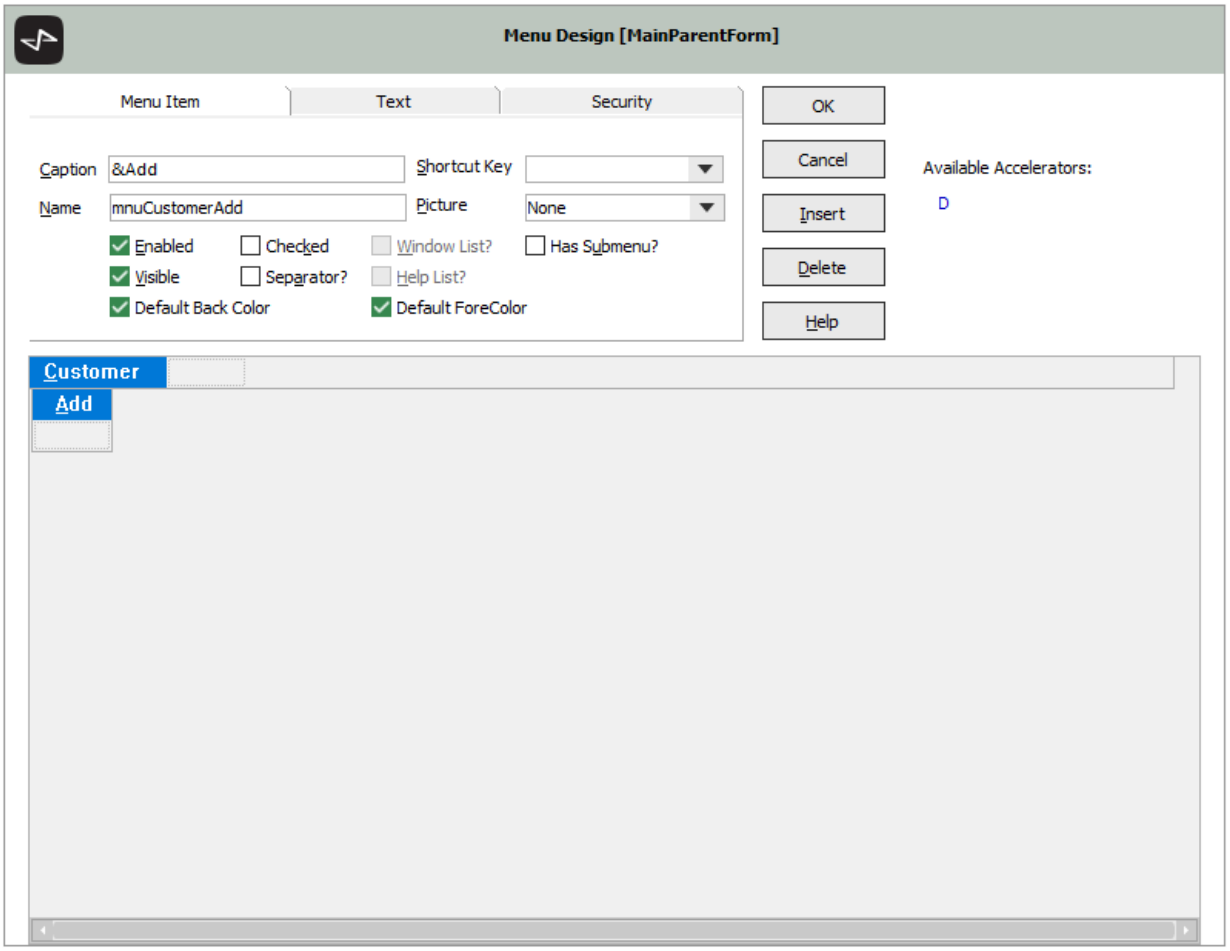
Menus

The menu designer in Painter is accessed by selecting the File menu **Menu Design** command.

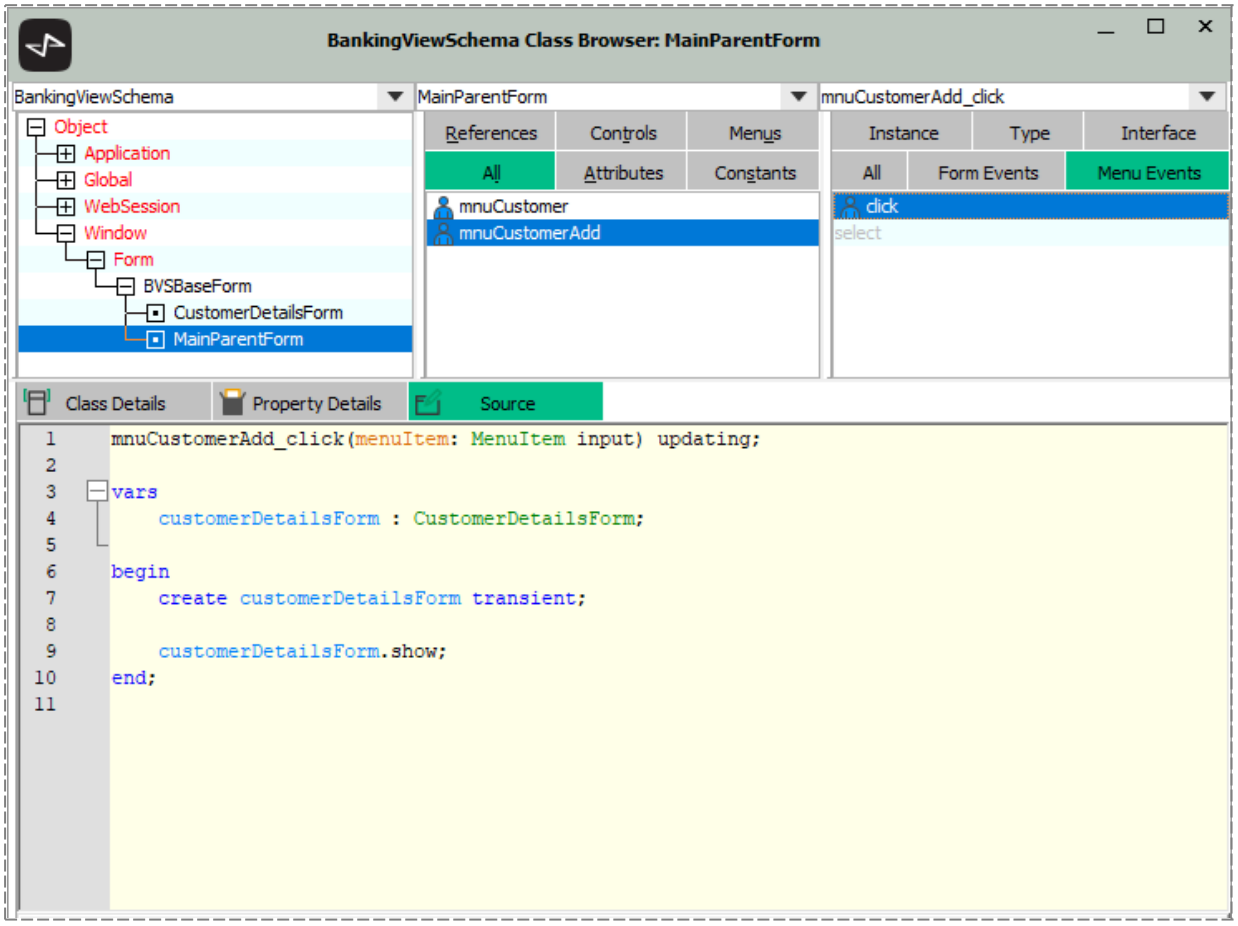


Note An ampersand character (**&**) in the caption causes the character that follows to be underlined. The underlined character becomes an accelerator key when the form is run.

Select a menu item in the designer and then enter values for the **Caption** and **Name**.

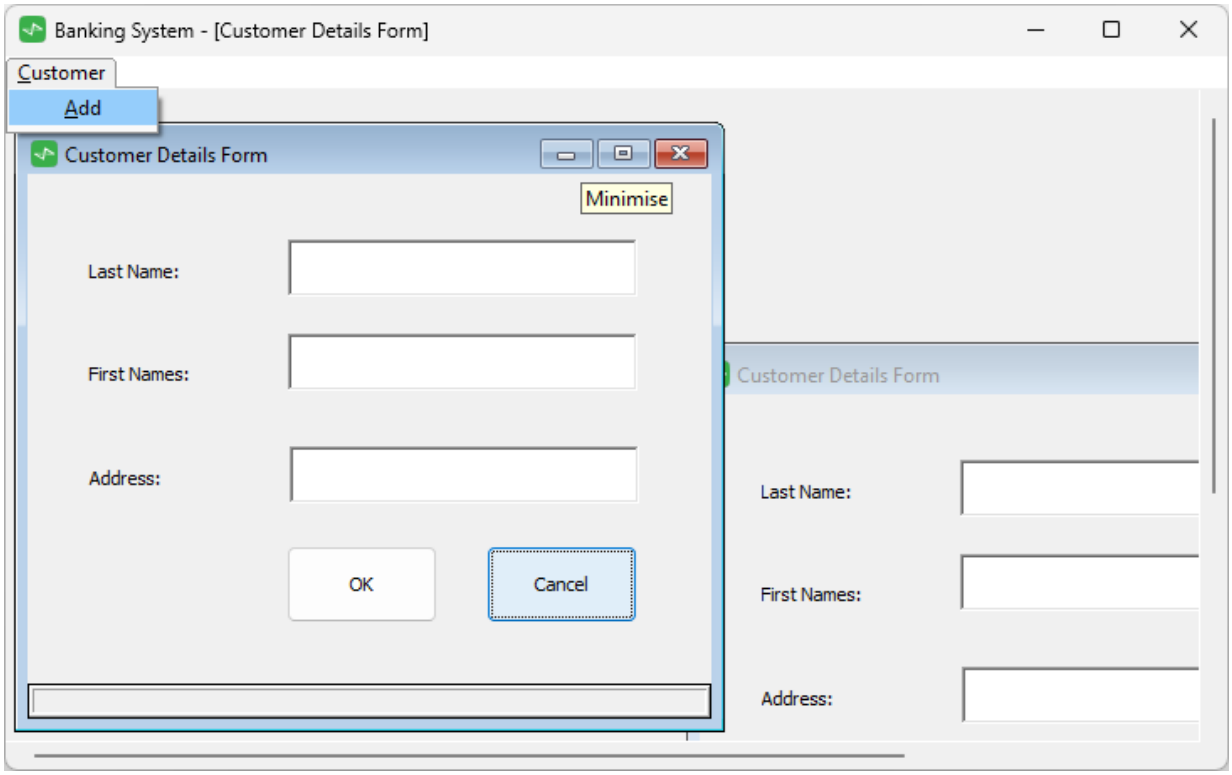


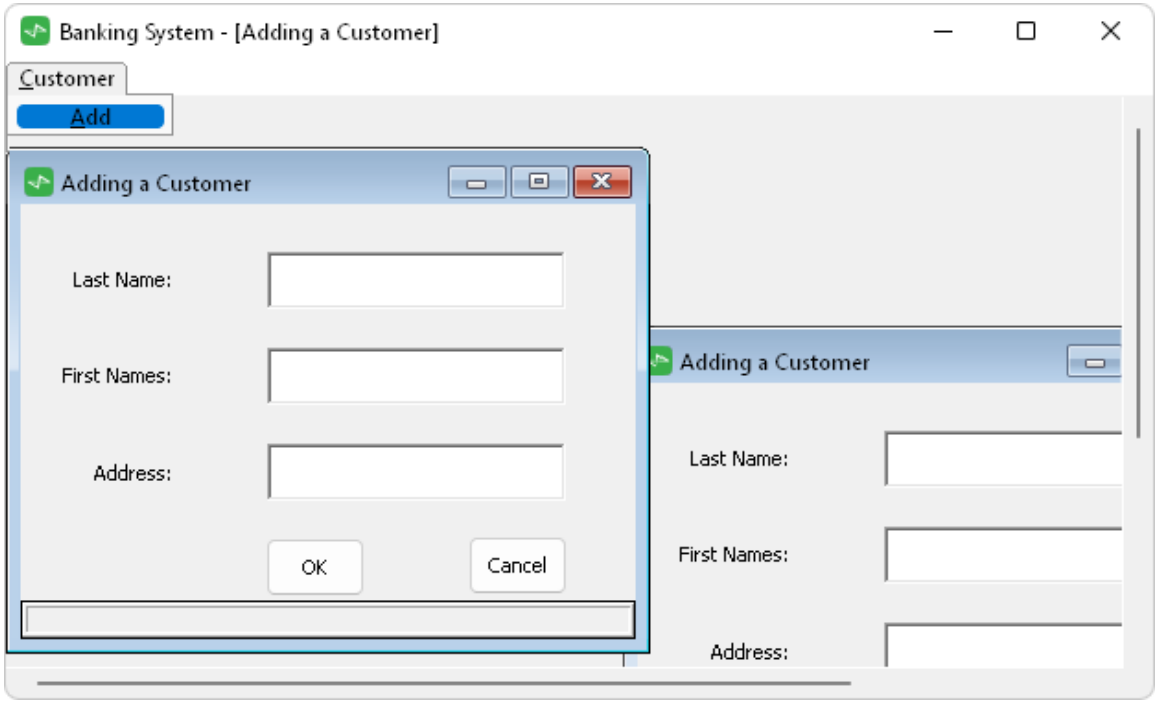
When you save the form and return to the Class Browser, the menu items are displayed in the central Properties List. Select a menu item and then code its **click** event method, as follows.



Multiple Document Interface

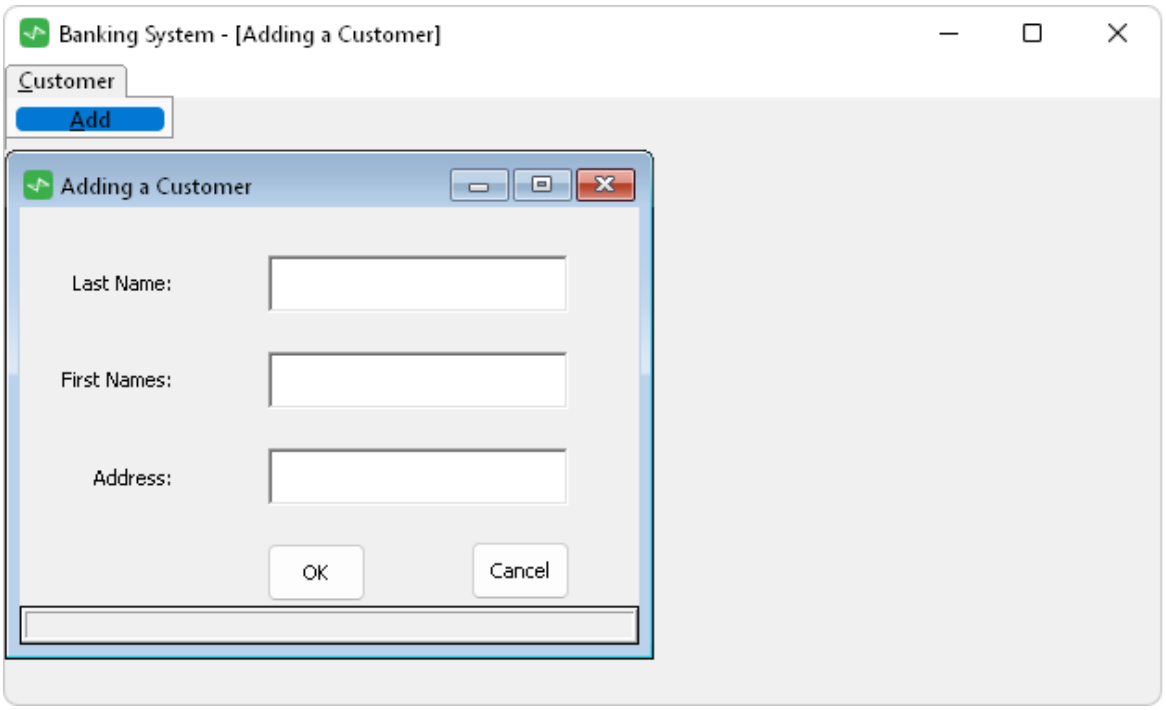
When you ran the **CustomerAdd** form in the previous exercise, it ran as a *multiple document application* (MDI), as shown in the following image.





In a multiple document application, forms are created as *child* windows that are confined within the boundaries of a *parent* window. When you painted the **CustomerDetails** form, you set the **mdiChild** property to make it an MDI child form.

The parent window in an MDI application is called the *MDI frame*. It is a form that is typically painted without any controls but with a menu, as shown in the following image.

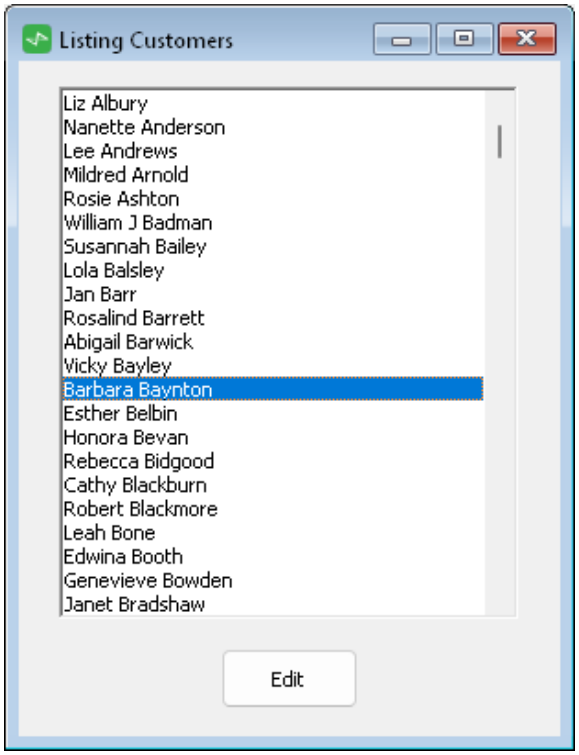


To make a form into an MDI frame, set the **mdiFrame** property to **true** and then add the following instruction when the form is loaded.

```
formLoad() updating, protected;

begin
    app.mdiFrame := MainParentForm;
end;
```

List Boxes

List boxes are used to display collections of objects in an application; for example, the root object's collection of customers.



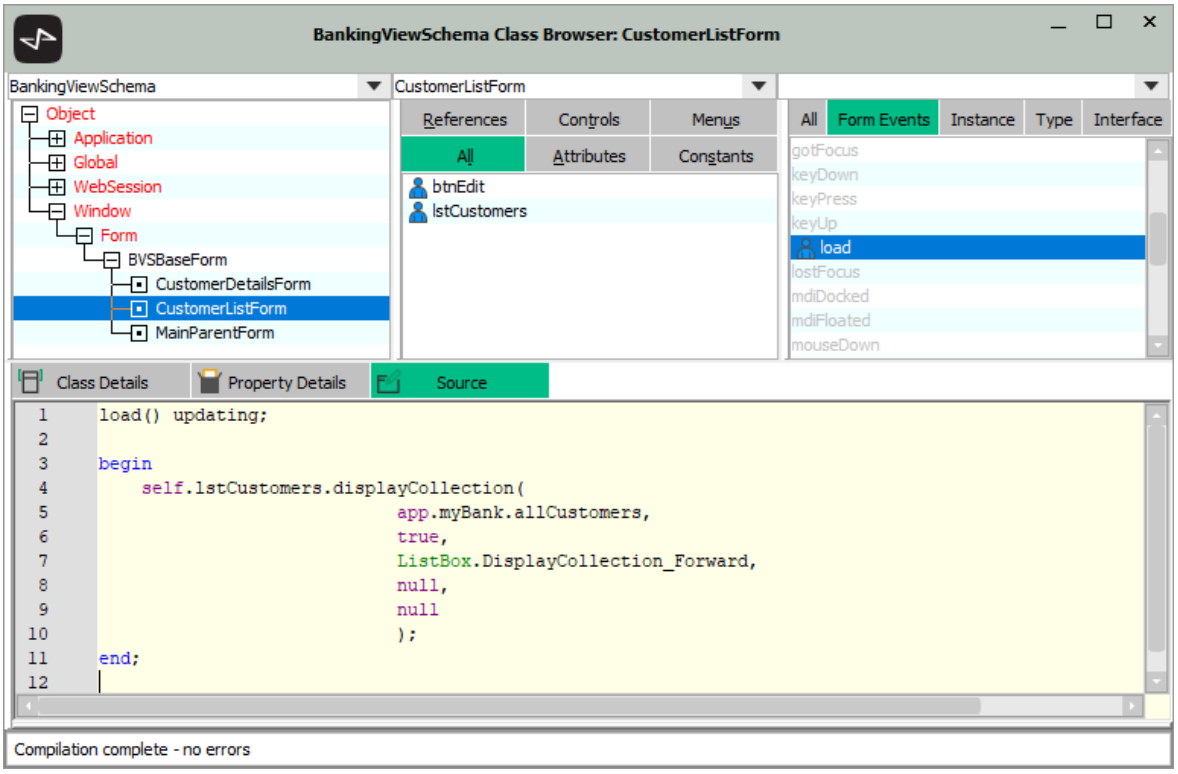
The **ListBox** class provides methods and properties for populating a list box and for determining the customer that the user has selected.

Populating a List Box

A simple and efficient way to populate a list box from a collection is as follows.

1. Associate the collection with the list box by using its **displayCollection** method.

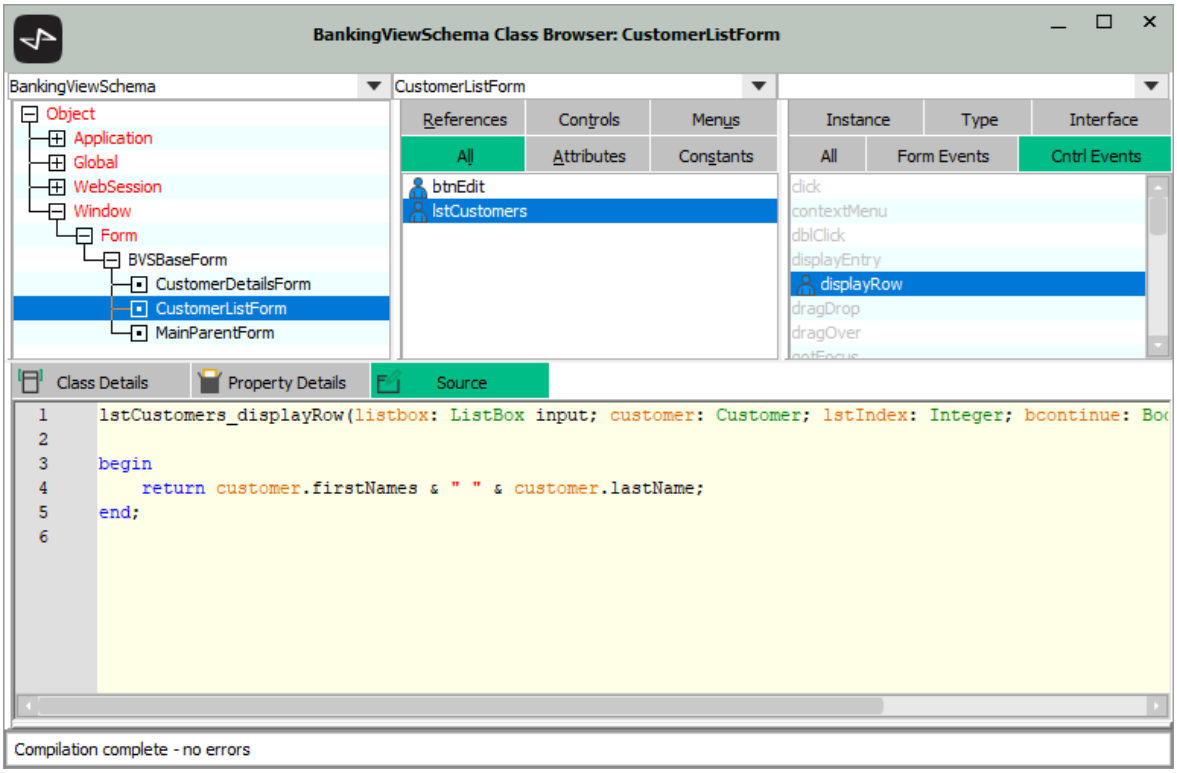
   This is usually done when the form loads.



   The parameters for the **displayCollection** method are:

   - Collection to be used.
   - **true** (the list box automatically refreshes if the collection changes) or **false** (no automatic refreshing).
   - **0** (normal collection order) or **1** (reversed collection order). There are constants on the **ListBox** class with the values **ListBox.DisplayCollection_Forward** and **ListBox.DisplayCollection_Reversed**. (**DisplayCollection_Forward** is not the value but the name, the value is **0**.)
   - Starting object (the list box is scrolled so that this object is at the top).
   - Extra text that is displayed as the first entry in the list box.

2. Specify the text that is displayed for each object. This is coded in the **displayRow** event method of the list box, which is called for each object in the visible part of the list box.



Note If the list box displays 15 objects at a time, the **displayRow** method is called 15 times only when the form is loaded. Subsequent scrolling results in the method being called for the next 15 customers.

Alternatively, you can add objects to a list box one at a time, by using the **addItem** method and the **itemObject** array, as shown in the following example.

```
foreach customer in app.myBank.allCustomers do
    newIndex := self.lstCustomers.addItem( customer.firstNames & " " & customer.lastName );
    self.lstCustomers.itemObject[ newIndex ] := customer;
endforeach;
```

Determining the Selected Object

When a user selects an entry in a list box, the **listIndex** property is set to that row number. If the first entry is selected, the value of **listIndex** is **1**, and if no entry is selected, the value of **listIndex** is **-1**.

The customer selected in a list box can be obtained from the **itemObject** array, as follows

```
customer := self.lstCustomers.itemObject[ self.lstCustomers.listIndex ].Customer;
```

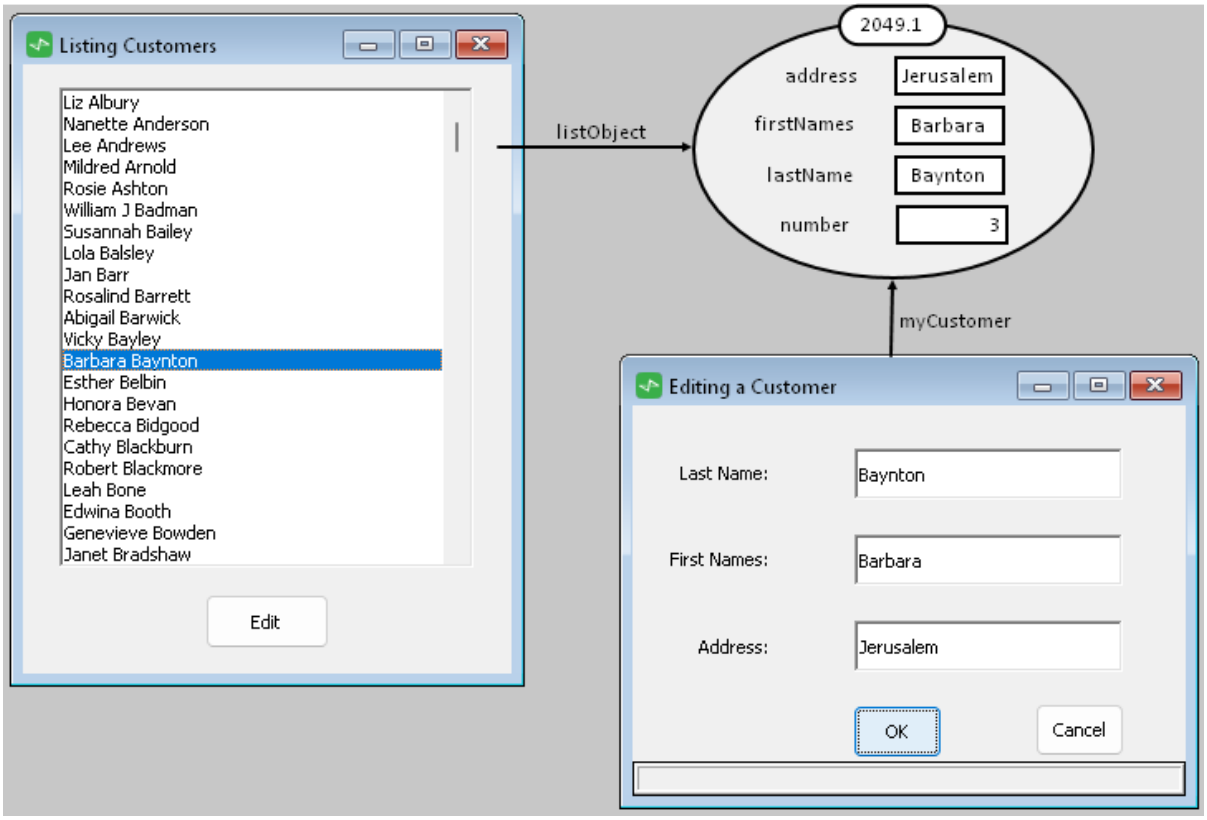
You can achieve the same result by using the **listObject** property, as follows.

```
customer := self.lstCustomers.listObject.Customer;
```

Editing a Customer

In the application, a customer to be edited is selected in the list box and stored in the **listObject** property. When the **Edit** button is clicked, a **CustomerDetailsForm** form is created.

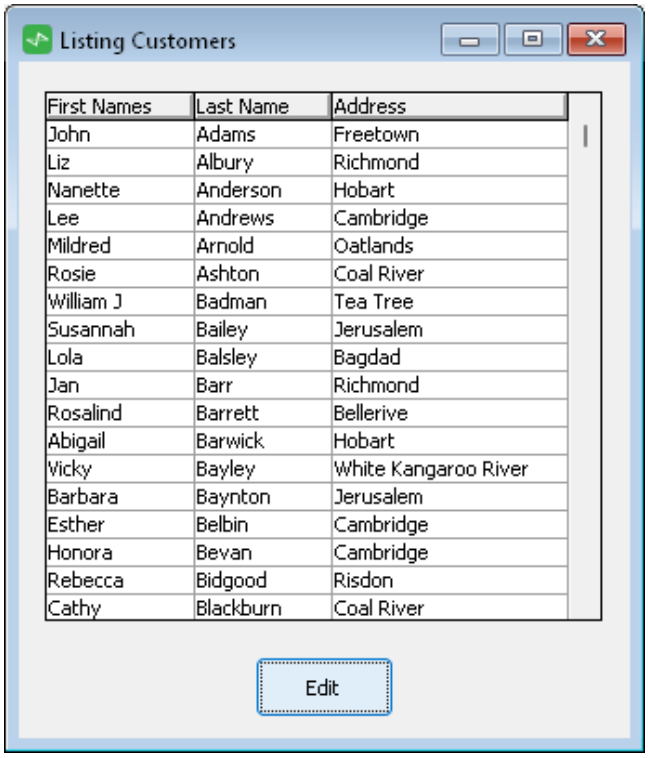
The **CustomerDetailsForm** inherits the **myCurrentObject** reference from the **BVSBaseForm**, which identifies the **Customer** object whose details are loaded into the text boxes.



When the customer details are changed, the Transaction Agent Framework (TAF) will be used to update the properties of the **Customer** object.

Tables

A table can display objects in a collection, using a number of columns.



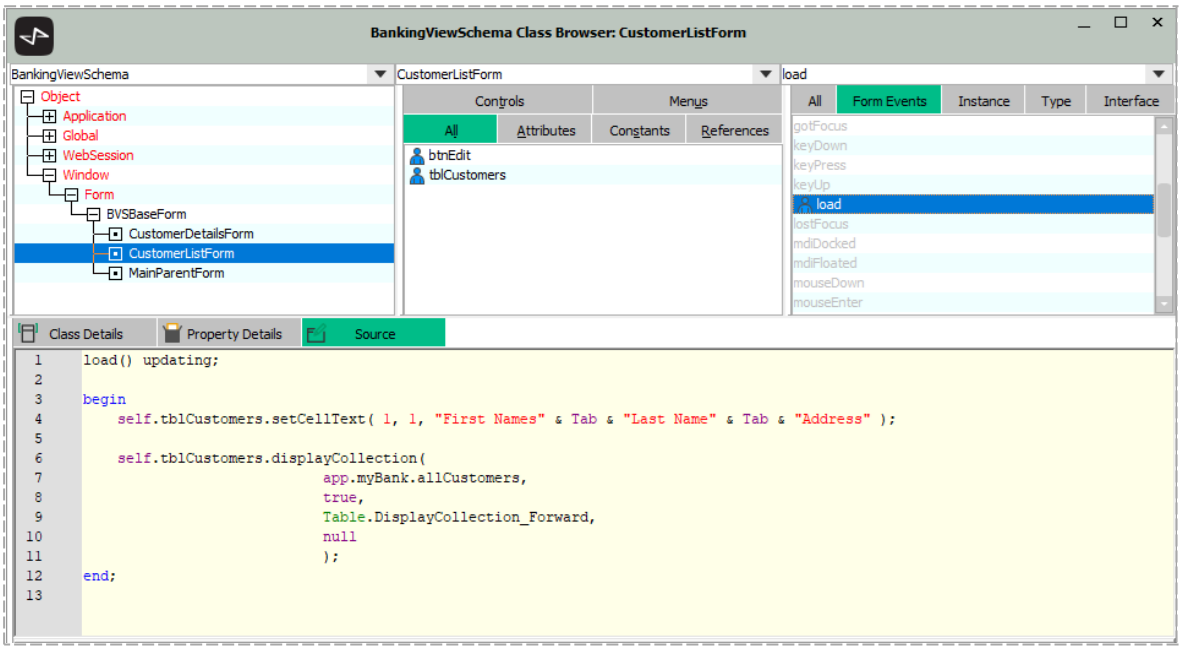
The **Table** class provides similar methods and properties to the **ListBox** class for populating a table and for determining the customer that the user has selected.

Populating a Table

A simple and efficient way to populate a table from a collection is:

1. Associate the collection with the table using its **displayCollection** method.

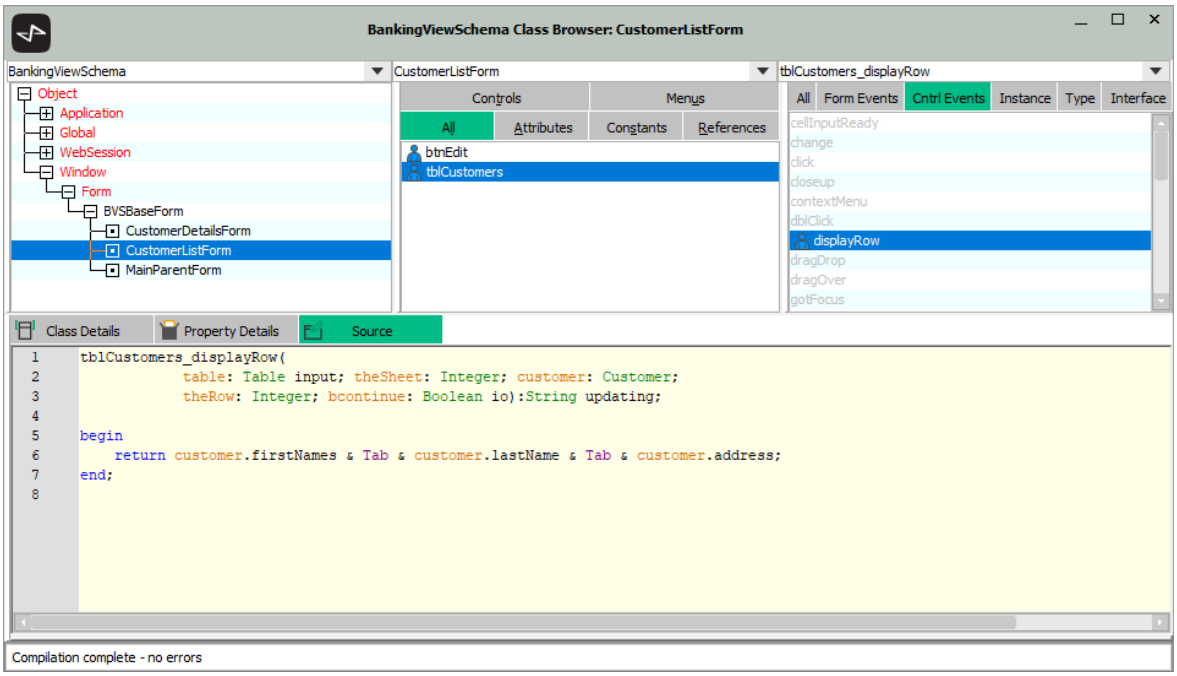
   This is usually done when the form loads.



The parameters for the **displayCollection** method are:

- Collection to be used.
- **true** (table automatically refreshes if the collection changes) or **false** (no automatic refreshing).
- **0** (normal collection order) or **1** (reversed collection order). There are constants on the **Table** class to use for this parameter, **Table.DisplayCollection_Forward** has the value **0** and **Table.DisplayCollection_Reversed** has the value **1**.
- Starting object (table is scrolled so that this object is at the top).

2. Specify the text that is displayed for each object. This is coded in the **displayRow** event method of the table, which is called for each object in the visible part of the table.



Alternatively, you can add objects to a table one at a time, by using the **addItem** method and the **itemObject** of an associated **JadeTableRow** object, as shown in the following example.

```
foreach customer in app.myBank.allCustomers as Customer do
    newRow := self.tblCustomers.addItem(
                            customer.firstNames & Tab &
                            customer.lastName & Tab &
                            customer.address
                            );
    self.tblCustomers.accessRow( newRow ).itemObject := customer;
endforeach;
```

Determining the Selected Object

When a user selects an entry in a table, the **row** property is set to that row number. If the first entry is selected, the value of **row** is **1**, which often contains column headings.

The customer selected in a table can be obtained from the **itemObject** property of the **JadeTableRow** object for the selected row, as follows.

```
customer := self.tblCustomers.accessRow( self.tblCustomers.row ).itemObject.Customer;
```

Exercise 11.6 - Adding a MainParentForm Form

In this exercise, you will add a form with a menu and make the form the MDI frame.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **MainParentForm** as the name of the form and select **BVSBaseForm** in the **Sub-Form of** drop-down list.
3. In the **specific** group of the Properties dialog, set the **mdiFrame** property of the form to **True**.
4. Set the **caption** property for the form to **Banking System**.
5. Save the form.
6. Return to the Class Browser.
7. Select the **MainParentForm** and add a new method called **formLoad**. Click **Yes** to acknowledge any warning messages about reimplementing a method.
8. Code the method as follows.

```
formLoad() updating, protected;

begin
    app.mdiFrame := MainParentForm;
end;
```

9. Return to the Painter and then open the menu designer by selecting the File menu **Menu Design** command.
10. For the first menu, enter **&Customer** in the **Caption** field and **mnuCustomer** in the **Name** text box.
11. Select the first menu item under the **Customer** menu and then enter **&Add** in the **Caption** text box and **mnuCustomerAdd** in the **Name** text box.
12. Click the **OK** button to close the menu designer, and then save the form.
13. In the Class Browser, select the **mnuCustomerAdd** menu item and then select the **click** event method.
14. Code the method as follows.

```
mnuCustomerAdd_click(menuItem: MenuItem input) updating;

vars
    customerDetailsForm : CustomerDetailsForm;

begin
    create customerDetailsForm transient;

    customerDetailsForm.show();
end;
```

15. Change the JadeScript **runForm** method to open **MainParentForm** instead of **CustomerDetailsForm**.

16. Execute the JadeScript **runForm** method and test the MDI parent-child functionality.

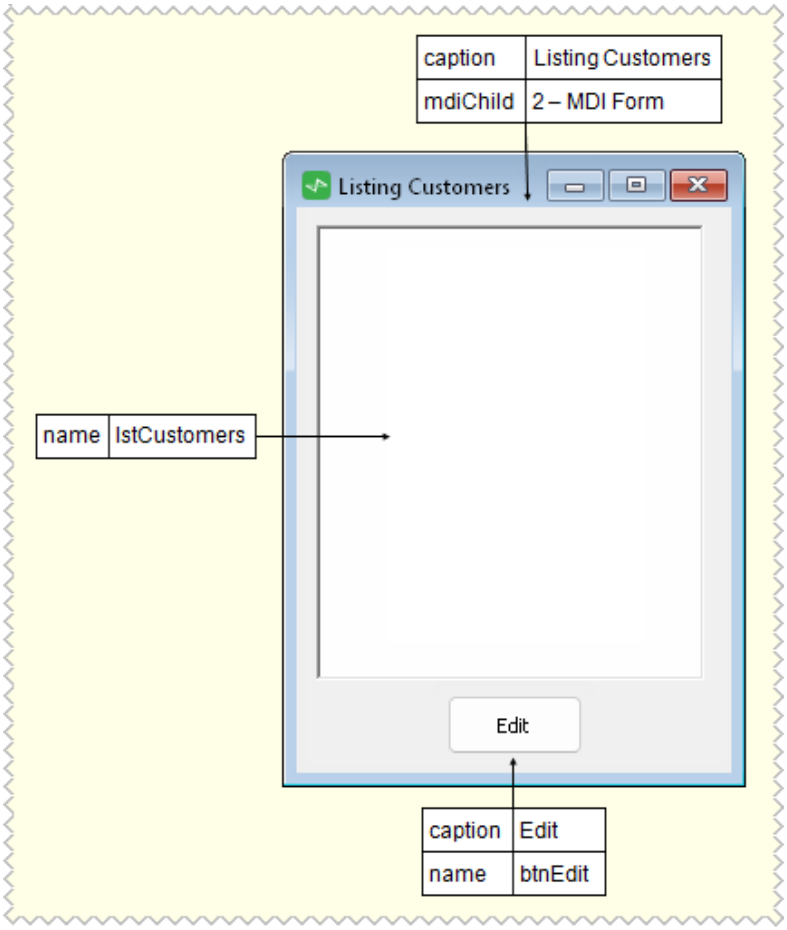
```
runForm();

vars
    mainParentForm : MainParentForm;

begin
    app.initialize();

    create mainParentForm transient;

    mainParentForm.show();
end;
```

Exercise 11.7 - Adding a CustomerListForm Form

In this exercise, you will add a **CustomerListForm** form that will display the root object's collection of customers. You will then add an option to the Customer menu on the **MainParentForm** form to open the **CustomerListForm** form.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **CustomerListForm** as the name of the form, and select **BVSBaseForm** in the **Sub-Form of** drop-down list.

3. Paint the form with a list box and a button, as shown in the following diagram.



4. Save the form and then return to the Class Browser.

5. Select the **CustomerListForm** in the Class Browser, add a new method called **formLoad**, and then code the method as follows.

```
formLoad() updating, protected;

begin
    inheritMethod();

    self.lstCustomers.displayCollection(
                                app.myBank.allCustomersByLastName,
                                true,
                                ListBox.DisplayCollection_Forward,
                                null,
                                null
                                );
end;
```

6. Select the **lstCustomers** list box, and then select the **displayRow** event.

7. Code the **displayRow** method as follows.

```
lstCustomers_displayRow(
        listbox : ListBox input;
        customer : Customer;
        lstIndex : Integer;
        bcontinue : Boolean io
        ) : String updating;

begin
    return customer.firstNames & " " & customer.lastName;
end;
```

8. Select the **btnEdit** button, and then select the **click** event.

   a. Code the **click** event method to write the last name of the selected customer. (You will change this method in a later exercise.)

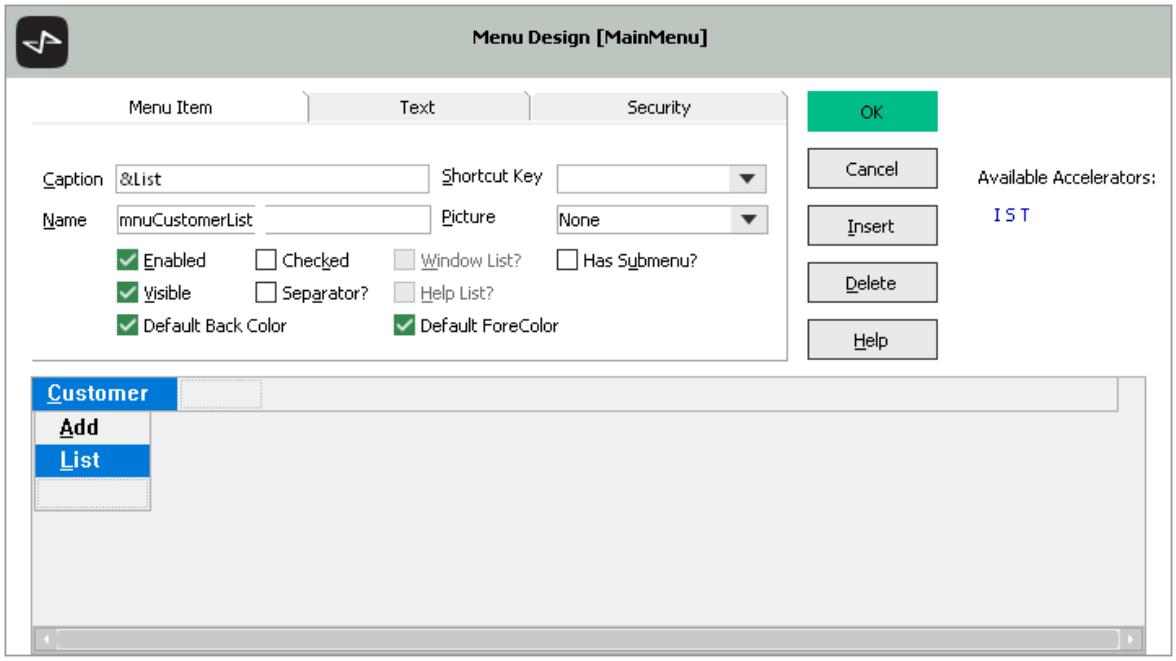
```
btnEdit_click(btn: Button input) updating;

vars
    customer : Customer;

begin
    customer := self.lstCustomers.listObject.Customer;

    if customer = null then
        app.msgBox( "Select a customer first", "Error", MsgBox_OK_Only );
    else
        write customer.lastName;
    endif;
end;
```

9. Open the **MainParentForm** form in Painter.

10. Open the menu designer by selecting the File menu **Menu Design** command.

Tip When you already have a visible menu, you can click on that menu in Painter to quickly open the menu designer.

11. Select the cell below the **Add** menu, and then enter **&List** in the **Caption** text box and **mnuCustomerList** in the **Name** text box.



12. Click the **OK** button to close the menu designer, and then save the form.

13. In the Class Browser, select the **mnuCustomerList** menu item and then select the **click** event method.

14. Code the method as follows.

```
mnuCustomerList_click(menuItem: MenuItem input) updating;

vars
    customerListForm : CustomerListForm;

begin
    create customerListForm transient;

    customerListForm.show();
end;
```

15. Execute the **runForm** JadeScript method and open the **CustomerListForm** form.

16. Test that the **btnEdit** button writes the correct message.

Exercise 11.8 - Editing an Existing Customer

In this exercise, you will add the logic to open the **CustomerDetailsForm** form to edit an existing customer.

1. In the **CustomerListForm** form, change the **click** method of the **Edit** button to open the **CustomerDetailsForm** form and set the current object reference, as follows.

```
btnEdit_click(btn: Button input) updating;

vars
    customer : Customer;
    customerDetailsForm : CustomerDetailsForm;

begin
    customer := self.lstCustomers.listObject.Customer;

    if customer = null then
        app.msgBox( "Select a customer first", "Error", MsgBox_OK_Only );
    else
        create customerDetailsForm transient;

        customerDetailsForm.setContextObject( customer );

        customerDetailsForm.show();
    endif;
end;
```

2. Execute the JadeScript **runForm** method and then open the **CustomerListForm** form.
3. Select the customer **Barbara Baynton** and change the name to **Barbara Jackson**, by clicking the **Edit** button.

   Does the list box on the **CustomerListForm** form update? Why?
4. On the **CustomerListForm** form, select the customer **Barbara Jackson** and change the name to **Alice Jackson**, by clicking the **Edit** button.

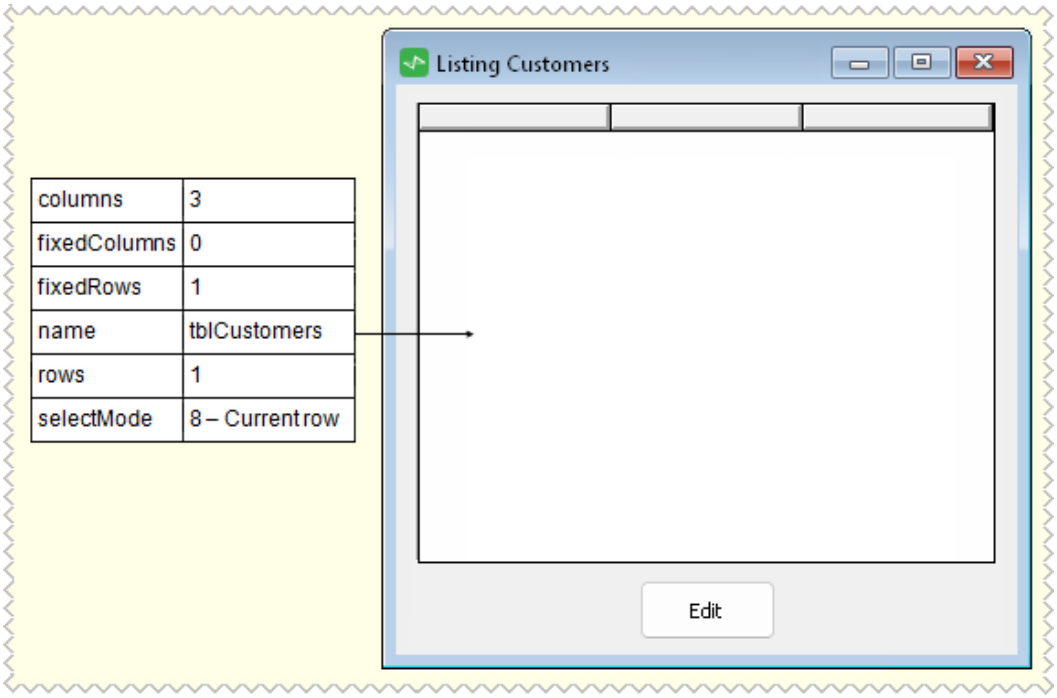
   Does the list box on the **CustomerListForm** form update? Why?

Exercise 11.9 - Changing the CustomerListForm Form

In this exercise, you will change the **CustomerListForm** form to use a table instead of a list box.

1. Open the JADE Painter.
2. Select the File menu **Edit Form** command, select **CustomerListForm**, and then click the **OK** button.

3. Replace the list box with a table, as shown in the following diagram.



4. Save the form and then return to the Class Browser.
5. Select the **formLoad** method for the **CustomerListForm** form.
   a. Replace the code, as follows.

```
formLoad() updating, protected;

begin
    inheritMethod();

//  self.lstCustomers.displayCollection(
//                              app.myBank.allCustomersByLastName,
//                              true,
//                              ListBox.DisplayCollection_Forward,
//                              null,
//                              null
//                              );

    self.tblCustomers.setCellText( 1, 1, "First Names" & Tab & "Last Name" & Tab & "Address" );

    self.tblCustomers.displayCollection(
                              app.myBank.allCustomersByLastName,
                              true,
                              Table.DisplayCollection_Forward,
                              null
                              );

end;
```

6. Select the **tblCustomers** table, and then select the **displayRow** event.

7. Code the **displayRow** method as follows.

```
tblCustomers_displayRow(
        table : Table input;
        theSheet : Integer;
        customer : Customer;
        theRow : Integer;
        bcontinue : Boolean io
        ) : String updating;

begin
    return customer.firstNames & Tab & customer.lastName & Tab & customer.address;
end;
```

Note Make sure to change the **obj: Object** parameter to **customer: Customer**.

8. Select the **btnEdit** button, and then select the **click** event.

9. Replace the code in the **click** method, as follows.

```
btnEdit_click(btn: Button input) updating;

vars
    customer : Customer;
    customerDetailsForm : CustomerDetailsForm;

begin
    //customer := self.lstCustomers.listObject.Customer;

    // Make sure the current row for the table is a valid row before attempting to access the itemObject for that row
    if self.tblCustomers.row > self.tblCustomers.fixedRows and self.tblCustomers.row <= self.tblCustomers.rows then
        customer := self.tblCustomers.accessRow( self.tblCustomers.row ).itemObject.Customer;
    endif;

    if customer = null then
        app.msgBox( "Select a customer first", "Error", MsgBox_OK_Only );
    else
        create customerDetailsForm transient;

        customerDetailsForm.setContextObject( customer );

        customerDetailsForm.show();
    endif;
end;
```

10. Test that the **CustomerListForm** form works correctly.

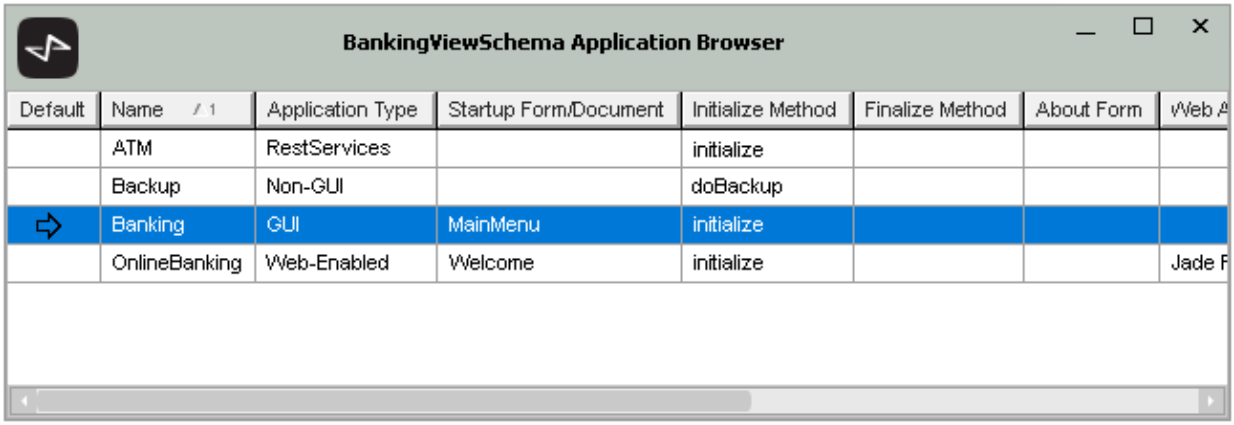
Module 12 Applications

This module contains the following topics.

Introduction

Applications are defined from the Application Browser, which is opened by clicking the **A** button (Browse Applications) from the Jade Platform development environment toolbar or by using the Ctrl+L shortcut keys. The Application Browser for the current selected schema is then displayed.

In the banking system, there are many types of users: customers using online banking, customers using ATMs, tellers working in a branch of the bank, the bank manager, and so on. There would be applications appropriate for different types of users, as well as utility and background applications, as shown in the following image.

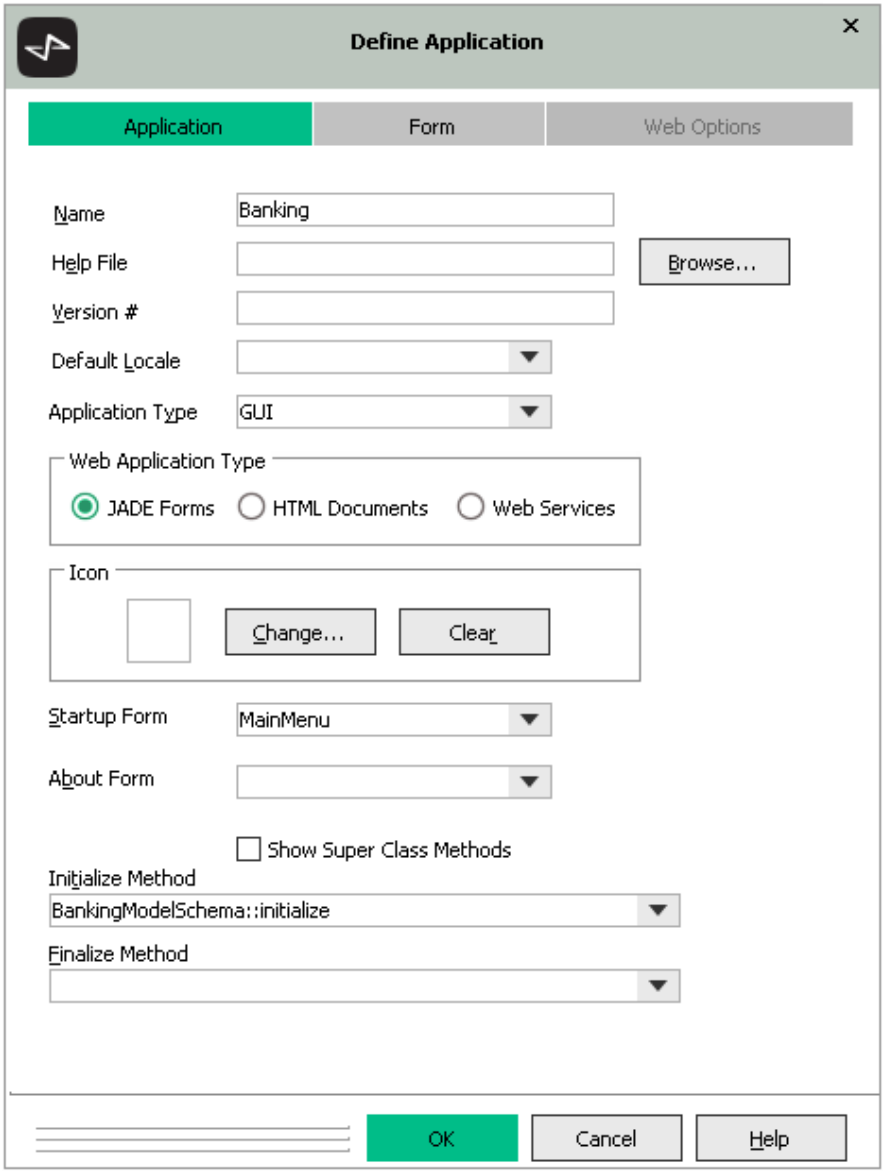


You can select an application in the Application Browser and set it as the *default* application, by using the Application menu **Set** command in the development environment background window.

You can start the default application by right-clicking the arrow button (Run Application) in the Jade Platform development environment background window toolbar.

Defining a GUI Application

In the Application Browser, you can select the Application menu **Add** or **Change** command to display the Define Application dialog, as shown in the following image.



After specifying a name for the application, select an application type.

The **GUI** application type is a standard desktop application, which displays forms that were designed in the JADE Painter. The other application types are:

- **GUI, No Forms** – an application that does not display forms on screen, but can print forms; for example, a print server that prints reports in the background.
- **Non-GUI** – an application that does not create screen or print forms; for example, a program that runs a scheduled backup.
- **Rest Services** – an application that provides REST-based web services, and displays requests from clients in a monitor window. A **Rest Services, Non-Gui** application does not display a monitor window.

- **Web-Enabled** – services browser clients running an application or requesting SOAP-based web services. A monitor window displays client requests.
  - **Jade Forms** – an application accessed from a browser. It uses forms designed in the JADE Painter. At run time, HTML generated by the application is sent from a Microsoft IIS or Apache web server.
  - **HTML Documents** – an application accessed from a browser. It uses forms designed outside the Jade Platform, which are then imported.
  - **Web Services** – an application that provides SOAP-based web services.

  A **Web-Enabled, Non-GUI** application does not display a monitor window.

The **Startup Form** combo box value is the form in the current schema (or a superschema) that is displayed when the application starts.

The method specified in the **Initialize Method** combo box is executed when the application starts before the startup form is displayed. The method specified in the **Finalize Method** combo box is executed when the application terminates. These methods must be defined in the **Application** subclass in the current schema (or a superschema).

Note Methods called **initialize** and **finalize** are used as the **Initialize Method** and **Finalize Method** if they exist and if no other method is specified.

Web Services and REST Services

Any computing device that can run a web browser can connect to a Jade web application. The application creates a session object with a unique session id for the web browser client, and includes the session id on every form that is sent to, and every reply that is received from, a web browser.

Web services can be exported from the providing system and imported into the consuming system using Web Services Description Language (WSDL). Many languages, including Jade and .NET, support web services. When a request arrives from a web browser, the Microsoft Internet Information Server (IIS) passes the request to the Jade web application using **jadehttp.dll** and the Transmission Control Protocol (TCP) connection information in the **jadehttp.ini** file.

The query string contains the name of the Jade web-enabled application, in the following format.

```
http://localhost/jade/jadehttp.dll?WebShop
<-URL path to jadehttp on server->?<-app->
```

The Jade web application processes this request and generates an HTML page in response. Because all communications are asynchronous, the Jade client can monitor and display system processing status when idle.

Windows provides security; standard IIS security for data access and Secure Sockets Layer for data transmission.

If an unhandled Jade exception occurs, it is logged on the web server machine and the operation is aborted.

The same architecture applies to all types of Jade web-enabled application.

- Jade forms, where the forms are designed in the Jade Painter
- HTML forms, where the forms are designed in an external HTML editor; for example, Dreamweaver
- Web services
- REST services

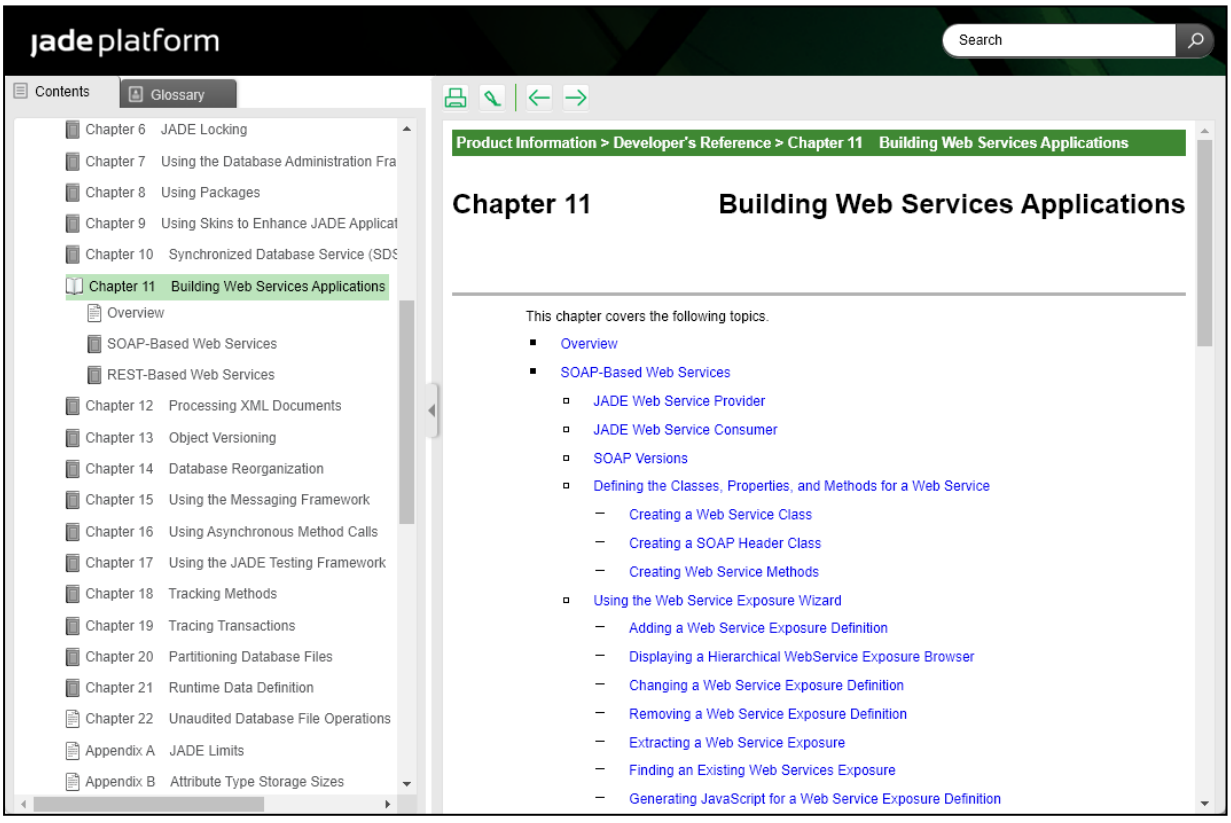
A web service usually uses HTTP to exchange data. Unlike a web application, which is typically HTML over HTTP, a web service is Extensible Markup Language (XML) over HTTP. A client sends a request in XML, and the server responds with an XML response. This XML can be Plain Old XML (POX), which is typically a non-standard XML that only the client and server can make sense of, or it is standard Simple Object Access Protocol (SOAP).

A Representational State Transfer (REST) Application Programming Interface (API) is a web service. A REST API differs from SOAP-based web services in the manner in which it is intended to be used. By using REST, the API tends to be lightweight and embraces HTTP. For example, a REST API leverages HTTP methods to present the actions a user would like to perform and the application entities would become resources on which these HTTP methods can act. Although SOAP is not used, messages (requests and responses) are either in XML or JavaScript Object Notation (JSON).

The **JadeJson** class, which is a transient-only **Object** subclass, provides standalone JSON functionality that is independent of the Representational State Transfer (REST) Application Programming Interface (API). The **JadeJson** class enables you to create, load, unload, and parse JSON in the same way you can with XML.

Although web services and REST services are not covered in depth in this course, the Jade Platform product information library provides you with resources that enable you to develop web service and REST service applications.

The following image shows the Jade Platform HTML5 contents pane in a browser with the "Building Web Services Applications" chapter of the *Developer's Reference* expanded in the **Contents** pane at the left.



For details about the location in HTML5 format of this web services application chapter that covers using both SOAP and REST-based web services, the web services white papers, and the REST services white paper in the Jade Platform product information library, see:

https://secure.jadeworld.com/developer-centre/Jade2025/OnlineDocumentation/

In addition, you can download the:

- PDF (print) format of the *Developer's Reference* from the **Development Environment** section of the **Jade Platform 2025** at https://www.jadeplatform.com/developer-centre/learn/documentation

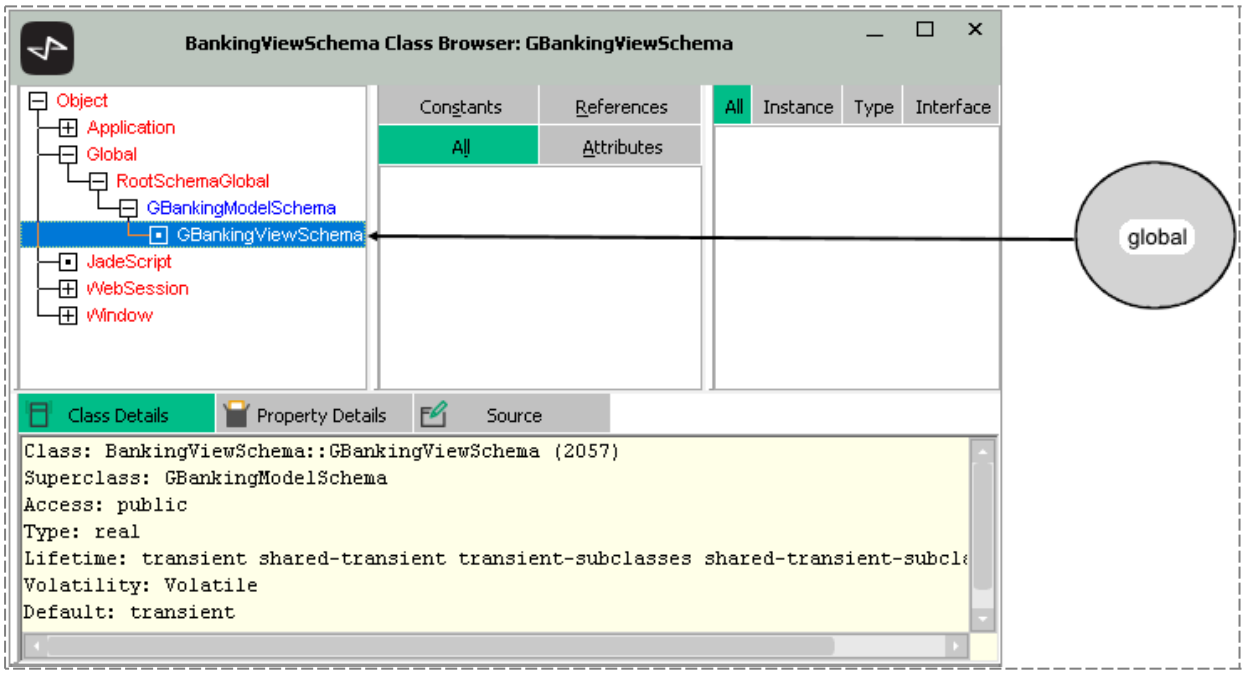
- Web services white papers, which include SOAP Web Services and REST Services, in print (PDF) format from **White Papers** in the **Resource Library** section of **Developer-Center** at https://www.jadeplatform.com/developer-centre/learn/whitepapers

Tip As the HTML5 format of the Jade Platform 2025 product information library contains not only the product information but the white papers and the *Erewhon Demonstration System Reference*, you can search the complete product information library. See the "Search and Print Tips for HTML5 Help" topic in the **Contents** pane at the left of your browser, for more details.

Logon Authentication

When you add a schema, a number of classes are created. One of these is a subclass of **Global**. The name of the subclass is the schema name prefixed with the letter **G**. A single persistent instance of this class is created. It can be referred to in your code by using the system variable **global**. However, when viewing the properties of an object whose class persistence lifetime is set to **Transient**, the **Allow Persistent Instances** check box is unchecked.

The **global** object inherits a lot of useful functionality, including logon validation methods, from the **Global** class.

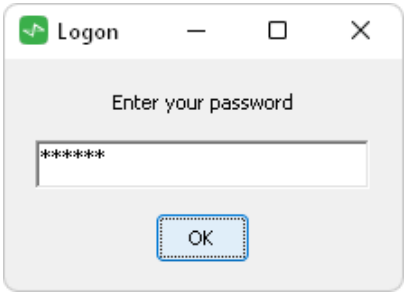


When an application starts, the **getAndValidateUser** method from the **Global** class is executed before anything else in the application happens, including the display of the startup form.

```
getAndValidateUser(usercode: String output; password: String output): Boolean;
```

The **getAndValidateUser** method is a **Boolean** method that returns **true** in the implementation in the **Global** class. If the method returns **true**, the application is allowed to continue. If the method returns **false**, the application is terminated.

You can reimplement the **getAndValidateUser** method in your **Global** subclass to return **true** only if the user authenticates himself or herself by entering the correct password on a logon form.



There is another method on the **Global** class, which is called **isUserValid**. This method is called immediately after the **getAndValidateUser** method, to provide secondary validation on the database server. The **usercode** and **password** parameters are set in the **getAndValidateUser** method. The default implementation returns **true**.

```
isUserValid(usercode: String; password: String): Boolean;
```

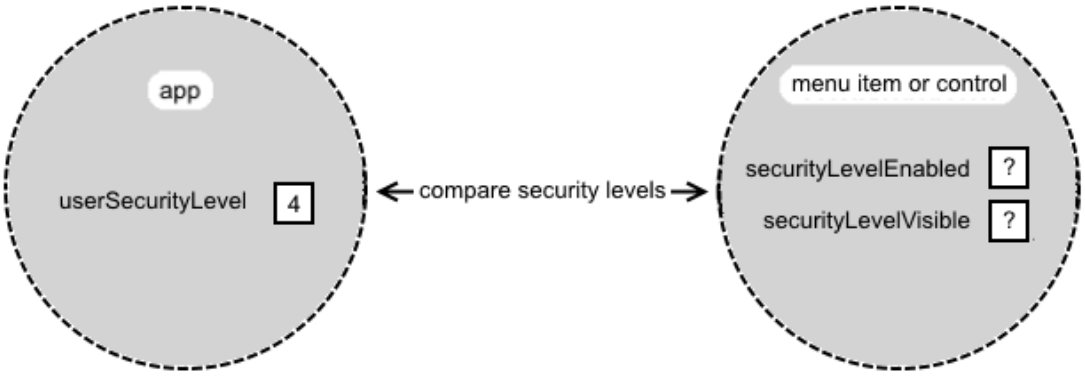
Application Security

You can implement basic security by setting the **userSecurityLevel** attribute on the **app** object. This is usually done when the user logs on.

```
app.userSecurityLevel := 4;
```

Every form, control, and menu item has a **securityLevelVisible** attribute and a **securityLevelEnabled** attribute, which by default are set to zero (**0**). These attributes are usually set in the JADE Painter but they can be set at run time.

For a user to see or use a control or menu item, the value of **app.userSecurityLevel** must be at least as high as the security level attribute of the control or menu item.



Shortcut to Run an Application

You can set up a shortcut on the desktop to run the **Banking** application.

The shortcut is as follows.

```
C:\JadeCourse\bin\jade.exe path=C:\JadeCourse\system
                           ini=C:\JadeCourse\system\jade.ini
                           server=multiuser
                           app=Banking
                           schema=BankingViewSchema
```

Exercise 12.1 - Defining a Banking Application

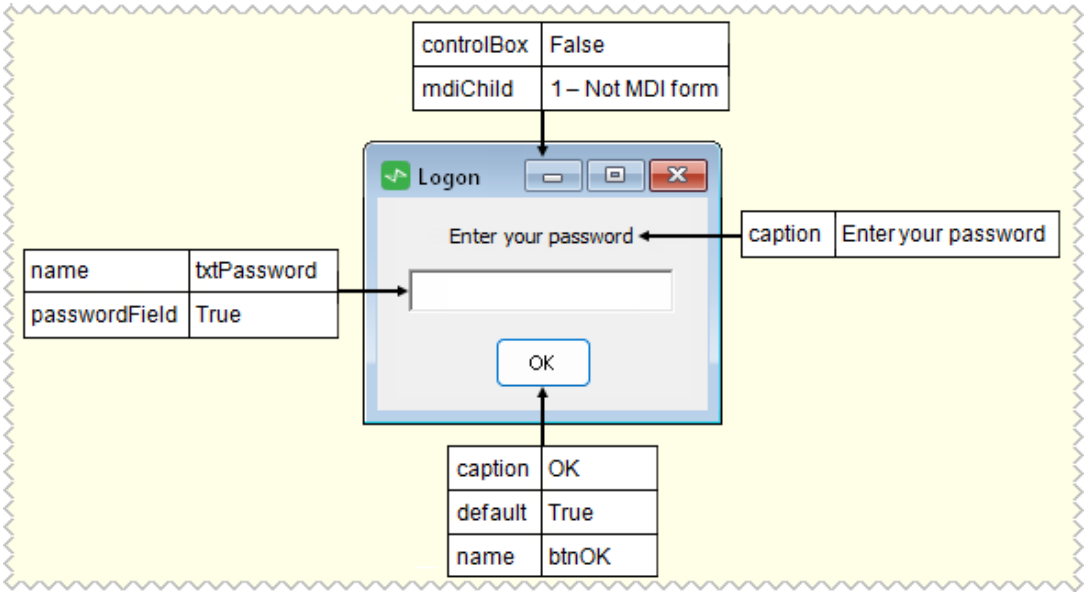
In this exercise, you will change the application that was automatically added when the schema was created, which has the same name as the schema.

1. Open the Application Browser and then select the **BankingViewSchema** application.
2. Select the Application menu **Change** command.
3. Change the name of the application to **Banking**.
4. Select **MainParentForm** as the **Startup Form**.
5. Select **initialize** as the **Initialize Method**, and then click the **OK** button.
6. Run the application, by right-clicking the green arrow in the Jade Platform development environment toolbar.

Exercise 12.2 - Adding a LogonForm Form

In this exercise, you will create a new form called **LogonForm**.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **LogonForm** as the name of the form and select **BVSBaseForm** in the **Sub-Form of** field.
3. Paint the form as shown in the following diagram.



4. Save the form and then return to the Class Browser.

5. In the **LogonForm** form, select the **btnOK** button and then select the **click** event.
6. Code the **click** method as follows.

```
btnOK_click(btn: Button input) updating;

begin
    self.unloadForm();
end;
```

Exercise 12.3 - Reimplementing the getAndValidateUser Method

In this exercise, you will reimplement the **getAndValidateUser** method to test whether the correct password, which is **secret**, is entered on the **LogonForm** form.

1. Select the **GBankingViewSchema** class.
2. Add a **getAndValidateUser** method. A message box warns that there is already a method of that name in a superclass. Click the **Yes** button, to continue.
3. Code the method as follows.

```
getAndValidateUser(usercode: String output; password: String output): Boolean;

vars
    logonForm : LogonForm;

begin
    // Skip authentication if application is not a Windows GUI desktop application
    if app.applicationType <> Application.ApplicationType_GUI then
        return true;
    endif;

    create logonForm transient;

    logonForm.showModal();

    if logonForm.txtPassword.text.toLower() = "secret" then
        return true;
    else
        app.msgBox( "Incorrect passwerd", "Logon Error", MsgBox_OK_Only );
        return false;
    endif;
end;
```

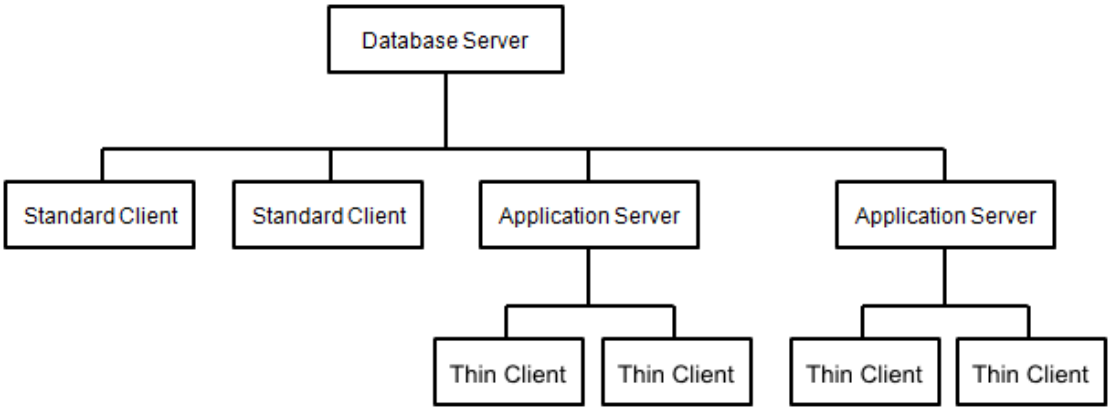
4. Run the **Banking** application and test the logon authentication.

Challenge

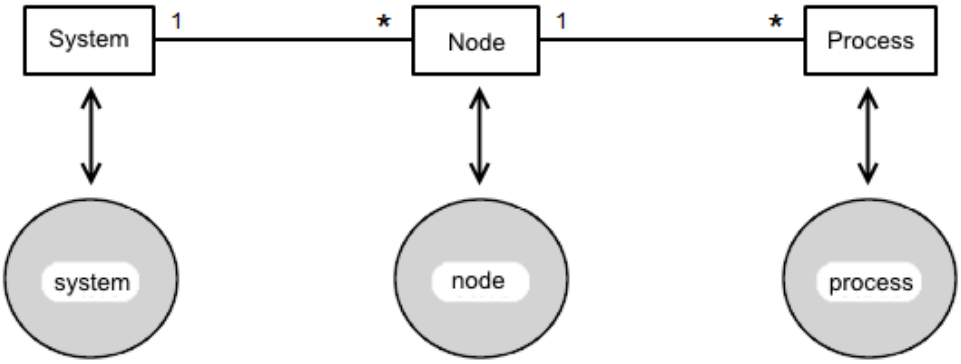
Change the code to give the user three chances to correctly enter the password.

Environmental Objects

The architecture of a Jade Platform multiuser system was explained in an earlier module.



The components of the architecture correspond to instances of the **System**, **Node**, and **Process** classes in **RootSchema**.



The **system** variable represents the collection of all nodes, the **node** variable represents the current node, and the **process** variable represents the current process.

startApplication Methods

The **startApplication**, **startApplicationWithParameter**, **startApplicationWithString**, and **startAppMethod** methods of the **Application** class start a new application or thread from the currently running application.

```
app.startApplication("BankingViewSchema", "Banking");
```

The new application runs in parallel with the application that launched it.

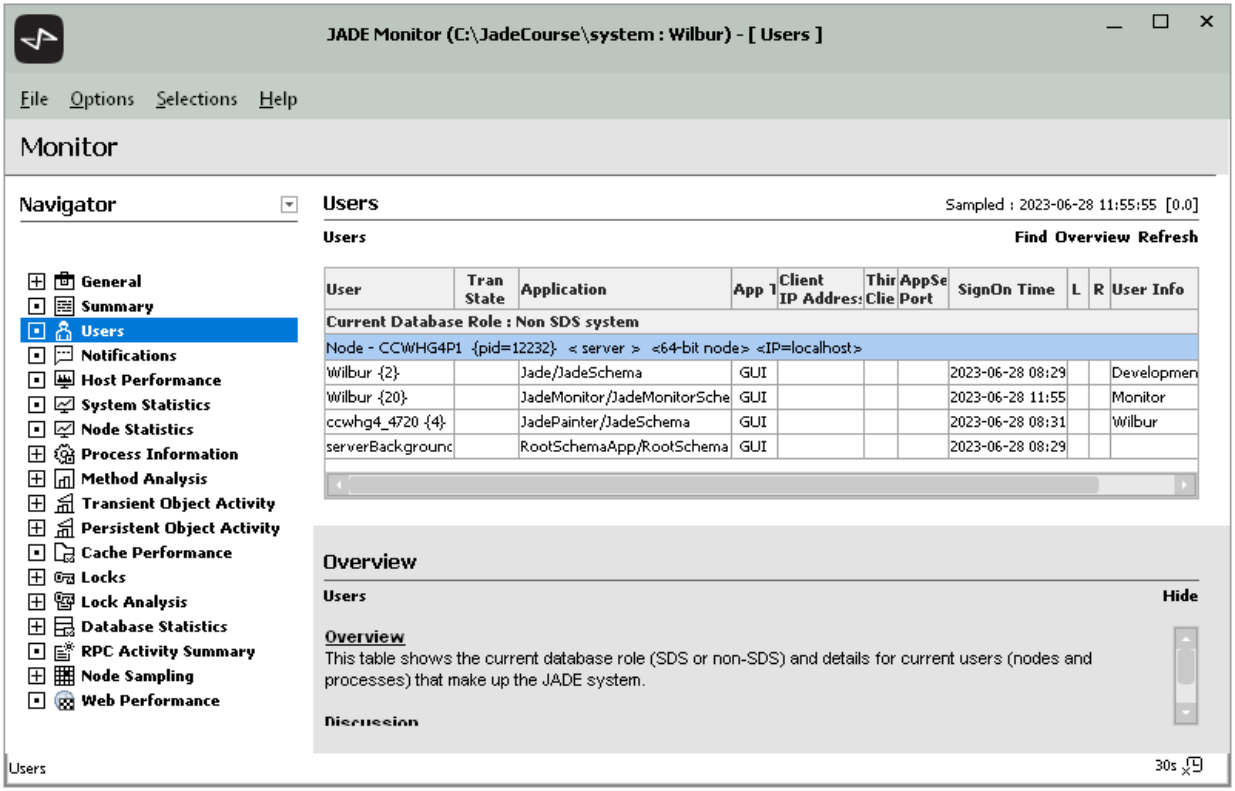
You can use persistent objects or shared transient objects to share information between the applications.

A shared transient object (or a persistent object) can be passed as a parameter with the **startApplicationWithParameter** and **startAppMethod** methods.

Note If the method is used in a **serverExecution** method, the new application runs on the server node. In this case, the parameter passed to the new application must be a persistent object and the new application must not display forms or messages.

Jade Monitor

The Jade Monitor, which can be started by selecting the File menu **Monitor** command, uses functionality from the **System**, **Node**, and **Process** classes.



createExternalProcess Method

The **createExternalProcess** method of the **Node** class starts a new Windows application; for example, you could start **Notepad** as follows.

```
node.createExternalProcess("", "Notepad", null, "", false, false, exit);
```

The signature of the **createExternalProcess** method is:

```
createExternalProcess(directory:  String;
                      command:    String;
                      args:       StringArray;
                      alias:      String;
                      thinClient: Boolean;
                      modal:      Boolean;
                      result:     Integer output): Integer;
```

If the program is not in the current directory or a directory included in the path, the program name must be fully qualified.

As Notepad is a default Windows application, you can leave the path specified in the **directory** parameter blank (that is, "").

The **command** parameter is the name of the process to open, which is Notepad in this topic.

The **args** parameter is for applications that require command line arguments to be able to run.

As the **alias** parameter is ignored, we can just pass in an empty string (that is, "").

The **thinClient** parameter is relevant only when running Jade from a thin (presentation) client. When set to **true**, the external application runs on the presentation client workstation. When it is **false**, it runs on the application server. This parameter in ignored in single-user mode.

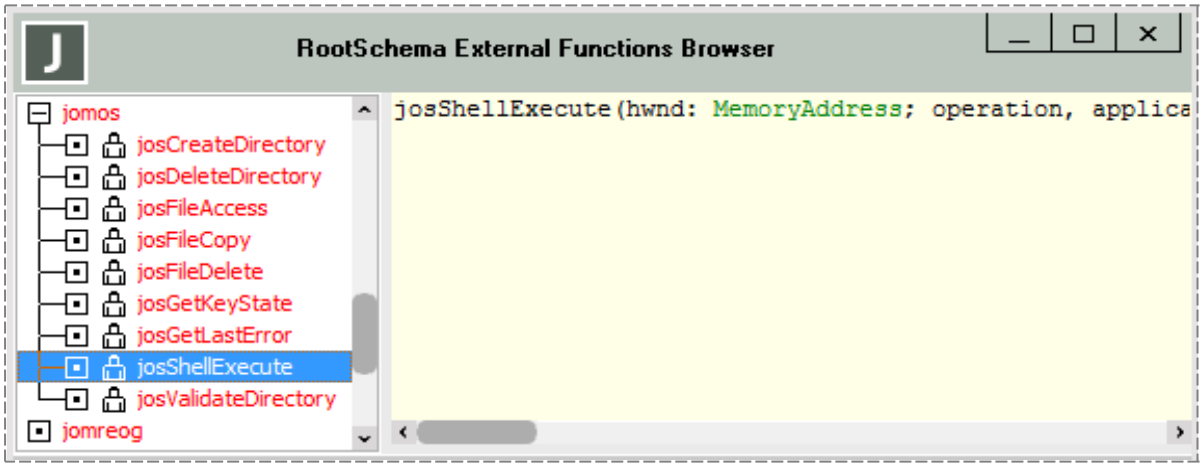
Setting the **modal** parameter to **true** suspends the Jade application until the external application terminates.

Note The **result** parameter is the exit value from the external process. This has meaning only when the **modal** parameter is set to **true**.

Calling External Functions

An external function is a function implemented in a Windows library (DLL). External functions are called directly, by using the **call** instruction. The library that contains the external function could be written by you, by a third party, or provided by the operating system.

You can add libraries and external functions by using the Library Browser and the External Function Browser, respectively.



An external function signature has the following syntax.

```
<function-name>([parameters]) [: <return-type>] is <entry point> in <library>
[presentationClientExecution | applicationServerExecution];
```

The following examples use the **josShellExecute** function in the Jade **jomos** library to open your default Internet browser and e-mail client.

```
// Open default Internet browser
call josShellExecute(null, "open", "http://www.jadeworld.com", "", "", 0);
```

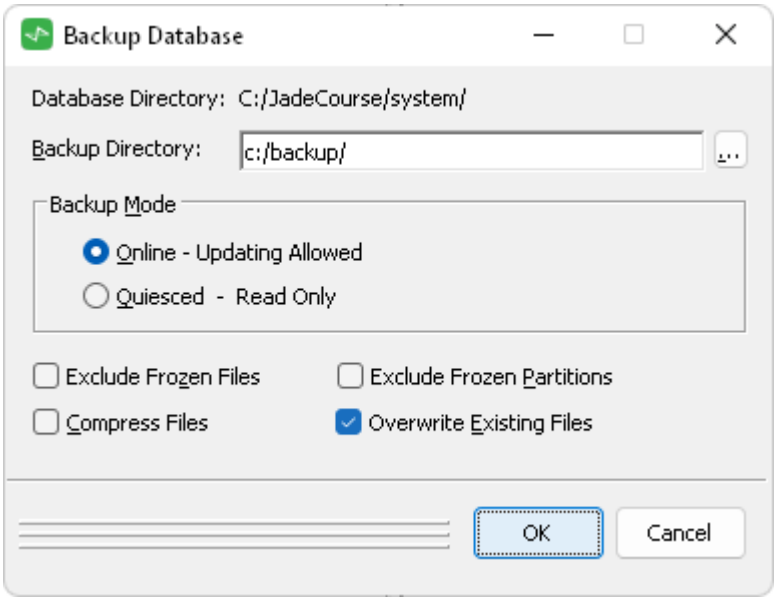
```
// Open default e-mail client
call josShellExecute(null, "open", "mailto:wilbur@jadeworld.com?" &
 "subject=Hello World&body=A traditional greeting.", "", "", 0);
```

Database Backup

The **JadeBackupDatabaseDialog** form is provided in **RootSchema** to enable you to backup database files. Open the form in the standard way, as follows.

```
vars
    dlg: JadeBackupDatabaseDialog;
begin
    create dlg transient;
    dlg.showModal();
end;
```

The form is opened as a modal dialog.



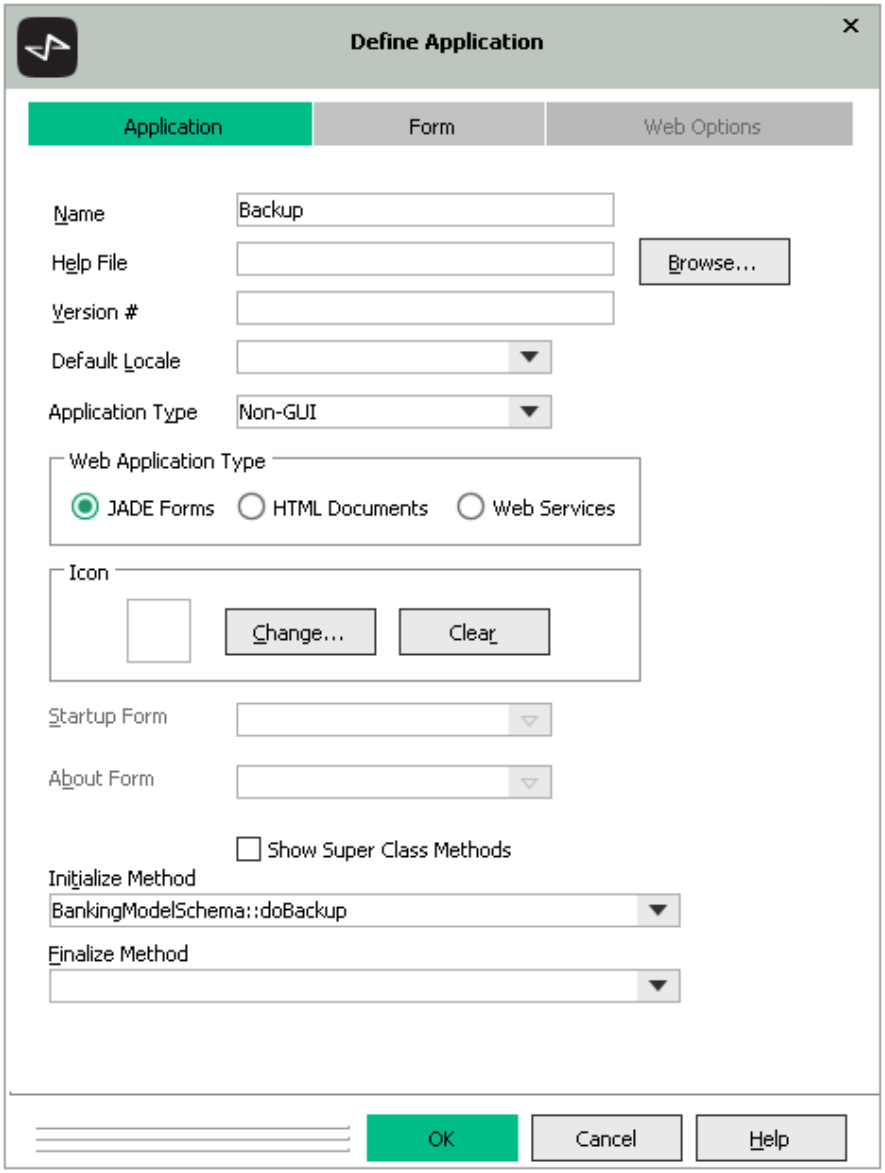
The **JadeDatabaseAdmin** class provides backup and database-related operations. The **backupAllDbFiles** method requires the same kind of information as the **JadeBackupDatabaseDialog** form but it enables the backup to be carried out as a non-GUI operation.

```
vars
    dba: JadeDatabaseAdmin;
begin
    create dba transient;
    dba.backupAllDbFiles("C:\backup", true, false, false, true, false, null);
    terminate;
epilog
    delete dba;
end;
```

Note The **terminate** instruction is used to terminate a non-GUI application. This instruction is not necessary for a GUI application, which is automatically terminated when the last form is closed.

Defining a Non-GUI Application

Non-GUI applications are used to perform tasks that do not require user input, so you do not specify a **Startup Form** but you *do* specify an **Initialize Method** value.



Non-GUI applications can be started from:

- The Jade Platform development environment
- An application using the **startApplication** method

- A shortcut using the **jadclient** program (**jadclient.exe** is the non-GUI equivalent of **jade.exe**); for example:

```
C:\JadeCourse\bin\jadclient.exe path=C:\JadeCourse\system
                                ini=C:\JadeCourse\system\jade.ini
                                server=multiuser
                                app=Backup
                                schema=BankingViewSchema
```

- An entry in the Jade initialization file

```
[JadeServer]
#The following entry runs a backup on the server at 2300 hours
ServerApplication1 = BankingViewSchema, Backup, 2300
```

Exercise 12.4 - Multitasking

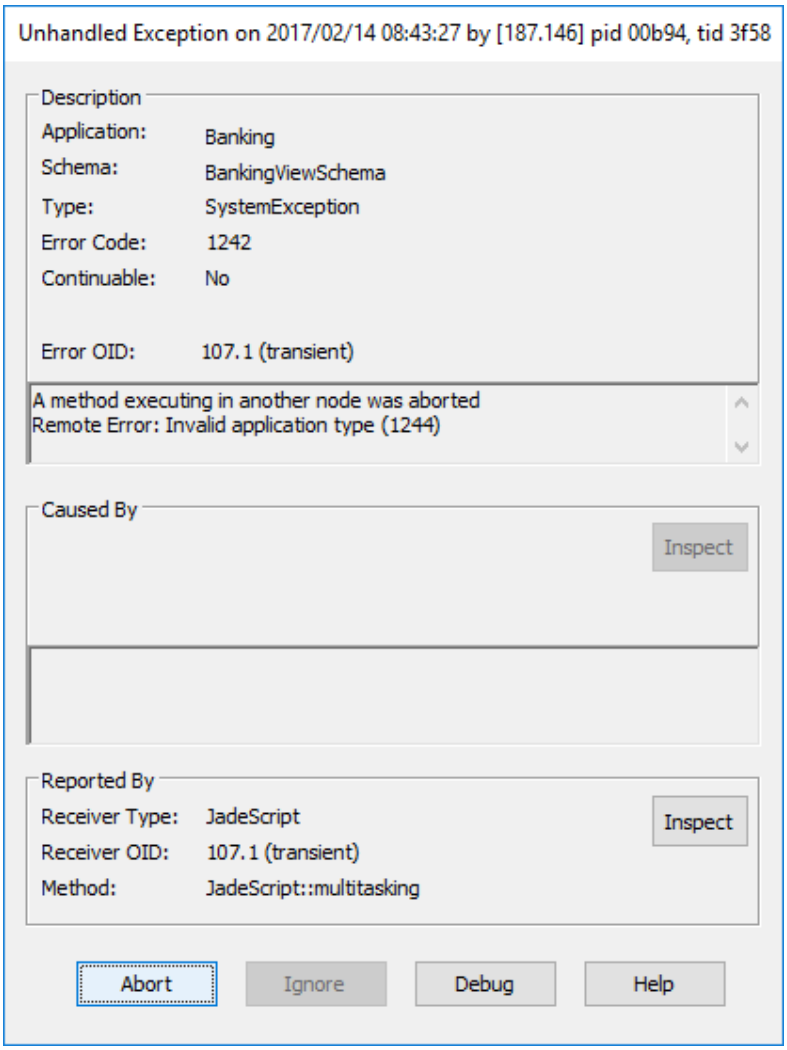
In this exercise, you will write a JadeScript method that uses the **startApplication** method to run a number of applications in parallel.

1. Find the **JadeScript** class.
2. Add a method called **multitasking**, with the following code.

```
multitasking();

begin
    app.startApplication( "BankingViewSchema", "Banking" );
    app.startApplication( "JadeSchema", "Jade" );
    app.startApplication( "JadeMonitorSchema", "JadeMonitor" );
    app.startApplication( "RootSchema", "SchemaInspector" );
end;
```

3. Execute the method.

4. Add the **serverExecution** option to the signature line and then execute the method again. If you are working in multiuser mode, the following dialog will be displayed.



Why does this exception occur?

Exercise 12.5 - Adding a Non-GUI Application

In this exercise, you will write the code for the backup in a method in your **Application** subclass. You will then add a non-GUI application that executes the method.

1. Select the **BankingViewSchema** (your **Application** subclass) in the Class Browser.
2. Add a method called **backup**, by selecting the Methods menu **New Jade Method** command.

3. Code the method as follows.

```
backup();

constants
    BackupFolderName : String = "c:\JadeCourseBackup";

vars
    databaseAdmin : JadeDatabaseAdmin;
    backupFolder : FileFolder;

begin
    // First lets make a backup folder
    create backupFolder transient;
    backupFolder.fileName := BackupFolderName;
    backupFolder.make();

    // Now perform the backup
    create databaseAdmin transient;
    databaseAdmin.backupAllDbFiles( BackupFolderName, true, false, false, true, false, null );

    // We don't need to stay running after the backup completes
    terminate;

epilog
    // Prevent any transient leaks
    delete backupFolder;
    delete databaseAdmin;
end;
```

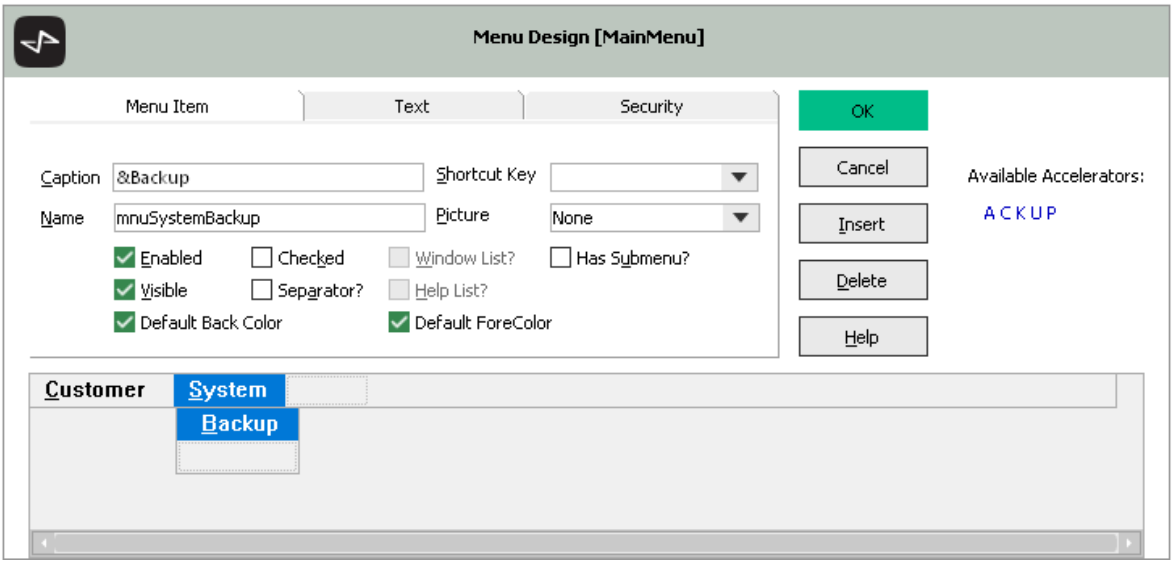
4. Open the Application Browser.
5. Select the Application menu **Add** command.
6. Enter **Backup** as the name of the application.
7. Select **Non-GUI** as the application type.
8. Select **backup** as the **Initialize Method**, and then click the **OK** button.
9. Run the application, by clicking the green arrow in the Jade Platform development environment toolbar and then selecting **Backup** from the combo box.

Exercise 12.6 - Adding Backup to the MainParentForm

In this exercise, you will add a menu item to perform a backup from your banking system.

1. Open the **MainParentForm** form in Painter.
2. Open the menu designer by selecting the File menu **Menu Design** command.
3. Select the menu item to the right of the **Customer** menu, and then enter **&System** in the **Caption** field and **mnuSystem** in the **Name** field.

4. Select the menu item under the **System** menu, and then enter **&Backup** in the **Caption** field and **mnuSystemBackup** in the **Name** field.



5. Click the **OK** button to close the menu designer, and then save the form.

6. In the Class Browser, select the **mnuSystemBackup** menu item and then select the **click** method.

7. Code the method as follows.

```
mnuSystemBackup_click(menuItem: MenuItem input) updating;

begin
    app.startApplication( "BankingViewSchema", "Backup" );
end;
```

8. Run your application and then test the backup function.

Module 13 Exceptions

This module contains the following topics.

Introduction

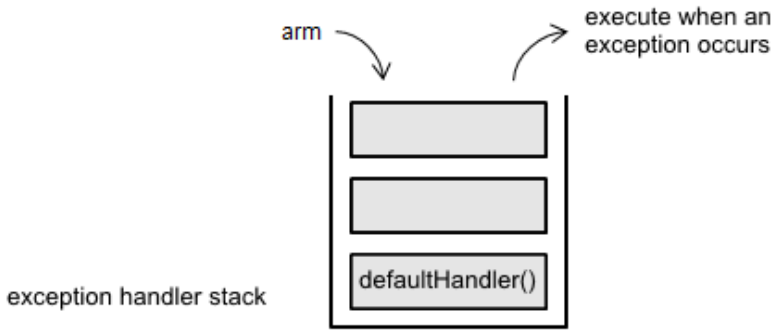
When an application is running, methods execute without error most of the time. Exceptions are error conditions that occur relatively rarely. A pessimistic approach to errors is to check constantly for things that could possibly go wrong, thereby attempting to prevent exceptions from ever occurring. However, there is a performance cost involved in constantly checking. In addition, code involving checks (**if** instructions) is more complicated and difficult to read.

The alternative optimistic approach is to regard exceptions as relatively rare error situations and to deal with them when they happen. Code to handle exceptions is written in separate *exception handler* methods.

The way that an exception is handled depends on the type of application; for example, by displaying a message box in a GUI application and by creating an error log file in a non-GUI application.

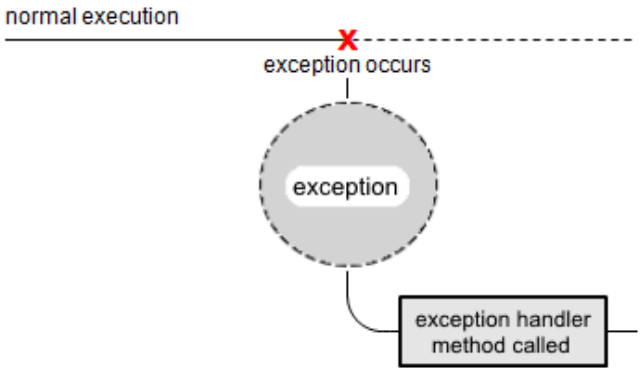
When an error occurs in an application, an instance of **Exception** or one of its subclasses is created by Jade or by your application code. This object contains information about the condition that resulted in the exception being raised; for example, a **FileException** object contains a reference to the file object in use at the time, and a **ConnectionException** contains a reference to the connection object that encountered the error. Control is automatically passed, together with the exception object, to an exception handler method.

Exception handling code is written in separate methods from the methods involved in normal execution flow. At an appropriate place in your code when you judge an exception could occur, you add an instruction to *arm* an exception handler. This instruction adds the exception handler at the top of a stack of armed exception handlers.



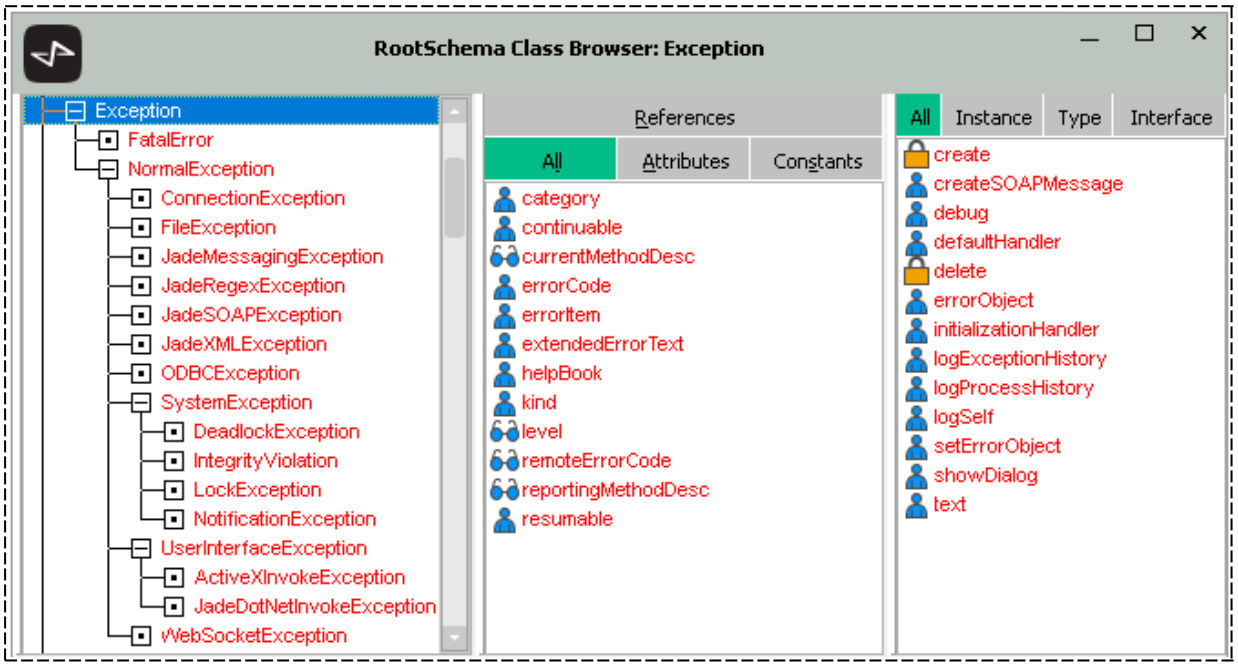
There are two exception handler stacks: a stack for locally armed handlers that are automatically disarmed at the end of the method, and a stack for globally armed handlers that are usually armed when an application starts and that are not disarmed until the application terminates. You can arm up to 128 local exception handlers and up to 128 global exception handlers for each process.

When an exception occurs, normal program flow is interrupted and control passes to the exception handler at the top of the local exception handler stack, and if there are no local handlers, to a global handler.



Exception Classes

There is a hierarchy of **Exception** classes defined in **RootSchema**.



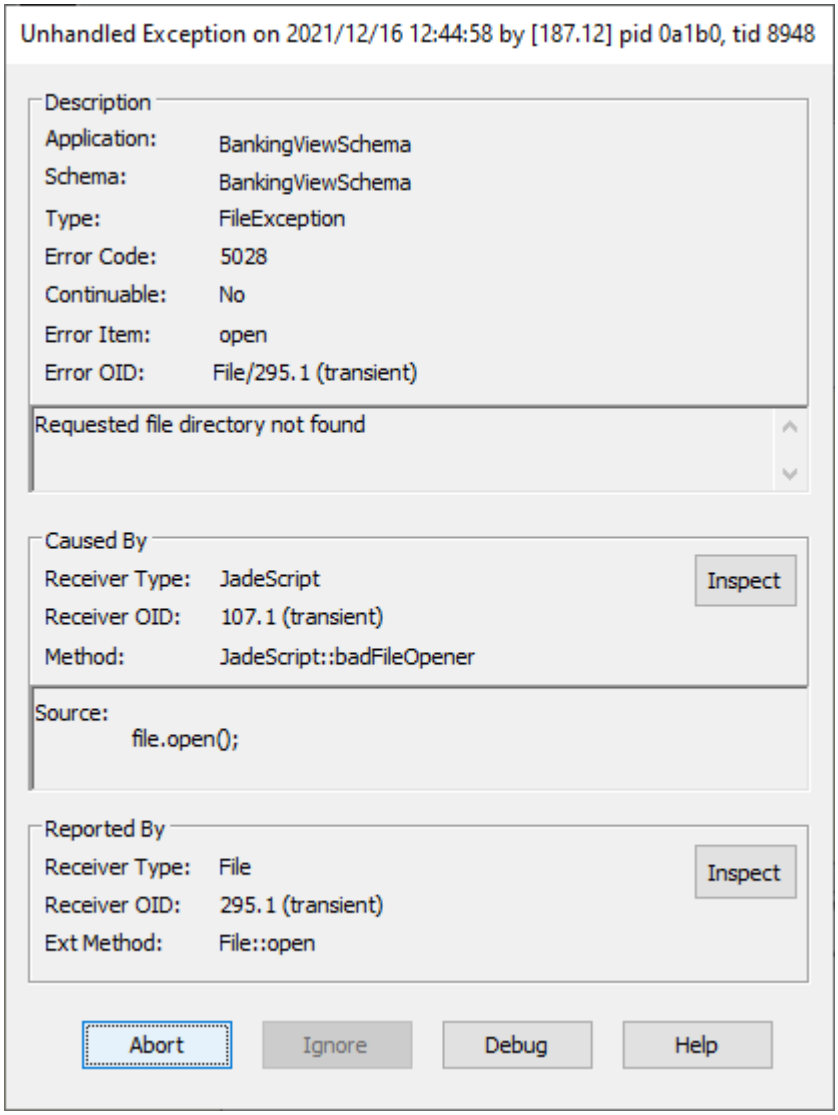
Each class has information and behavior specific to that type of exception. An exception handler is passed the exception object, so that it can use make use of this information and behavior.

The **Exception** class includes an **errorCode** integer attribute and a **text** method that looks up a brief description of the exception in a file called **jadmsgs.eng**. For example, an exception with **errorCode** 1090 has a **text** description *Attempted access via null object reference*.

There are a number of methods for logging exception details.

Default Exception Handler

The Jade Platform provides a default exception handler, which displays the Unhandled Exception dialog and logs exception information. The dialog is displayed if you do not code and arm your own exception handler.



The dialog provides useful information for developers in debugging an exception. However, it is not appropriate for application users.

The error object reported by the default exception handler includes the type name before the object identifier (OID) if the class number is valid; for example:

```
...
Error item: setFontProperties
Error object: TextBox/509.21 (transient)
Caused By:
   Receiver: MainForm/1004290.1 (transient)
   Method: MainForm::setupClipText(1037)  --  tb.setFontProperties
   (tblClipBoard.fontName, tblClipBoard.fontSize, tblClipBoard.fontBold);
Reported By:
   Receiver: TextBox/509.21 (transient)
   Method: Control::setFontProperties  -- 'JadeControlSetFont' in 'jadpmap'
...
```

If there is no class in the current system that has the specified class number, only the OID is displayed.

Coding an Exception Handler

An exception handler method contains the exception object as its first parameter. It can contain additional parameters to provide more information about the context of the exception.

The method returns an integer to specify what is to happen next. There are four possible return values, which are described in the following section. What you do next depends on how successful you are in resolving the exception.

The following examples show exception handler method signatures.

```
exHandlerA(ex: Exception): Integer;
```

```
exHandlerB(ex: Exception; cust: Customer): Integer;
```

```
exHandlerC(ex: FileException): Integer;
```

The following method handles unanticipated exceptions in an application and would effectively replace the default exception handler.

```
genericExceptionHandler(exObj: Exception): Integer;

begin
    // Abort database transaction to release locks
    abortTransaction;
    exObj.logSelf("errors.log");
    app.msgBox("An unexpected error has arisen", "Application Error", MsgBox_OK_
Only);
    // Cut back the execution stack
    return Ex_Abort_Action;
end;
```

The following method handles a *string too long* exception, which could arise when too much text is entered in a text box on the **CustomerAdd** form or too much text is read from a file.

```
stringTooLongHandler(exObj: Exception): Integer;

begin
    if exObj.errorCode = 1035 then
        // Abort database transaction to release locks
        abortTransaction;
        exObj.logSelf("errors.log");
        app.msgBox("Reduce the amount of text", "Application Error", MsgBox_OK_
Only);
        // Cut back the execution stack
        return Ex_Abort_Action;
    else
        // Pass exception to next armed handler
        return Ex_Pass_Back;
    endif;
end;
```

Arming an Exception Handler

An exception handler can be armed:

- Locally, when it remains armed until the method in which it was armed has returned (unless explicitly disarmed).

  Local exception handlers are typically armed at the start of a method where the exception could occur.

- Globally, when it remains armed until the process terminates (unless explicitly disarmed).

  Global exception handlers are typically armed in the **initialize** method for the application.

There are two exception handler stacks: one for up to 128 locally armed exception handlers, and one for the default Jade exception handler and up to 127 globally armed exception handlers.

Handlers from the local exception handler stack are executed before handlers from the global exception handler stack, regardless of the order in which they are armed.

The syntax for locally arming an exception handler is as follows.

```
on Exception-class do exception-handler-method(exception[, parameters]);
```

The first parameter of an exception handler is the system variable **exception**, which is a reference to the exception object.

The following examples show the arming of local exception handlers.

- **exHandlerA** is called for any type of exception and is coded in the same class as the method causing the exception.

  ```
  on Exception do self.exHandlerA(exception);
  ```

- **exHandlerB** is passed additional information through the **cust** parameter, which is evaluated when the handler is invoked.

  ```
  on Exception do self.exHandlerB(exception, cust);
  ```

- **exHandlerC** is a method in an **Application** class that is invoked only for file exceptions.

```
on FileException do app.exHandlerC(exception);
```

The syntax for globally arming an exception handler is the same as for local arming, with the keyword **global** appended.

```
on Exception-class do exception-handler-method(exception[, parameters]) global;
```

The following examples show the arming of global exception handlers.

- **genericExceptionHandler** is called for any type of exception and is coded in one of the **Application** classes. This should be the first handler to be armed.

```
on Exception do self.genericExceptionHandler(exception) global;
```

- **lockExceptionHandler** is called only for lock exceptions. This should be the armed after **genericExceptionHandler**.

```
on LockException do self.lockExceptionHandler(exception) global;
```

Returning from an Exception

The integer that is returned from an exception handler, for which you can use a global constant, determines what happens next.

Global Constant	Description
Ex_Pass_Back	Control is given to any previously-armed local exception handler for this type of exception, or if a local exception handler is not found, a global exception handler. If no exception handler is found, the Jade default exception handler is invoked.
Ex_Abort_Action	Currently-executing methods are removed from the execution stack. The application reverts to an idle state in which it is waiting for user input or some other event. Returning **Ex_Abort_Action** does not abort a database transaction, so remember to include an **abortTransaction** instruction.
Ex_Continue	Execution resumes from the next expression following the expression that caused the exception. In order to use **Ex_Continue** as the return value, the exception must be **continuable**. Continuable exceptions assume that the cause of the problem has been fixed and the operation retried. This approach can be used for lock exceptions and user exceptions.
Ex_Resume_Next	Control is given to the method that armed the exception handler. Execution resumes at the next statement after the method call expression in which the exception occurred. **Ex_Resume_Next** is generally useful only for local exception handlers when the method that armed the exception handler is still executing.

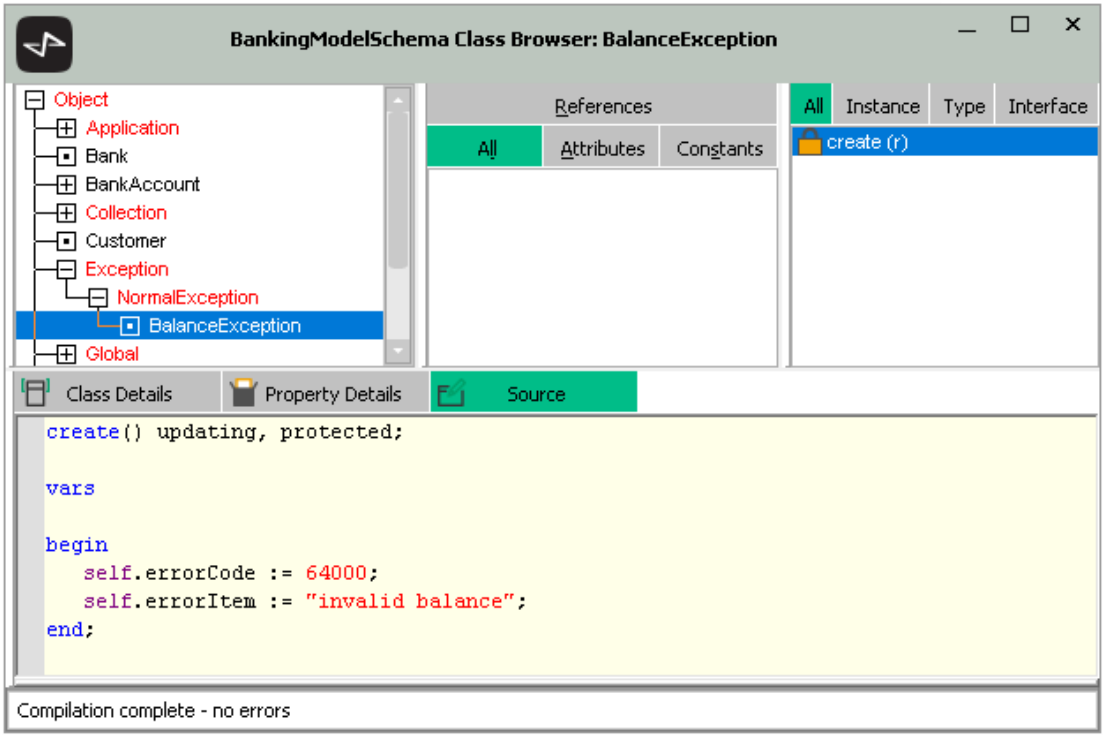
User Exceptions

As a Jade application developer, you can create an exception object and set its properties in your code. When the **raise** instruction is executed, control passes to an armed exception handler.

The following JadeScript method creates and raises an exception.

```
userException();

vars
    ex: Exception;
begin
    create ex;
    ex.errorCode := 12345;
    raise ex;
end;
```

You can add an exception class, as shown in the following image.



User exceptions are often used to enforce business rules; for example, you could protect against an invalid balance being set for a bank account by raising exceptions in the **create** and **setPropsOnUpdate** methods of a bank account class.

```
create(bal, od: Decimal; cust: Customer) updating;

vars
    ex: BalanceException;
begin
    if bal < 0 then
        create ex;
        raise ex;
    endif;
    self.balance := bal;
    self.overdraftLimit := od;
    self.myCustomer := cust;
    self.myBank := app.myBank;
end;
```

Mapping Method

A mapping method has the same name as a property and is automatically invoked when the property is read or modified in a method. It is used to reimplement the default *get* and *set* behavior for a property.

A mapping method always has the following signature.

```
<property-name>(set: Boolean; _value: <property-type> io) mapping;
```

The **set** parameter is **true** if the property is being assigned, and **false** if it is being read.

If **set** is:

- **true**, **_value** is the proposed new value of the property that is assigned.
- **false**, **_value** is the value of the property returned to the calling method.

Exercise 13.1 - Causing an Exception

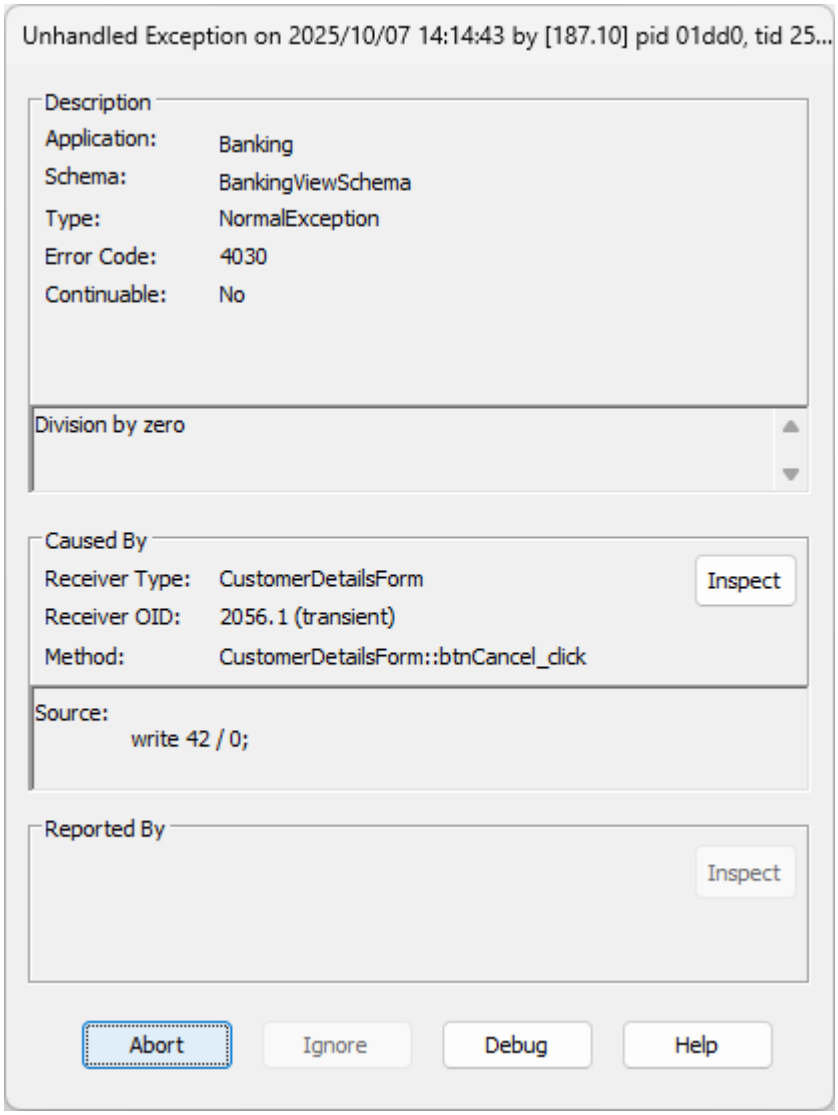
In this exercise, you will add code that deliberately causes an exception.

1. Open a Class Browser for the **BankingViewSchema**.
2. Select the **CustomerDetailsForm** form.
3. Change the **click** event method for **btnCancel**, as follows.

```
btnCancel_click(btn: Button input) updating;

begin
    write 42 / 0;

    self.unloadForm();
end;
```

4. Run the **Banking** application and open the **CustomerDetailsForm** form.

a. Click the **Cancel** button, to display the unhandled exception dialog shown in the following image.



Exercise 13.2 - Adding a Global Exception Handler

In this exercise, you will add a generic exception handler in your **Application** class to be invoked if an unforeseen application error occurs. You will arm the handler globally in the **initialize** method. Finally, you will run the application and test the handler.

1. Open a Class Browser for the **BankingModelSchema**.
2. Add a method called **genericExceptionHandler** in the **BankingModelSchema** class (your **Application** subclass).

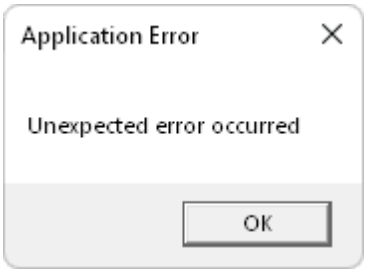
3. Code the method as follows.

```
genericExceptionHandler( pException : Exception ) : Integer;

begin
    abortTransaction;

    pException.logSelf( "errors.log" );

    app.msgBox( "Unexpected error occurred", "Application Error", MsgBox_OK_Only );

    return Ex_Abort_Action;
end;
```

   a. Arm the exception handler globally at the start of the **initialize** method, as follows.

```
initialize() updating;

begin
    // Arm our global exception handler
    on Exception do self.genericExceptionHandler( exception ) global;

    // Try to get the Bank, if it exists.
    self.myBank := Bank.firstInstance();

    // If it doesn't exist yet, then create the Bank singleton
    if self.myBank = null then
        beginTransaction;
        create self.myBank persistent;
        commitTransaction;
    endif;
end;
```

4. Run the **Banking** application in the **BankingViewSchema**.
5. Open the **CustomerDetailsForm** form and then click the **Cancel** button to display the message box.



Exercise 13.3 - Deliberately Causing Another Exception

In this exercise, you will add code that deliberately causes an exception if too much text is entered into a text box.

1. Open a Class Browser for the **BankingViewSchema**.
2. Select the **CustomerDetailsForm** form.

3. Add a new method called **formLoad** and code it as follows.

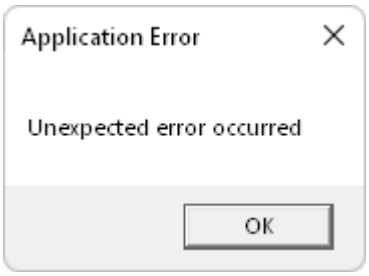
```
formLoad() updating, protected;

begin
    inheritMethod();

    self.txtLastName.maxLength := 0;
end;
```

Note When you painted the form, you set the **maxLength** attribute of the **txtLastName** text box to 15 characters. This restriction is removed by setting it to zero (**0**).

4. Run the **Banking** application and then open the **CustomerDetailsForm** form.

5. Enter information for a new customer who has a last name with more than 15 characters.

   When you click the **OK** button, the *unexpected error* message should be displayed, as shown in the following image.



Exercise 13.4 - Adding a Local Exception Handler

In this exercise, you will add a local exception handler in your **CustomerDetailsForm** form to be invoked if too much text is entered for a customer's last name. You will arm the handler locally at the start of the **btnOK_click** method. Finally, you will run the application and test the handler.

1. Select the **CustomerDetailsForm** class.

2. Add a method called **stringTooLongHandler** and code the method as follows.

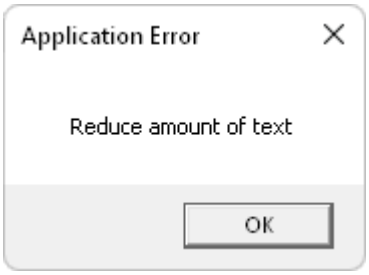
```
stringTooLongHandler( pException : Exception ) : Integer;

begin
    if pException.errorCode = 1035 then
        // Abort database transaction to release any locks
        abortTransaction;

        pException.logSelf( "errors.log" );

        app.msgBox( "Reduce amount of text", "Application Error", MsgBox_OK_Only );

        // Cut back the execution stack
        return Ex_Abort_Action;
    else
        // Pass back to next armed handler
        return Ex_Pass_Back;
    endif;
end;
```

3. Arm the exception handler locally at the start of the **btnOK_click** method, as follows.

```
btnOK_click(btn: Button input) updating;

vars
    isAddingCustomer : Boolean;

begin
    on Exception do self.stringTooLongHandler( exception );

    isAddingCustomer := self.getCurrentObject() = null;

    if not self.doSave() then
        return;
    endif;

    if isAddingCustomer then
        self.statusLine.caption := "Customer added successfully";
        self.displayObject( null );     // Clear the fields ready for the next customer to be added
    else
        self.statusLine.caption := "Customer updated successfully";
    endif;
end;
```

Run the **Banking** application and then open the **CustomerDetailsForm** form.

4. Enter information for a new customer who has a last name with more than 15 characters. When you click the **OK** button, a message box related to the error should be displayed.



Exercise 13.5 - Adding Validation Rules in the Transaction Agent Framework

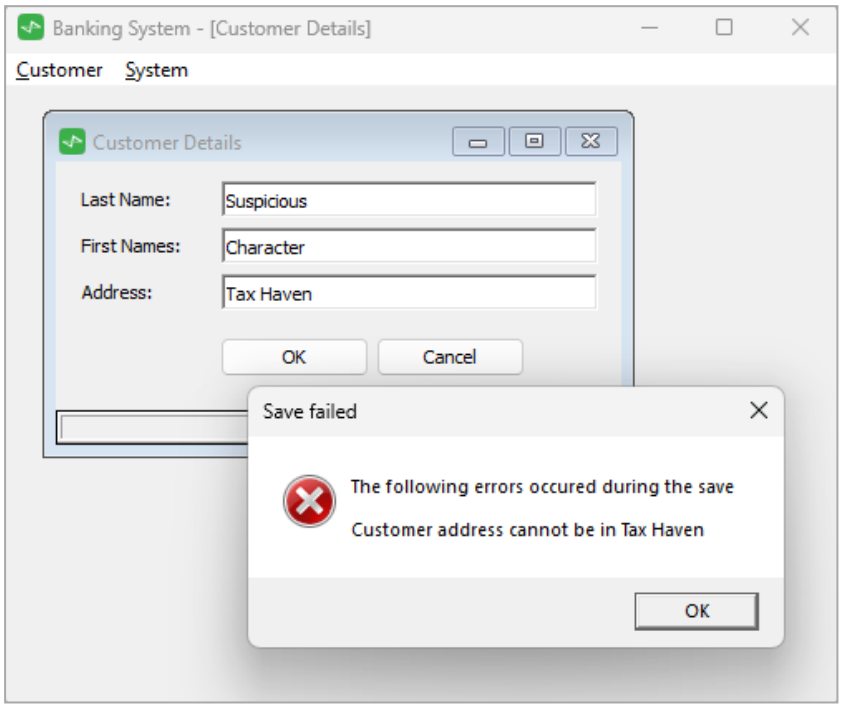
In this exercise, you will add a new validation rule to enforce the business rule that the address of a customer should not be **Tax Haven**, by returning a validation error when an attempt is made to assign that value. You will implement this rule by adding logic to the **CustomerTA** class **doValidate** method, and then testing it by running the **Banking** application.

1. Open a Class Browser for the **BankingModelSchema**.
2. Select the **CustomerTA** class.
3. Select the existing **doValidate** method and adjust the code as follows.

```
doValidate( pOperation : Integer ) : Boolean updating;

begin
    inheritMethod( pOperation );

    // We don't want to do this validation for BMS_TransactionType_Delete or BMS_TransactionType_Modify
    if pOperation.isOneOfTheseValues( BMS_TransactionType_Create, BMS_TransactionType_Update ) then
        if self.lastName = null then
            self.addError( "You need to provide a last name", BMS_FocusField_LastName );
        endif;

        if self.firstNames = null then
            self.addError( "You need to provide first names", BMS_FocusField_FirstNames );
        endif;

        if self.address = null then
            self.addError( "You need to provide an address", BMS_FocusField_Address );
        endif;

        if self.address = "Tax Haven" then
            self.addError( "Customer address cannot be in Tax Haven", BMS_FocusField_Address );
        endif;
    endif;

    return self.hasNoErrors();
end;
```

Run the **Banking** application and then open the **CustomerDetailsForm** using the **Add** command from the Customer menu.

4. Enter information for a new customer with an address of **Tax Haven**. When you click the **OK** button, a message should be displayed advising of the validation error.



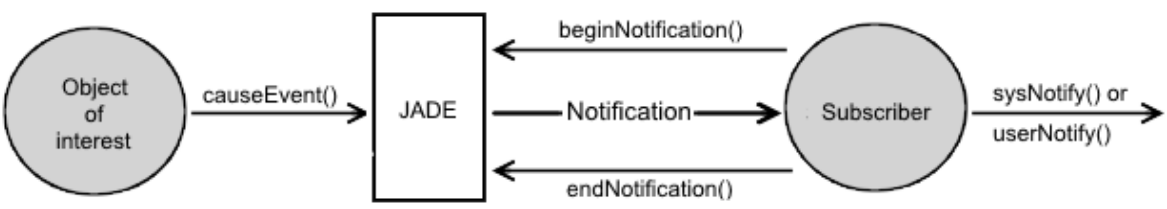
Module 14 Notifications and Timers

This module contains the following topics.

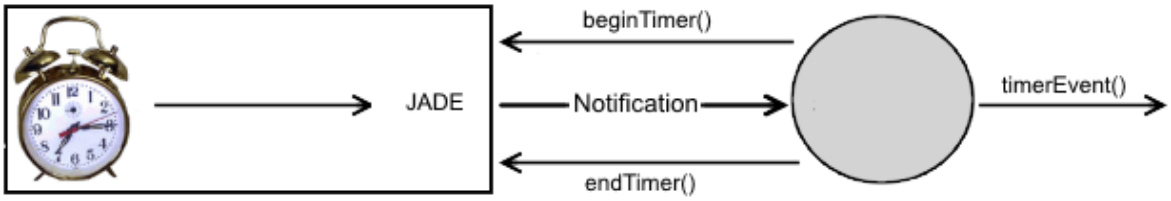
Introduction

A *notification* is a message sent by the Jade Object Manager to an object (for example, a form), to inform it that an event has happened to an object of interest.

The process begins with the subscriber to the notifications executing the **beginNotification** method specifying the object in which the subscriber is interested. When the event happens, the object of interest uses the **causeEvent** method to inform the Jade Object Manager, which then notifies the event to those who subscribed to it. Subscribers, on being notified of the event, execute the **sysNotification** or **userNotification** event method, if one has been coded.



A *timer* is a mechanism whereby an object triggers an event for itself at regular intervals. The process begins with the object executing the **beginTimer** method, to specify the interval between events. When the event occurs, the object executes the **timerEvent** method. The timer can be stopped by the object executing the **endTimer** method.



All of the methods involved in notifications and timers are defined in the **Object** class.

Notifications and Events

This section covers notification messages sent by the Jade Object Manager to an object, informing it that an event has happened to an object of interest.

For details, see the following subsections. See also "Timer Events", later in this module.

System Events

System events are the standard operations of creating, updating, and deleting a persistent object.

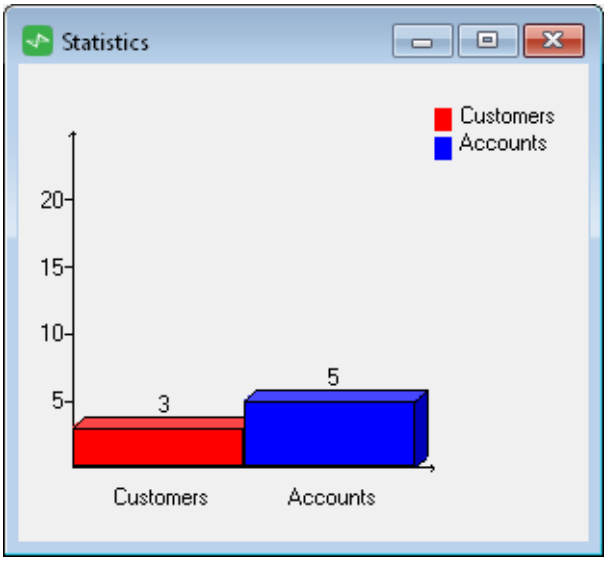
The following global constants are associated with system events. When a system event occurs, the Jade Object Manager sends notifications to any object that has subscribed to the event.

- Object_Create_Event (4)
- Object_Update_Event (3)
- Object_Delete_Event (6)
- Any_System_Event (0)

Notes System notifications are invoked for persistent objects only.

As the Jade Object Manager does not have to be informed about creating, updating, or deleting a persistent object, when the event occurs, the object involved does *not* have to execute the **causeEvent** method.

System notifications are often used to keep the display of information on a form current. The following image shows a form with a graphical display of the number of **Customer**, **ChequeAccount**, and **SavingsAccount** objects that are updated automatically when objects are added or deleted.



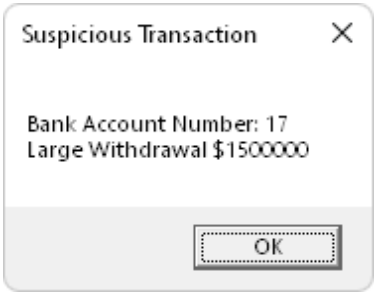
User Events

User events enable you to define your own events for which the Jade Object Manager will send notifications, in the same way as for system events. The object involved in the user event causes the event to be published by executing the **causeEvent** method. The Jade Object Manager then sends notifications to any object that has subscribed to the event.

Each user event is associated with an integer value that is greater than 15. (Integers in the range 0 through 15 are reserved for system events.)

Tip Define an integer global constant for a user event, to make your code more readable.

User notifications can be used to generate an alert when an unusual event occurs. The following image shows a message box that displays when a million dollars or more is withdrawn from a bank account.



Subscribing to Notifications

The **beginNotification** method requests notification of events that occur to a specified object.

```
beginNotification(theObject: Object;
                  eventType: Integer;
                  responseType: Integer;
                  eventTag: Integer);
```

The **beginClassNotification** method requests notification of events that occurs to any instance of a specified class or its subclasses.

```
beginClassNotification(theClass: Class;
                       transients: Boolean;
                       eventType: Integer;
                       responseType: Integer;
                       eventTag: Integer);
```

The parameters for these methods are described in the following table.

Parameter	Description
theObject	Object of interest.
theClass	Class (including subclasses) of objects of interest.
transients	Whether the objects of interest are transient or persistent.
eventType	Number identifying the type of event.
responseType	Whether notifications are automatically canceled after the first event. Possible values are: ■ **Response_Continuous** – continue to send notifications ■ **Response_Cancel** – cancel notifications after the first event
eventTag	Value that is returned as part of the notification – can be used to tag subscriptions.

Unsubscribing from Notifications

The **endNotification** method cancels notification of events that occur to a specified object.

```
endNotification(theObject: Object;
                eventType: Integer);
```

The **endClassNotification** method cancels notification of events that occur to any instance of a specified class, or its subclasses.

```
endClassNotification(theClass: Class;
                     transients: Boolean;
                     eventType: Integer);
```

Note You should cancel notifications for a subscriber (for example, a form) before it is deleted. An exception is raised for a notification that cannot be delivered.

Publishing a User Event

The **causeEvent** method, defined on the **Object** class, informs the Jade Object Manager that a user event has occurred so that user notifications can be sent.

```
causeEvent(eventType: Integer;        // Number identifying the type of event
           immediate: Boolean;        // Whether notifications are sent immediately
or
                                      // at the next commitTransaction instruction
           userInfo: Any);            // Value passed to userNotification method
```

An example of a user event is a bank account withdrawal that exceeds a threshold value (for example, a million dollars). The **causeEvent** could be coded in the **withdraw** method (or in the mapping method for the **balance** property), as follows.

```
withdraw(amount: Decimal) updating;

begin
   if self.canWithdraw(amount) = true then
        self.balance := self.balance - amount;
        if amount > 1000000 then
            self.causeEvent(LargeWithdrawal, false, amount);
        endif;
    endif;
end;
```

Responding to Notifications

The **sysNotification** method is invoked when a system event (creating, updating, or deleting an object) occurs for a persistent object.

```
sysNotification(eventType: Integer;       // Number identifying the type of event
                theObject: Object;        // Object that caused the event
                eventTag: Integer);       // Value passed from beginNotification
method
```

Note If the event is the deletion of a persistent object, the **theObject** parameter references an object that no longer exists. Attempting to access this object raises an exception.

The **userNotification** method is invoked when a user event occurs.

```
userNotification(eventType: Integer;      // Number identifying the type of event
                 theObject: Object;       // Object that caused the event
                 eventTag: Integer;       // Value passed from beginNotification
method
                 userInfo: Any);          // Value passed from the causeEvent method
```

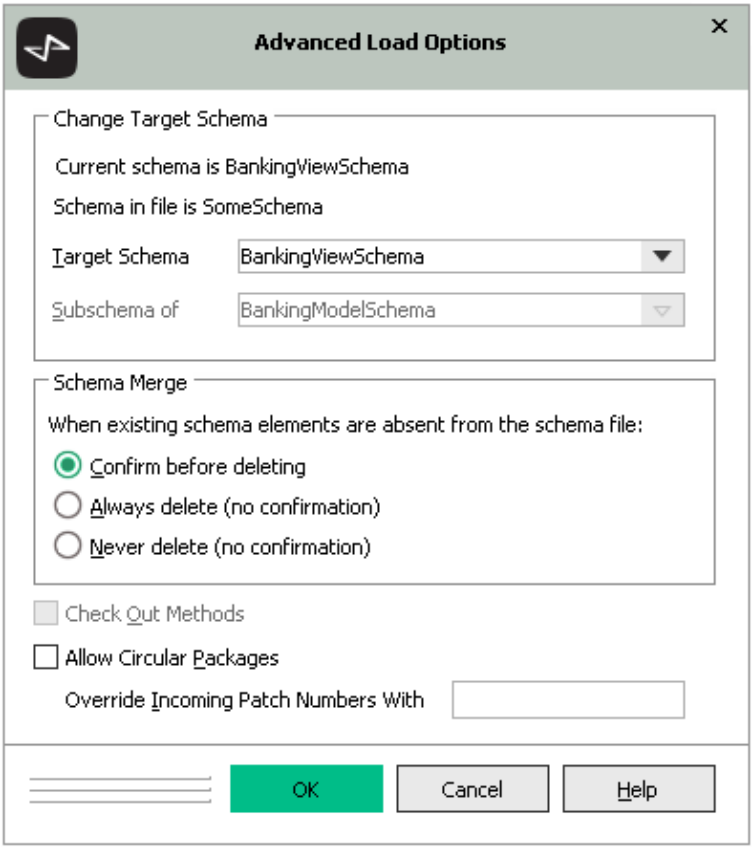
For controls and forms, you can code the **sysNotify** and **userNotify** event methods instead of the corresponding **sysNotification** and **userNotification** methods.

Exercise 14.1 – Loading a Class

In this exercise, you will load a class for drawing bar graphs (which was created in another Jade schema) into the **BankingViewSchema**. You will use this control in the next exercise.

1. Select the Schema Browser.
2. Select the Schema menu **Load** command.
3. In the **Schema File Name** text box, browse for the **C:\JadeCourse\Files\ThreeDeeGraph.cls** file.
4. In the **Forms File Name** text box, browse for the **C:\JadeCourse\Files\ThreeDeeGraph.ddx** file.

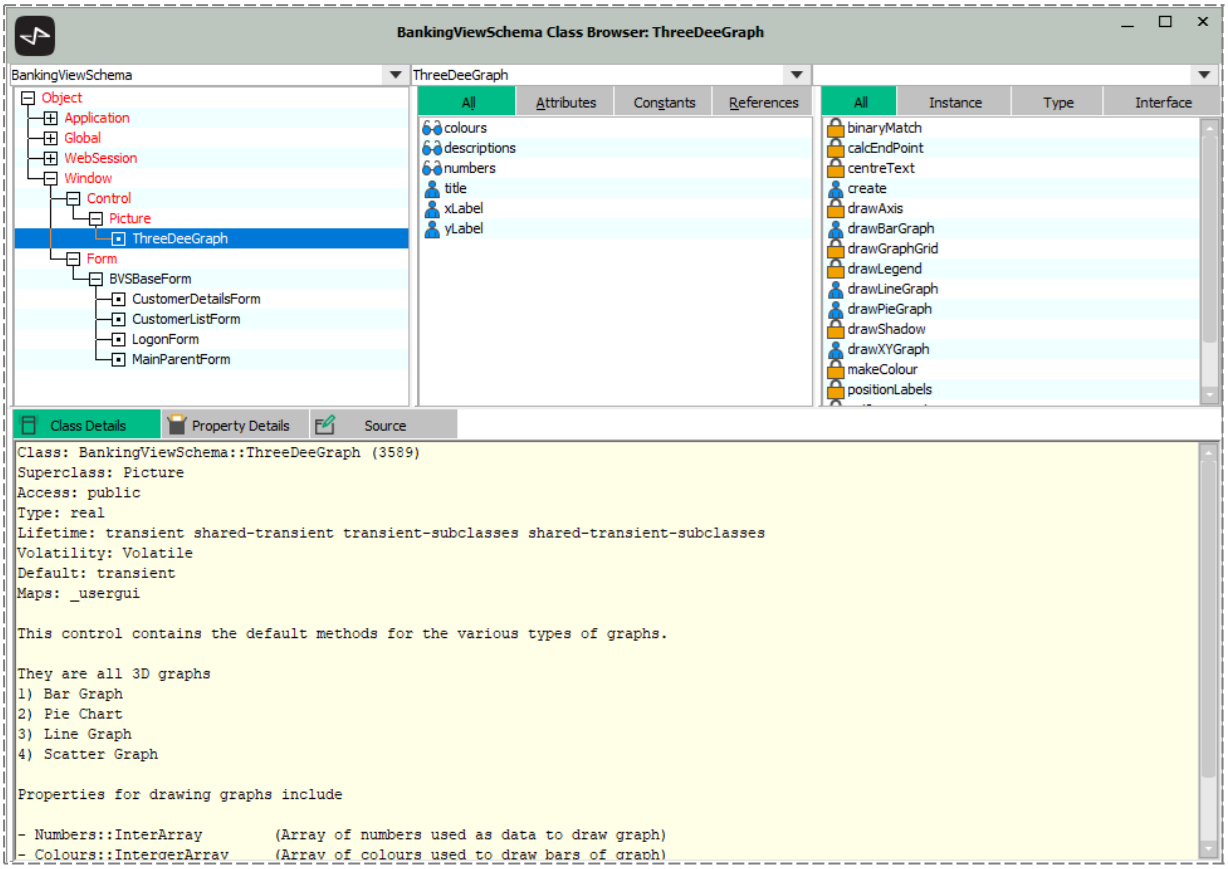
5. Click the **Advanced** button, to open the Advanced Load Options dialog shown in the following image.



6. Select **BankingViewSchema** as the **Target Schema** and then click the **OK** button of the Advanced Load Options dialog.

7. Click the **OK** button on the Load Options dialog, to load the class.

In the **BankingViewSchema**, a subclass of **Picture** has been loaded.



Exercise 14.2 – Using System Notifications

In this exercise, you will add a **Statistics** form and paint a **ThreeDeeGraph** control on it. You will add a method called **draw** to the **Statistics** form, which sets the values of the **colours**, **descriptions**, and **numbers** arrays.

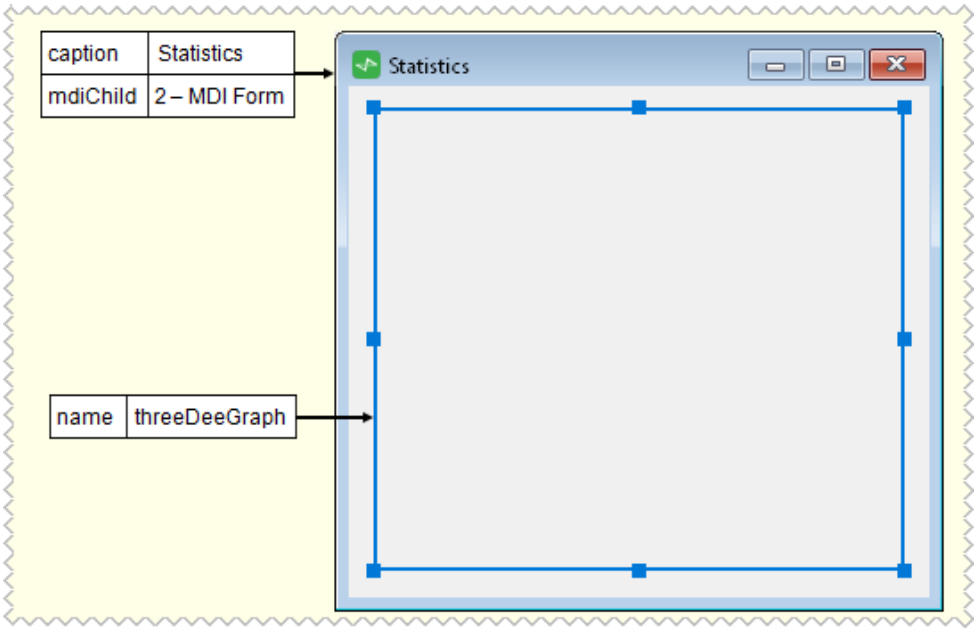
The arrays control the appearance of the bars when the control's **drawBarGraph** method is executed. The **numbers [1]** value is the height of the first bar, which is the number of customers. The value is obtained from the **size** of the **app.myBankAllCustomers** collection. The **colours[1]** value is an integer that determines the color of the bar. The **descriptions[1]** value is the string that is displayed below the bar.

The bar graph is drawn by calling the **draw** method from the **load** method.

Finally, you will add notifications to automatically redraw the bar graph when a new **Customer** object is added.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **StatisticsForm** as the name of the form and select **BVSBaseForm** in the **Sub-Form of** field.

3. Paint a **ThreeDeeGraph** control on the form and then save the form.



4. Add a **draw** method to the **StatisticsForm** form and code it as follows.

```
draw();

begin
    self.threeDeeGraph.descriptions[ 1 ] := "Customers";
    self.threeDeeGraph.descriptions[ 2 ] := "Accounts";

    self.threeDeeGraph.colours[ 1 ] := Red;
    self.threeDeeGraph.colours[ 2 ] := Blue;

    self.threeDeeGraph.numbers[ 1 ] := app.myBank.allCustomersByLastName.size();
    self.threeDeeGraph.numbers[ 2 ] := app.myBank.allBankAccountsByNumber.size();

    self.threeDeeGraph.drawBarGraph();
end;
```

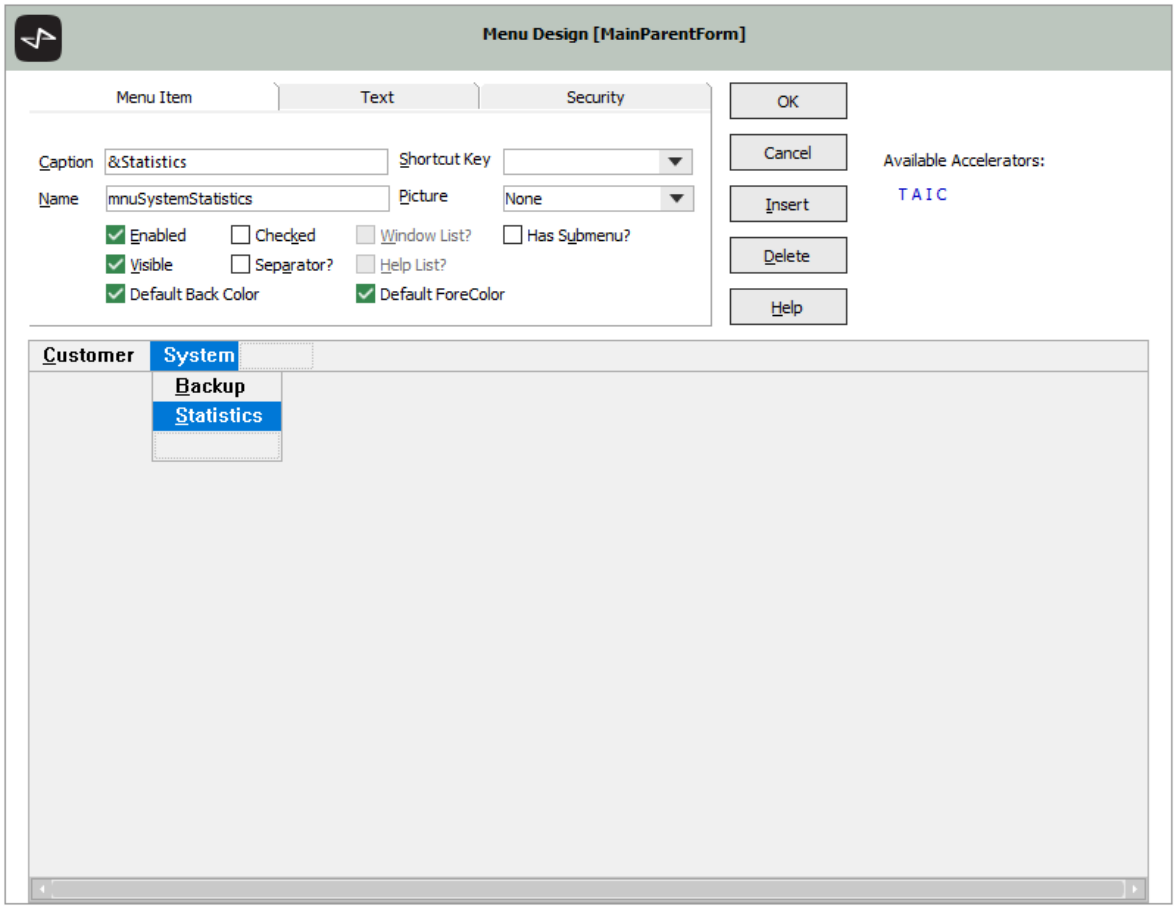
5. Add code to the **formLoad** method for the **StatisticsForm** form to call the **draw** method and subscribe to create and delete notifications on the **Customer** and **BankAccount** classes, as follows.

```
formLoad() updating, protected;

begin
    inheritMethod();

    self.beginClassNotification( Customer, false, Object_Create_Event, Response_Continuous, 0 );
    self.beginClassNotification( Customer, false, Object_Delete_Event, Response_Continuous, 0 );
    self.beginClassNotification( BankAccount, false, Object_Create_Event, Response_Continuous, 0 );
    self.beginClassNotification( BankAccount, false, Object_Delete_Event, Response_Continuous, 0 );

    self.draw();
end;
```

6. Add code to the **formUnload** method for the **StatisticsForm** form, to unsubscribe from the notifications.

7. Add code to the **formSysNotify** method for the **StatisticsForm** form, to redraw the graph by calling the **draw** method.

8. Add a menu item called **mnuSystemStatistics** to the **MainParentForm** form, as shown in the following image.



9. Add code to the **mnuSystemStatistics_click** method, to display the **StatisticsForm** form.

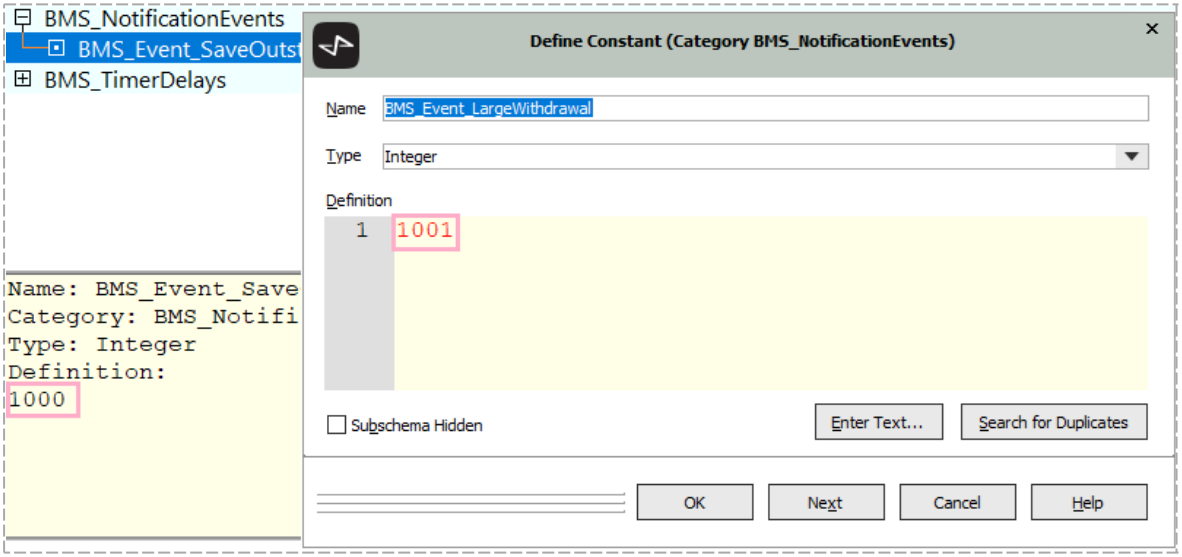
10. Test your notifications, by leaving the **StatisticsForm** form open while you add customers.

Exercise 14.3 – Defining a Global Constant

In this exercise, you will return to the **BankingModelSchema** and add a global constant category called **UserNotifications**, to which you will add a constant called **LargeWithdrawal** that has a value of **20**.

In the next exercise, you will use the **LargeWithdrawal** constant for a user notification.

1. Select **BankingModelSchema** in the Schema Browser.
2. Open the Global Constants Browser by selecting the Browse menu **Global Constants** command.
3. On the Global Constants Browser, select the existing **BMS_NotificationEvents** category.
4. On the Global Constants menu, select the **Sorted by Value** command from the Category submenu.
5. Select the last-existing constant in this category so that you can see the current highest-number notification.
6. On the Global Constants menu, select the **Add** command from the Constants submenu and then add a constant called **BMS_Event_LargeWithdrawal** of type **Integer** and with a value +1 larger than the current largest value.



Exercise 14.4 – Using User Notifications

In this exercise, you will demonstrate user notifications in action by making the following changes.

- In the **BankingModelSchema**, the **BMS_Modify_PerformWithdrawal** section of the **onModify** method of the **BankAccount** class will cause a **BMS_Event_LargeWithdrawal** user event if more than $1,000,000 is withdrawn.
- In the **BankingViewSchema**, the **MainParentForm** form will subscribe to notifications of the **BMS_Event_LargeWithdrawal** user event. The form will respond to the notifications by displaying a message box.
- To test the notifications, you will code a JadeScript method that creates a bank account with a balance of $2,000,000 and which performs a withdrawal of $1,500,000.

This should trigger the display of the message box for any user running the **Banking** application.

To demonstrate user notifications in action, perform the following actions.

1. Open a Class Browser for the **BankingModelSchema** schema.
2. Select the **BankAccount** class.

3. Change the **onModify** method, as follows.

```
onModify( pTA : BankAccountTA ) updating;

begin
    if pTA.modificationCode = BMS_Modify_PerformWithdrawal then
        self.balance -= pTA.transactionAmount;

        if pTA.transactionAmount > 1000000 then
            self.causeEvent( BMS_Event_LargeWithdrawal, false, pTA.transactionAmount );
        endif;

    elseif pTA.modificationCode = BMS_Modify_PerformDeposit then
        self.balance += pTA.transactionAmount;

    endif;
end;
```

4. Open a Class Browser for the **BankingViewSchema** schema.

5. Select the **MainParentForm** form.

6. In the **formLoad** event method, subscribe to notifications of the **BMS_Event_LargeWithdrawal** event, as follows.

```
formLoad() updating, protected;

begin
    app.mdiFrame := MainParentForm;

    self.beginClassNotification( BankAccount, false, BMS_Event_LargeWithdrawal, Response_Continuous, 0 );
end;
```

7. Add code to the **formUnload** event method to unsubscribe from notifications of the **BMS_Event_LargeWithdrawal** event.

Tip Call the **endClassNotification** method.

8. Code the **formUserNotify** method, as follows.

```
formUserNotify( pEventType : Integer; pBankAccount : BankAccount; pEventTag : Integer; pUserInfo : Any ) updating;

vars
    accountDetails : String;
    warningMessage : String;

begin
    accountDetails := "Bank Account Number: " & pBankAccount.number.String;

    warningMessage := "Large Withdrawal $" & pUserInfo.String;

    app.msgBox( accountDetails & CrLf & warningMessage, "Suspicious Transaction", MsgBox_OK_Only );
end;
```

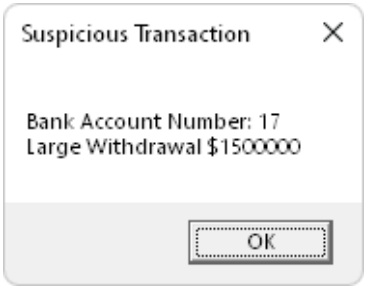
Note Make sure to change the **pObject : Object** parameter to **pBankAccount : BankAccount**.

9. Add a JadeScript method called **makeLargeWithdrawal**, and code it as follows.

```
makeLargeWithdrawal();

vars
    chequeAccountTA : ChequeAccountTA;

begin
    app.initialize();

    create chequeAccountTA transient;
    chequeAccountTA.balance := 2000000;
    chequeAccountTA.overdraftLimit := 0;
    chequeAccountTA.myCustomer := null;
    chequeAccountTA.persistEntity( BMS_Full_update );

    chequeAccountTA.transactionAmount := 1500000;
    chequeAccountTA.persistEntity( BMS_Modify_PerformWithdrawal );

epilog
    delete chequeAccountTA;
end;
```

10. Run the **Banking** application.

11. Execute the **makeLargeWithdrawal** JadeScript method. The **Banking** application should display a message box similar to the following.



Timer Events

Timer events are events that occur after a specified delay. The event can happen on a one-off basis or it can repeat at regular intervals.

Timer events can be used for scheduling purposes; for example, to schedule a nightly backup.

Beginning and Ending a Timer

The **beginTimer** method starts a timer for the **self** object.

```
beginTimer(delay: Integer; option: Integer; eventTag: Integer);
```

The parameters are described in the following table.

Parameter	Description
delay	Time in milliseconds until the timer event occurs.
option	Whether timer notifications are automatically canceled after the first event. Possible values are: ■ **Timer_Continuous** – continue to send timer notifications ■ **Timer_OneShot** – cancel notifications after the first event
eventTag	Value that is returned as part of the timer notification and identifies the timer.

The **endTimer** method stops a timer.

```
endTimer(eventTag: Integer);
```

Responding to a Timer

The **timerEvent** method is invoked when a timer notification is received.

```
timerEvent(eventTag: Integer) updating;
```

Exercise 14.5 – Using a Timer

In this exercise, you will use a timer in the **MainParentForm** form to change its background color every second. The timer will be started in the **formLoad** method and stopped in the **formUnload** method. You will implement the **timerEvent** method for the form.

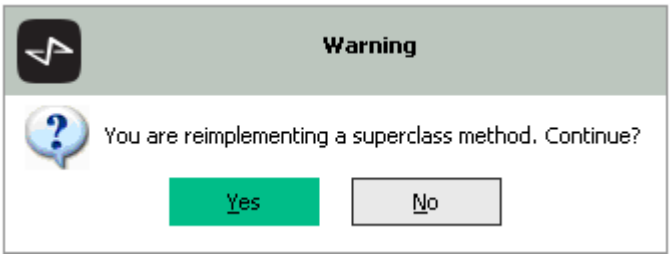
1. Open a Class Browser for the **BankingViewSchema**.
2. Select the **MainParentForm** form.
3. Add an instruction to the **formLoad** method to start the timer, as follows.

```
formLoad() updating, protected;

begin
    app.mdiFrame := MainParentForm;

    self.beginClassNotification( BankAccount, false, BMS_Event_LargeWithdrawal, Response_Continuous, 0 );

    self.beginTimer( 1000, Timer_Continuous, 0 );
end;
```

4. Stop the timer in the **formUnload** method, as follows.

```
formUnload() updating, protected;

begin
    inheritMethod();

    self.endNotificationForSubscriber( self );

    self.endTimer( 0 );
end;
```

5. Add a method called **timerEvent**. A dialog warns you that there is already a method of that name in the **Application** hierarchy. Click the **Yes** button, to continue.



6. Code the **timerEvent** method, as follows.

```
timerEvent( eventTag : Integer ) updating;

begin
    self.backColor := app.random( #FFFFFF );
end;
```

Run the **Banking** application and test that the background color of the **MainParentForm** form changes randomly.

Module 15 Nodes, Processes, and Caches

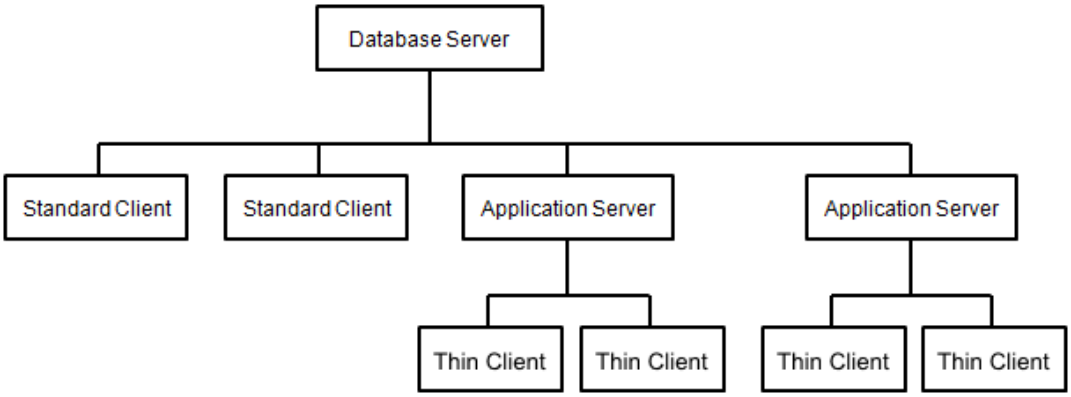
This module contains the following topics.

Introduction

This module contains an overview of the architecture of a Jade system, which is based on the concept of a *node*.

Distributed Processing

The Jade Platform has a distributed processing architecture in which application processing is shared between a single database server and its clients.



The database server:

- Contains the persistent database
- Can execute application code and process objects (that is usually done by clients)
- Accepts connections from standard clients and application servers
- Manages system-wide services such as locking, cache coherency, and notifications

A standard client:

- Connects to the database server
- Displays forms
- Executes application code and processes objects
- Requires a high-bandwidth (LAN) connection to the database server

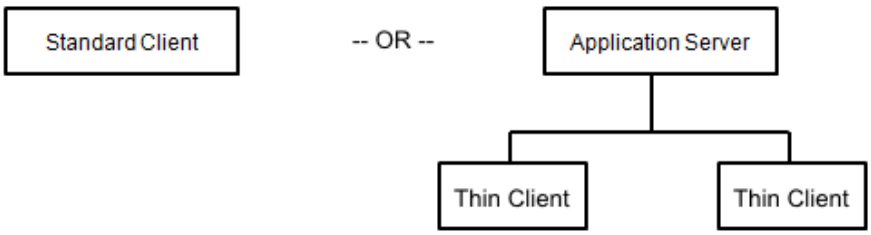
An application server:

- Connects to the database server
- Accepts connections from thin (presentation) clients
- Does *not* display forms
- Executes application code and processes objects for connected presentation clients
- Requires a high-bandwidth (LAN) connection to the database server

A presentation client (also known as a thin client):

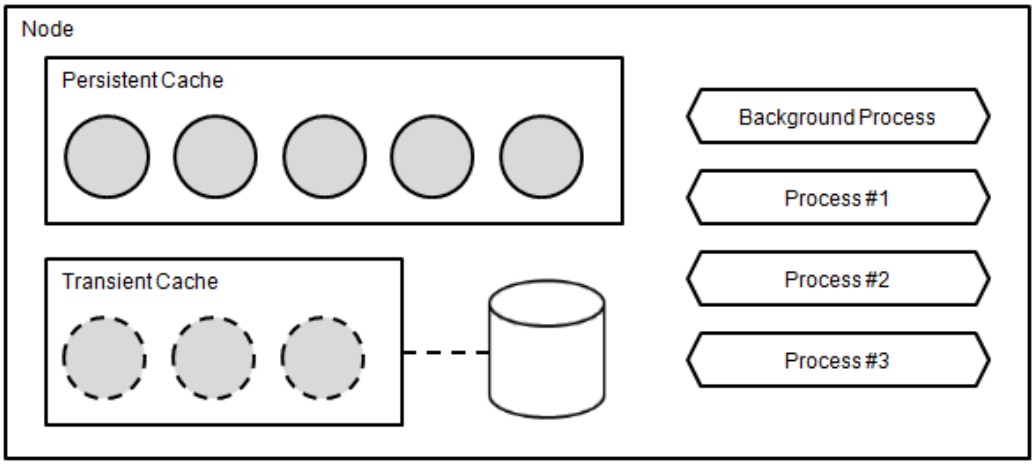
- Connects to an application server
- Does *not* execute application code or process objects (that is done by the application server)
- Does *not* require a high-bandwidth (LAN) connection to the application server

In single user mode, there is no separate database server node. You can run a single standard client or a single application server.



Nodes and Processes

A node is a component of a Jade system where application code is executed and where objects are processed. The following diagram shows the structure of a node.



A number of applications can be executed in the same node, each with its own thread of execution, the Jade term for which is *process*. A node has a background process and a number of other processes; one for each application.

The following parts of the architecture of a Jade system are nodes.

- Standard client, because it executes application code and processes objects.
- Database server, because it can execute methods with the **serverExecution** option in the method signature, and server applications that are specified in the Jade initialization file.

  Note Code executed by the database server must not attempt to display forms and message boxes.

- Application server, because it executes application code and processes objects for connected presentation clients. There is a process for each connected presentation client.

A presentation client is *not* a node, because it does not execute application code or process objects; those functions are carried out by the application server.

Persistent Cache

A node has a persistent cache for persistent objects, which are fetched from the database server. The single persistent cache is shared by all processes in the node. When a process needs a persistent object, it is automatically fetched from the database server into persistent cache, unless it is already present.

When an update transaction is committed, modified objects are copied back to the database server. However, the object remains in persistent cache and is available for subsequent accesses by any process in the node, thereby avoiding fetching the object from the database server again.

Objects that have been updated by another node are discarded from cache using a cache coherency mechanism managed by the database server.

When persistent cache becomes full, the least-recently used objects are discarded. If they are modified and not yet committed, they are sent to the server.

Transient Cache

A node has a single transient cache for process transient objects and shared transient objects, which are created locally in the node. The single transient cache is shared by all processes in the node.

Process transient objects can be accessed only by the process in which they were created. They are removed when the process that created them terminates, or when the process deletes it.

Shared transient objects can be accessed by all processes in the node, but not by a process in a different node. They are removed when the node terminates, or when a process deletes it.

When transient cache is full, it overflows to a transient database on disk. For this reason, you should delete transient objects that are no longer required, because accessing transient objects from disk is much slower than accessing them from memory.

Persistent, Transient, and Shared Transient Objects

A persistent object is stored in the database. It can be accessed by all nodes. You must be in transaction state to create, update, or delete a persistent object.

```
beginTransaction;
// Create, update, and delete persistent objects
commitTransaction;
```

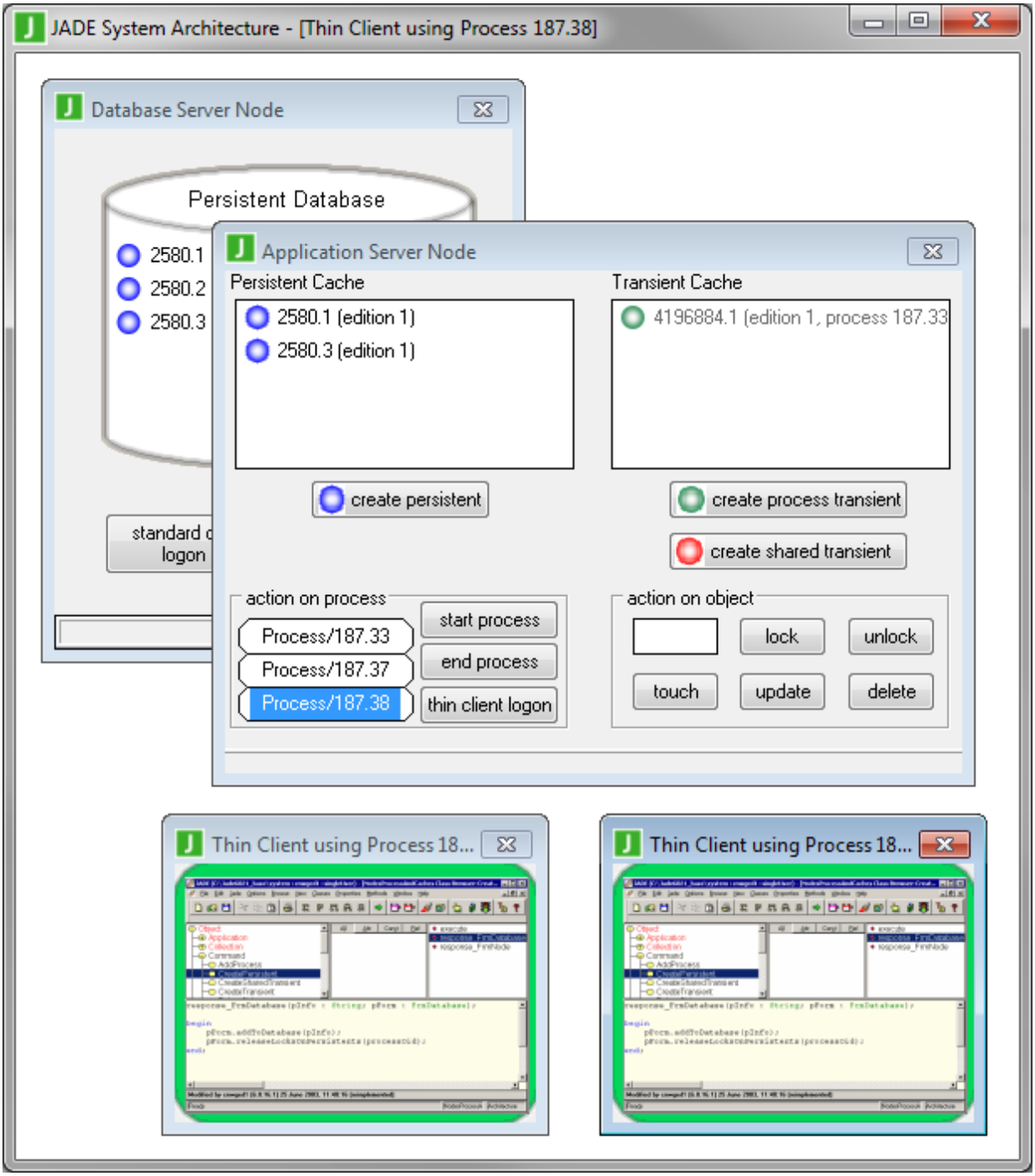
A transient object is stored locally in transient cache. It can be accessed only by the process that created it, and becomes unavailable when that process terminates or when it is explicitly deleted.

A shared transient object is a special type of transient object, which can be accessed by other processes in a node in addition to the process that created it. It becomes unavailable when the node terminates or if it is explicitly deleted. Shared transient objects can be used to safely share information in a multi-threaded application. You must be in transient transaction state to create, update, or delete a shared transient object.

```
beginTransientTransaction;
// Create, update, and delete shared transient objects
commitTransientTransaction;
```

Demonstration

Your instructor will use an example schema to demonstrate the architecture of a Jade system.



Module 16 Transactions and Locking

This module contains the following topics.

Introduction

In a multiuser system, persistent objects are fetched from the database and held in caches on the different nodes. Locking is an important mechanism in controlling whether an object can be updated.

Note Locking an object does not prevent other processes accessing it, but it does prevent them updating it.

Lock a persistent object when you want to:

- Update it

  When more than one process attempts to update the same object, locking determines which process can proceed, because a process must obtain an exclusive lock on an object before it can update it.

- Prevent it from being updated

  An application may require objects to remain unmodified while an operation is carried out; for example, a trial balance in which account objects are locked before reading the balance, to guarantee that the latest edition of each account is used. The locks are held until the trial balance calculation is complete.

You do not need to write a lot of code to explicitly lock objects, because of the implicit locking that occurs with transactions and collections.

Update Transactions

In an updating transaction, a number of persistent object creates, updates, and deletes are performed as a single unit of work. The **ACID** requirements for a transaction are:

- **Atomicity** – operations that make up a transaction must all complete or all fail.
- **Consistency** – database moves from one consistent state to another.
- **Isolation** – intermediate data from one transaction is not visible to a concurrent transaction or query.
- **Durability** – committed transactions survive application software, operating system, and hardware failure.

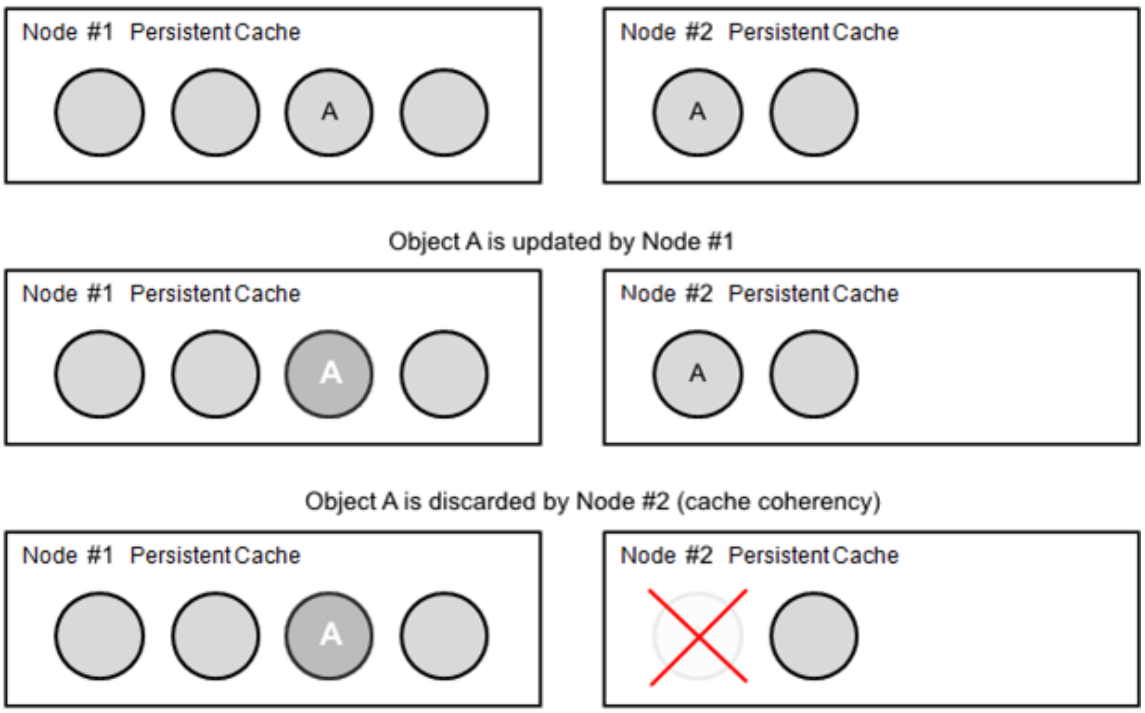
An updating transaction starts with the **beginTransaction** instruction. If the transaction is successful, the **commitTransaction** instruction releases all transaction duration locks and causes the new, updated, and deleted objects to be committed to the database.

If the transaction is *not* successful, the **abortTransaction** instruction releases all transaction duration locks and discards modified objects from persistent cache. The next time the object is required, it is fetched from the database.

Cache Coherency

Cache coherency is a service provided by the database server to assist nodes to discard *stale* objects from caches. A *stale* object is one that has been updated by another node.

The database server maintains a list of objects that are present in the persistent cache of each node and sends messages to the nodes when transactions are committed to the database.



Note Cache coherency messages cannot be sent instantaneously, so you can be sure you have the latest edition of an object only if you lock it.

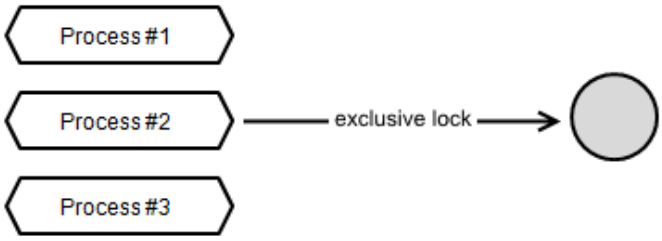
Lock Types

The type of lock you choose to acquire for an object will determine the type of locks other processes can apply to the object while you have it locked. As such, the type of lock determines the type of access one process can have to an object locked by another process.

When you lock an object with any type of lock, the latest edition of the object is fetched from the database server.

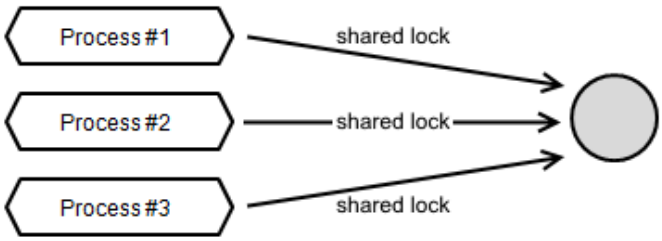
The lock types are:

- Exclusive lock, which is required before an object can be updated.



An attempt to acquire an exclusive lock is made automatically when a property of an object is updated. Other processes cannot apply any type of lock to the object.

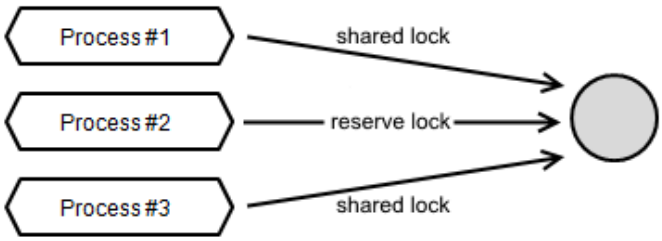
- Shared lock, which prevents other processes from updating the object while it is locked.



Other processes can share lock the same object and one process can reserve lock the object.

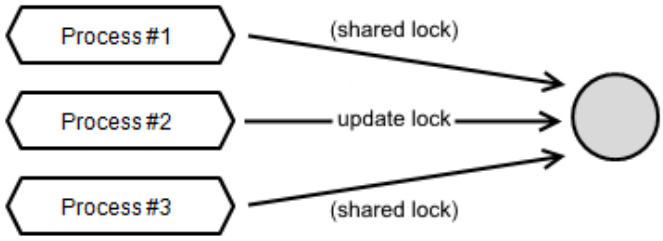
Shared locks are automatically acquired on a collection that is being iterated using a **foreach** instruction, unless the **discreteLock** clause is specified. The shared lock is acquired for the duration of the iteration.

- Reserve lock, which is similar to a shared lock, but with the intention to upgrade to an exclusive lock at some stage.



Shared locks can co-exist with a reserve lock; however, there can be one reserve lock only on the object.

- Update lock, which is an alternative to an exclusive lock, but allows other processes to have shared locks on the object.



The exclusive lock is still required when the updates are committed. If the exclusive lock cannot be obtained, the updates will be discarded.

Lock Durations

The duration of a lock determines when it is released. There are two lock durations, as follows.

- Transaction duration, which is released at the end of a transaction

  All transaction duration locks held for persistent objects are released automatically when the transaction ends (**commitTransaction**, **abortTransaction**, **endLoad**, or **endLock** instruction), even if they were acquired before the transaction began.

  Attempts to manually unlock a persistent object, using the **unlock** method, are ignored in transaction state (after a **beginTransaction**, **beginLoad**, or **beginLock** instruction).

  Transaction duration locks are acquired automatically before a persistent object is updated or deleted.

- Session duration

  Session duration locks are automatically released at the end of a session, when the process that owns the lock terminates. Session locks can also be released earlier, by using the **unlock** method.

  Session duration locks are useful when you need to hold a lock on an object across transaction boundaries. For example, the JADE Painter applies a session lock to a form object when you edit the form. This session lock prevents two users editing a form at the same time and it is held across any transactions that may occur as a result of saving the form.

Locking Methods

The **lock** method, defined in the **Object** class, has the following signature:

```
lock(lockTarget: Object; lockType, lockDuration, lockTimeout: Integer);
```

The **lock** method parameters are as follows.

- **lockTarget** is the object to be locked.
- **lockType** is the type of lock. Possible values are **Exclusive_Lock, Reserve_Lock** or **Share_Lock**.
- **lockDuration** is the duration of the lock. Possible values are **Transaction_Duration** and **Session_Duration**.
- **lockTimeout** is the maximum time to acquire the lock before an exception is raised. Possible values are **LockTimeout_Server_Defined**, **LockTimeout_Immediate**, and **LockTimeout_Infinite**, or a number of milliseconds.

The following code fragments apply a specific lock type. The equivalent **lock** syntax is shown.

```
self.sharedLock(object);
self.lock(object, Share_Lock, Transaction_Duration, LockTimeout_Server_Defined);
```

```
self.exclusiveLock(object);
self.lock(object, Exclusive_Lock, Transaction_Duration, LockTimeout_Server_
Defined);
```

```
self.reserveLock(object);
self.lock(object, Reserve_Lock, Transaction_Duration, LockTimeout_Server_Defined);
```

```
self.updateLock(object);
self.lock(object, Update_Lock, Transaction_Duration, LockTimeout_Server_Defined);
```

The **tryLock** method is an alternative to the **lock** method. It returns **false** instead of raising an exception when a lock request times out. The **tryLock** method has the following signature.

```
tryLock(lockTarget: Object; lockType, lockDuration, lockTimeout: Integer): Boolean;
```

Tip In a lock exception handler, to avoid raising further exceptions use the **tryLock** method instead of the **lock** method.

The **unlock** method is defined in the **Object** class and has the following signature.

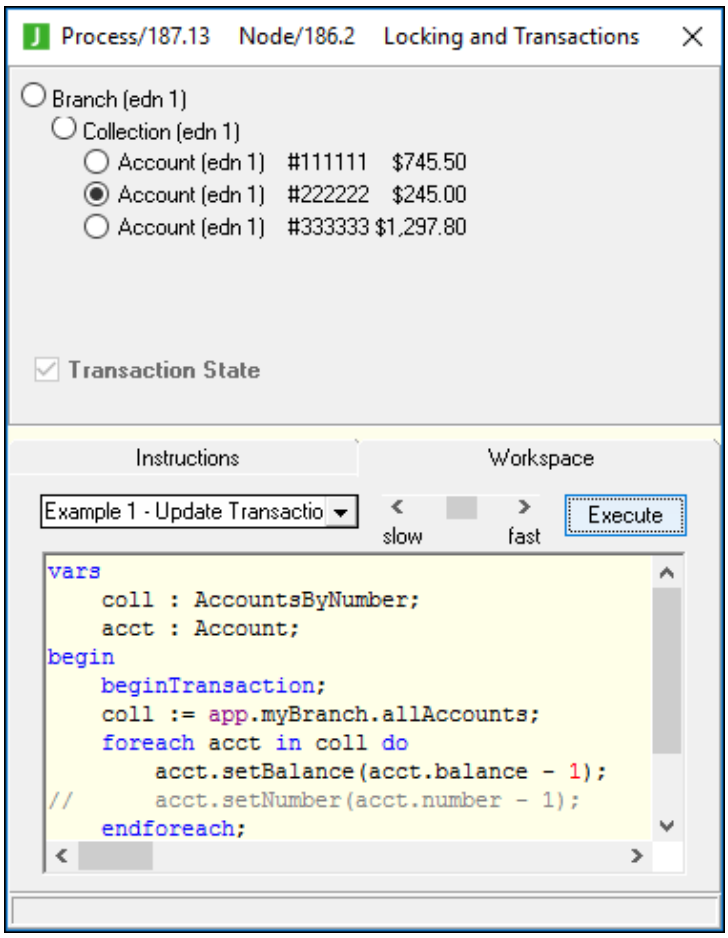
```
unlock(unlockTarget: Object);
```

Attempts to unlock objects inside a transaction are ignored.

Tip Use **abortTransaction** instruction, which can be used even when not in transaction state, to unlock all persistent objects for a process.

Demonstration

Your instructor will demonstrate transactions and locking using a **TransactionsAndLocking** example schema.



Read Transactions

Locking an object brings the latest edition into persistent cache *and* prevents other users from updating it.

A trial balance provides a good example of a read transaction, where locks are used to prevent objects from being updated. In a trial balance, the total of the balances of all accounts is calculated. Each account object should be locked before its balance is read, and the locks released only after the trial balance calculation is complete.

A simple implementation could use the **sharedLock** and **unlock** methods.

```
vars
    total: Decimal;
    account: Account;
begin
    foreach account in accounts do
        self.sharedLock(account);                // Account explicitly locked
        total := total + account.balance;
    endforeach;
    foreach account in accounts do
        self.unlock(account);                    // Account explicitly unlocked
    endforeach;
    write total;
end;
```

A more-efficient implementation uses the **beginLock** and **endLock** instructions. After the **beginLock** instruction, accessing the value of a property (or executing a method) of an object automatically acquires a transaction duration shared lock on the object. The **endLock** instruction releases all locks in a single operation.

```
vars
    total: Decimal;
    account: Account;
begin
    beginLock;
    foreach account in accounts do
        total := total + account.balance;        // Account implicitly locked
    endforeach;
    endLock;                                     // All accounts implicitly unlocked
    write total;
end;
```

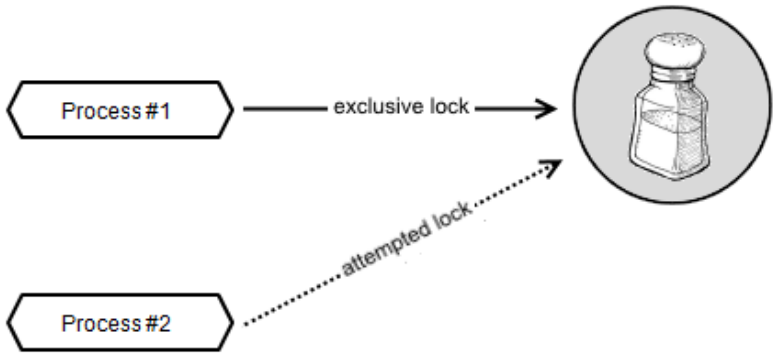
The **beginLoad** and **endLoad** instructions are similar to the **beginLock** and **endLock** instructions, but enable you to selectively lock objects.

```
vars
    total: Decimal;
    account: Account;
begin
    beginLoad;
    foreach account in accounts do
        self.sharedLock(account);                // Account explicitly locked
        total := total + account.balance;
    endforeach;
    endLoad;                                     // All accounts implicitly unlocked
    write total;
end;
```

Lock and Deadlock Exceptions

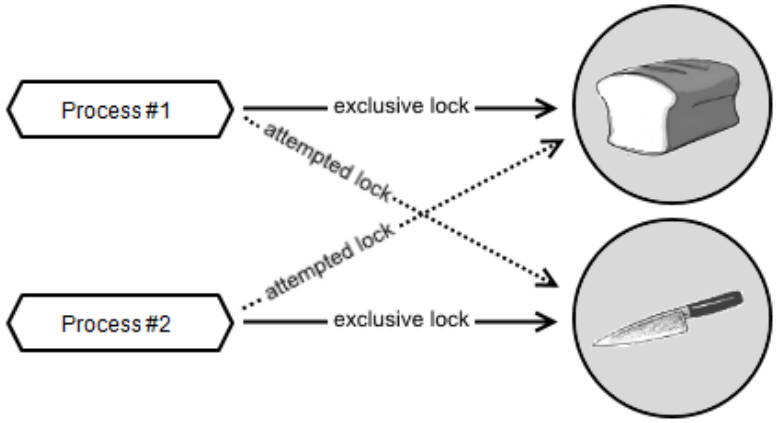
When a lock cannot be obtained (because another process already has the object locked with an incompatible lock), an exception is raised. The following analogies explain the difference between lock exceptions and deadlock exceptions, and the different ways they are handled.

The analogy for a lock exception is two people wanting to add salt to their food at the start of a meal when only one salt shaker available.



One person (**Process #1**) is first to grab hold of the salt shaker. The other person (**Process #2**) is unsuccessful. The failed attempt to grab the salt shaker corresponds to the lock exception. The situation is easily handled by **Process #2** waiting until the salt shaker becomes available. Typical coding of a lock exception handler involves periodically retrying the lock.

The analogy for a deadlock exception is two people wanting to cut a slice of bread for which you need both the loaf and the knife.



If one person (**Process #2**) has the knife and the other person (**Process #1**) has the bread, the strategy of waiting for the other object to become available (which worked for an ordinary lock) leads to an indefinitely long wait and gets you nowhere. The first process to detect the deadlock should give way and release the lock. Alternatively, you can set the **DoubleDeadlockException=true** parameter in the [JadeServer] section of the Jade initialization file and allow the priorities of the processes to determine which process should give way.

Note A deadlock can also arise with a single object, typically a collection where two processes have shared locks on the collection that they attempt to upgrade to exclusive.

Debugging Lock Exceptions

The Jade Platform supports the optional recording of the current call stack when a process locks an object. Any process can retrieve this information while the lock is held; for example, you can use it to help find and resolve locking problems during application development, by tracking down where in the code any long-lived lock was obtained.

This information, which is passed to the lock manager and stored in the lock entry, can be retrieved by any process while the lock is held. When a lock is obtained, the saved information includes each method in the current call stack and the call position (source code offset) within each method. You can use this information to produce a call stack summary similar to that shown when you click the **Debug** button on the Unhandled Exception dialog.

Notes The values of local variables are not available, as the code is no longer executing.

This feature is intended for you to use when developing and testing applications. Because of the overhead involved in capturing and saving the extra information, we do not expect that this feature is permanently enabled in a production system.

Automatically enable the debugging of lock exceptions for all client processes on startup, by specifying the **DefaultProcessSaveLockCallStack** parameter with a value of **true** in the [JadeClient] section of the Jade initialization file. To enable the automatic debugging of exceptions for server applications on the database server, specify this parameter and value in the [JadeServer] section of the Jade initialization file. (The default value is **false** on both client and server nodes.)

In addition, the Jade:

- **Object** and **Process** classes provide methods that enable you to dynamically enable and manage the debugging of lock exceptions for a process.
- Monitor **Users** view provides the **Enable Save Lock Call Stack** and **Disable Save Lock Call Stack** commands in the popup menu when you right-click on a user, and the **Locks** view provides the **Show Lock Call Stack** command in the popup menu when you right-click on a locked option.

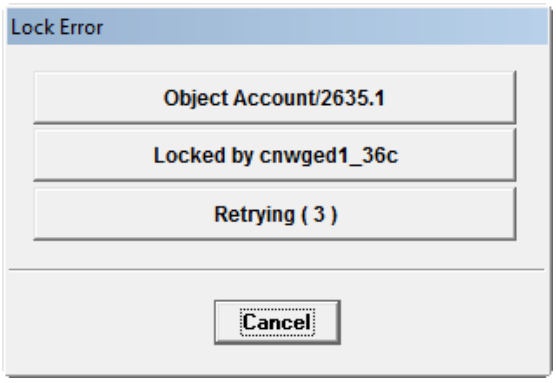
Lock Exception Object

When a lock attempt fails, a lock exception is raised and a lock exception object is created. The lock exception object is an instance of the **LockException** class and is passed as a parameter to any lock exception handler you may have armed.

The lock exception object provides information about the nature of the lock exception that has occurred, and it contains the information listed in the following table.

Property or Method	Description
lockDuration property	Duration of failed lock attempt
lockTimeout property	Timeout value of failed lock attempt
lockType property	Type of the failed lock attempt
retryCount property	Number of times the lock has been retried
targetLockedBy property	Process that has locked the object
lockTarget method	Object that is the target of the failed lock attempt
retryLock method	Retries lock operation and increments **retryCount**

You can write a lock exception handler, but there is one called **globalLockException** provided in the **Application** class. It displays the Lock Error dialog and continues to retry the lock until the user clicks the **Cancel** button.

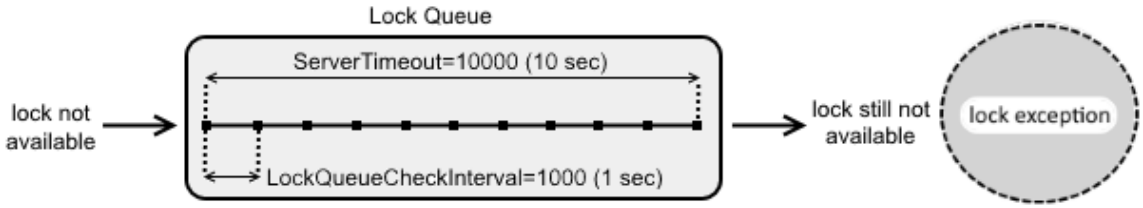


You would arm a lock exception handler globally when the application starts, as follows.

```
initialize() updating;

begin
    on LockException do app.globalLockException(exception) global;
end;
```

Queued Locks

When a process attempts to lock an object, the lock is acquired immediately unless there are incompatible locks, in which case the lock request enters the lock queue.

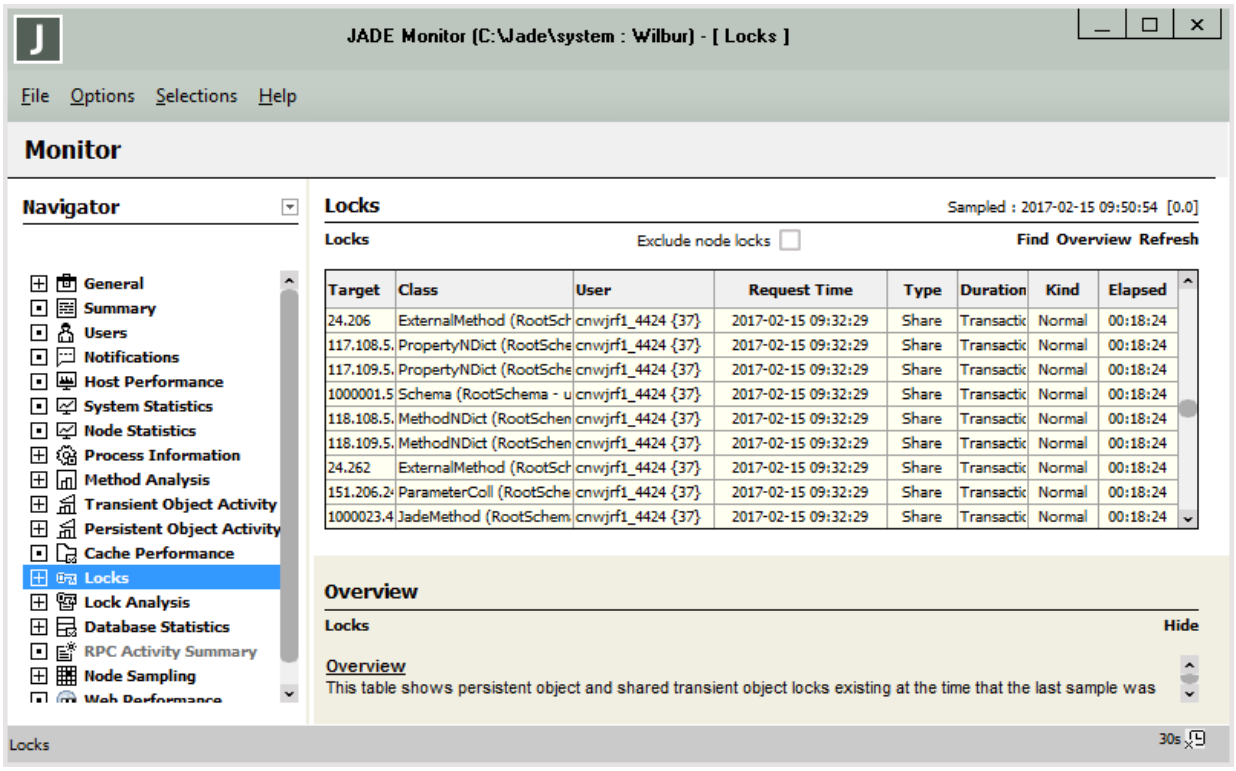


The lock queue is checked when an object is unlocked. It is also checked periodically, at an interval specified by the value of the **LockQueueCheckInterval** parameter in the [JadeServer] section of the Jade initialization file.

If the lock is not acquired by the end of the timeout period, the lock request is removed from the queue and a lock exception is raised (or **false** is returned for the **tryLock** method).

Monitoring Locks

The JADE Monitor utility enables you to view locks already acquired and locks pending in the lock queue.



Shared Locks on Collections

A lock on a collection prevents objects being added to or removed from the collection. (A lock on a dictionary prevents changes to key values of member objects). However, a lock on a collection does not prevent updates to member objects.

When a collection executes a non-updating method (for example, the **size** method), a shared lock is automatically acquired on the collection, to ensure that the latest edition of the collection is used. The lock is released after executing the method, unless the process is in transaction state, load, or lock state.

By default, the **foreach** instruction acquires a shared lock on the collection being read, to prevent the collection being changed during the iteration. The lock is released after the **endforeach** instruction, unless the process is in transaction, load, or lock state.

Shared Transient Objects

Persistent objects are shared by all processes across all nodes in the system.

Transient objects are not shared at all. They are local to the process that created them and they are deleted when the process terminates.

Shared transient objects are shared by all processes within the node that created them and they exist for the lifetime of the node. Concurrency control is enforced by the node in which they live.

Updates to shared transients must be done within a transient transaction, which is similar to a persistent transaction, as shown in the following code fragment example.

```
beginTransientTransaction;
    create object sharedTransient;
commitTransientTransaction;
```

Shared transient objects are locked using the same methods as for persistent objects, and the same implicit locking occurs for transactions and collections.

A significant difference between transient and persistent transactions is that transient transactions cannot be rolled back. If a transient transaction is aborted, any transaction locks are released but the state of the updated objects remains as it was at the point the transaction was aborted.

Exercise 16.1 - Using Locking to Check Editions

In this exercise, you will modify the **CustomerTA** class to check the edition when updating an existing customer.

The update will be allowed to proceed only if the edition is unchanged, which ensures that the customer has not been updated in the interim. If the edition has changed, a message box will be displayed and the **CustomerDetailsForm** form will be reloaded with the latest edition of the customer.

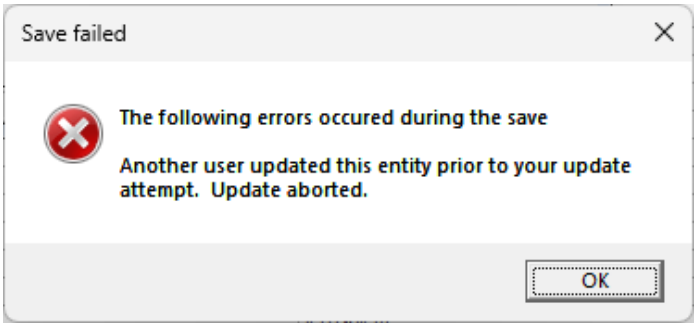
Finally, you will test the edition, checking by opening two **CustomerDetailsForm** forms for the same customer and then updating the customer on each.

1. Select the **CustomerTA** class in a Class Browser opened against the **BankingModelSchema**.
2. Add a method called **checkEdition** and code it as follows.

```
checkEdition( pExpectedEdition : Integer ) : Boolean protected;

begin
    return pExpectedEdition = null                              // No edition provided
        or self.getModelObject() = null                         // Creating new Customer
        or pExpectedEdition = self.getModelObject().edition()   // Is the edition as expected?
        ;
end;
```

3. Run the **Banking** application and then open the **CustomerListForm** form.
4. Select **Charles Piggott** and then click the **Edit** button twice.
5. On the first **CustomerDetailsForm** form, change the name to **Charles Smith** and then click the **OK** button.
6. On the second **CustomerDetailsForm** form, change the name to **Charles Jones** and then click the **OK** button.

   The following message box should then be displayed.



Module 17 Printing

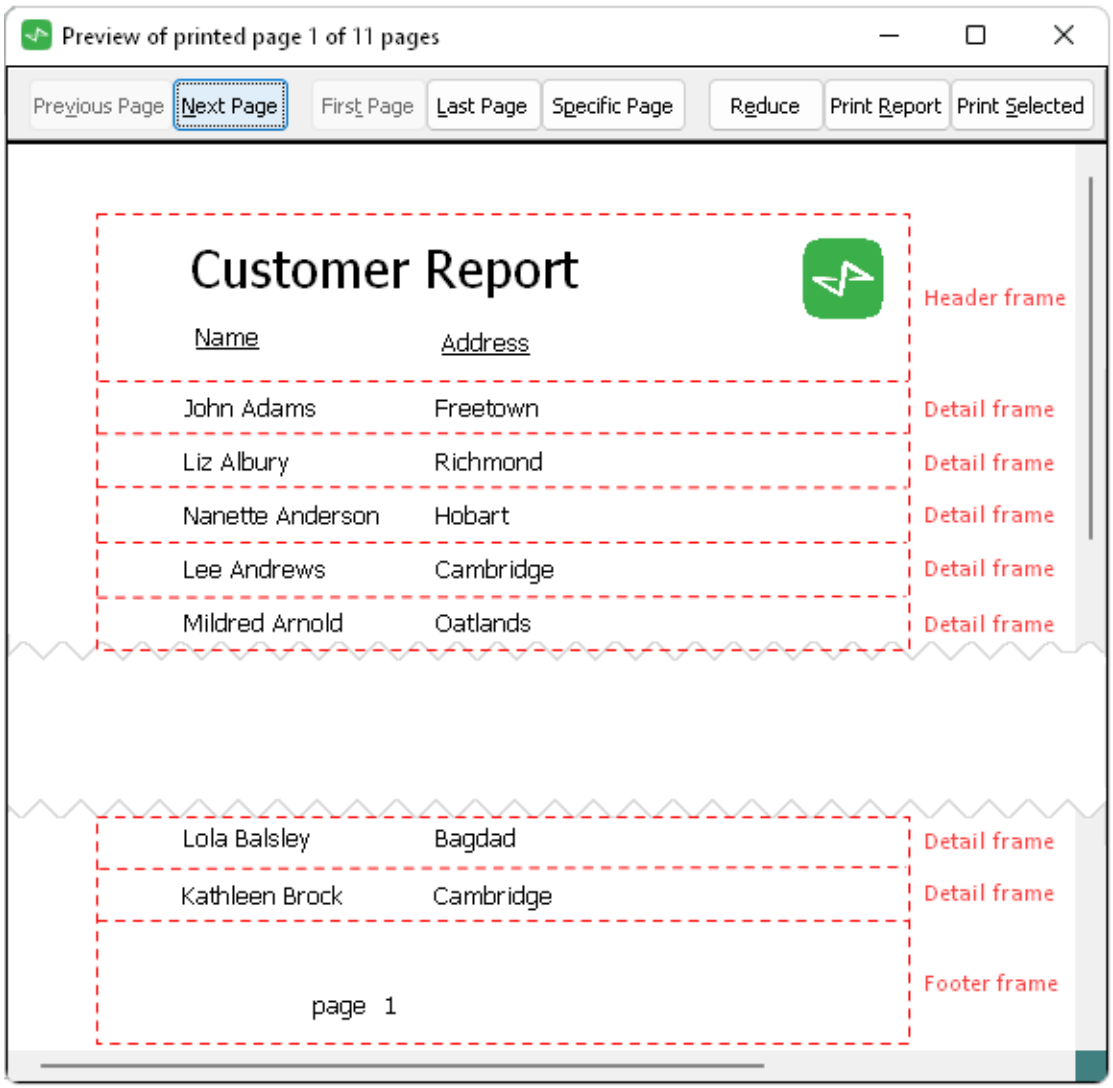
This module contains the following topics.

Introduction

Design reports in the JADE Painter in a similar way to designing forms for a GUI desktop application. A report form has a number of frame controls, which are the basic unit to be printed.

The frames specified in code as the *header* and *footer* frames are automatically printed at the top and bottom, respectively, of every page. Other frames (for example, a detail frame and summary frames) are printed in the sequence specified in the code. For a customer listing report, a detail frame would have labels with captions that are set before printing to the data from a **Customer** object.

The following image shows the print preview output from a customer report. The space between the *header* frame at the top of the page and the *footer* frame at the bottom of the page contains several *detail* frames, which display information for a single customer.



The **Printer** class from the **RootSchema** contains properties and methods that enable you to print a report that you designed in the JADE Painter.

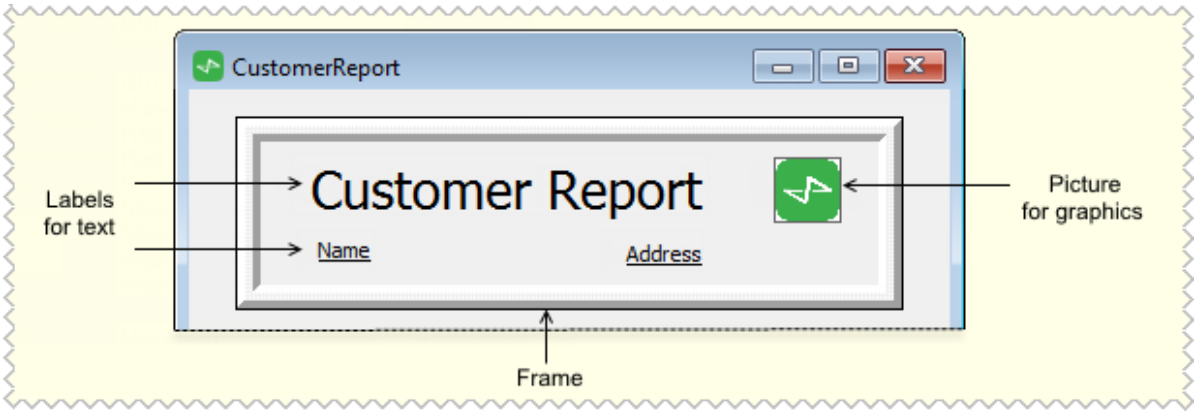
Designing a Report

The controls in the JADE Painter that are typically used in report design are as follows.

- Frame
- Label
- Picture

The **Frame** control, which is the basic unit for printing, contains the other controls.

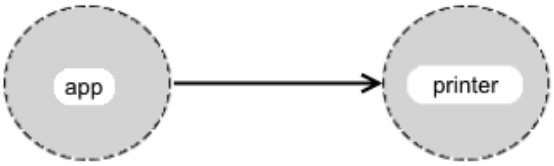
The following diagram shows a header frame containing three labels for text and a picture control for the company logo.



Printer Object

You can create a transient instance of the **Printer** class, which you should delete when the printing is finished.

Alternatively, you can use the instance that is automatically created along with the application object and that is referred to in your code as **app.printer**.



Printer Methods

The following methods and properties are defined for the **Printer** class in **RootSchema**.

Method or Property	Example
setMargins method	Specifies the paper orientation followed by the top, bottom, left, and right margins in millimeters.
	`app.printer.setMargins(Print_Portrait, 10, 10, 10, 10);`
setHeader method	Specifies the report frame to be printed at the top of the page.
	`app.printer.setHeader(fraHeader);`
setFooter method	Specifies the report frame to be printed at the bottom of the page.
	`app.printer.setFooter(fraFooter);`

Method or Property	Example

print, **abort**, and **close** methods	The **print** method prints the specified frame and returns an integer value, which shows whether the user has clicked the **Cancel** or **Stop** button.
	■ If the **Cancel** button is clicked, the **abort** method discards the print buffer, so a print file is not created.
	■ If the **Stop** button is clicked, the **close** method closes the print buffer and sends it to the printer.
	`result := app.printer.print(fraDetail);`
	`if result = Print_Cancelled then`
	`  app.printer.abort();`
	`  break;`
	`elseif result = Print_Stopped then`
	`  app.printer.close();`
	`  break;`
	`endif;`

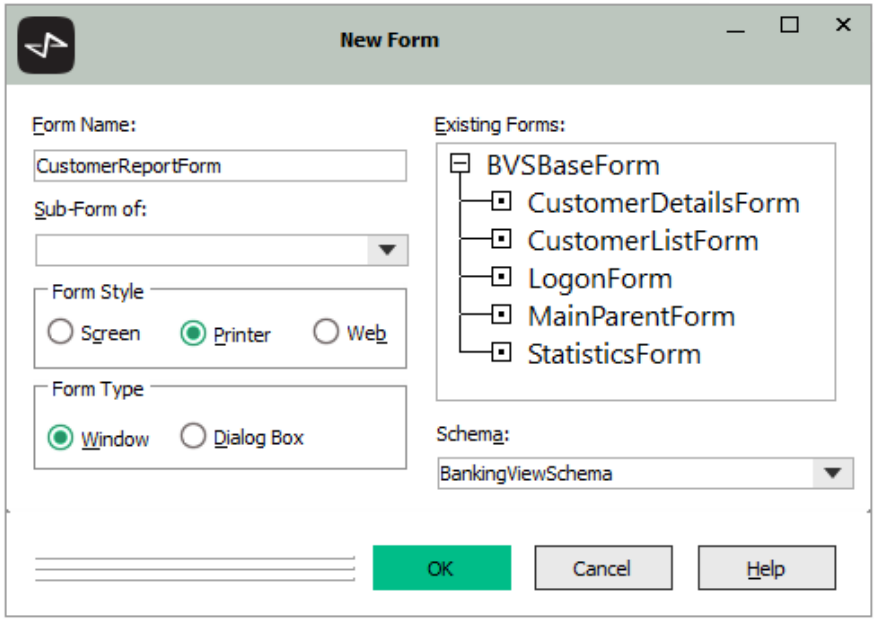
frameFits and **newPage** methods	Returns **true** if the specified report frame fits on the current page. The **newPage** method causes printing to skip to the next page.
	`if not app.printer.frameFits(fraDetail) then`
	`  app.printer.newPage();`
	`endif;`

printActive method	Prints the currently active form. This is effectively a screen snapshot. `app.printer.printActive(self);`
pageNumber property	The page number, which is automatically incremented unless **app.printer.autopaging** is set to **false**. `app.printer.pageNumber := 6;`
pageBorderWidth property	Sets the width of the border in points. `app.printer.pageBorderWidth := 1;`
printPreview property	Specifies if printed output is first displayed on screen or sent directly to the printer. `app.printer.printPreview := true;`

Exercise 17.1 - Adding a Customer Report

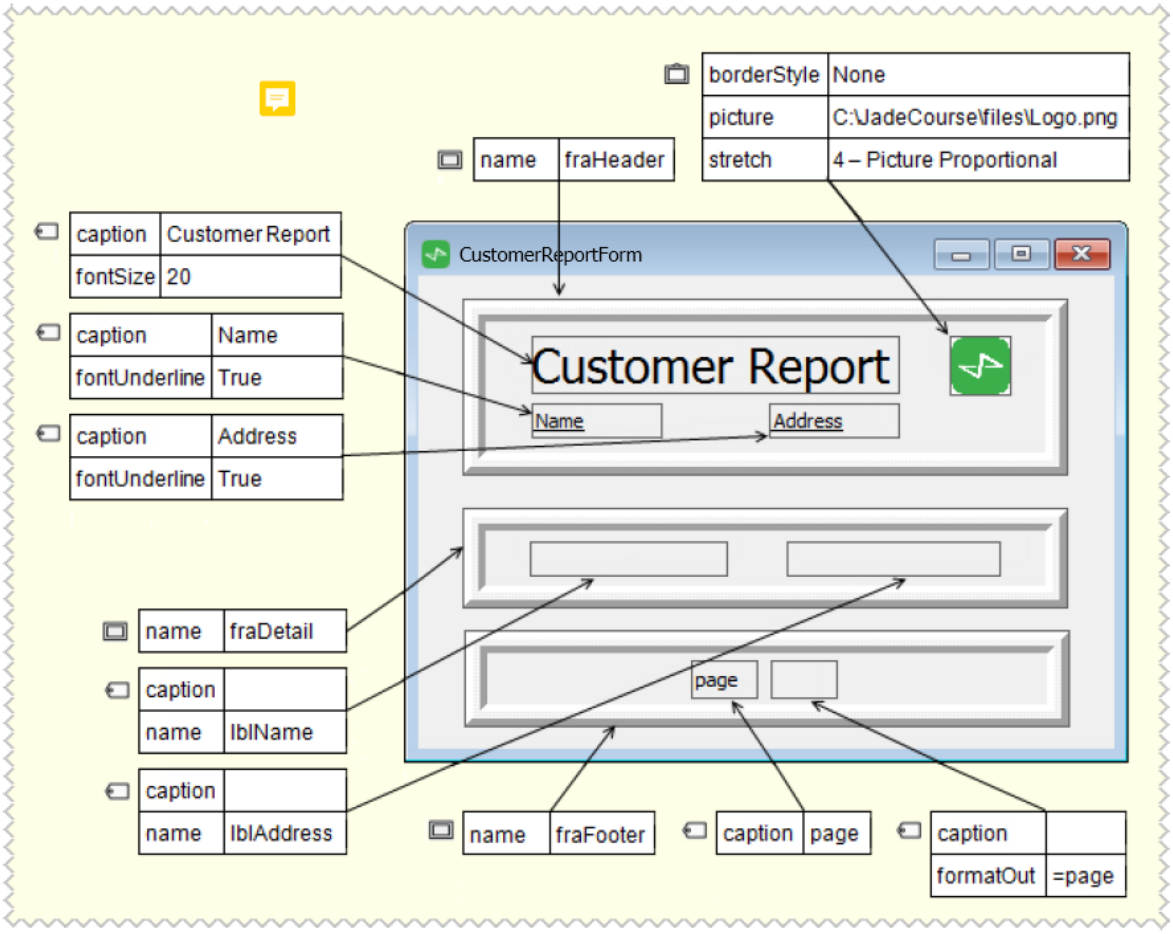
In this exercise, you will add a **CustomerReportForm** form in the JADE Painter.

1. Open the JADE Painter.
2. Select the File menu **New Form** command.



3. Enter **CustomerReportForm** as the name of the form and then select the **Printer** option as the **Form Style**.

4. Paint the report with **Frame** controls, **Label** controls, and a **Picture** control, as shown in the following diagram.



Exercise 17.2 - Coding a Customer Report

In this exercise, you will add a method called **print** to the **CustomerReportForm** class. This method will print a report using the root object's collection of all customers.

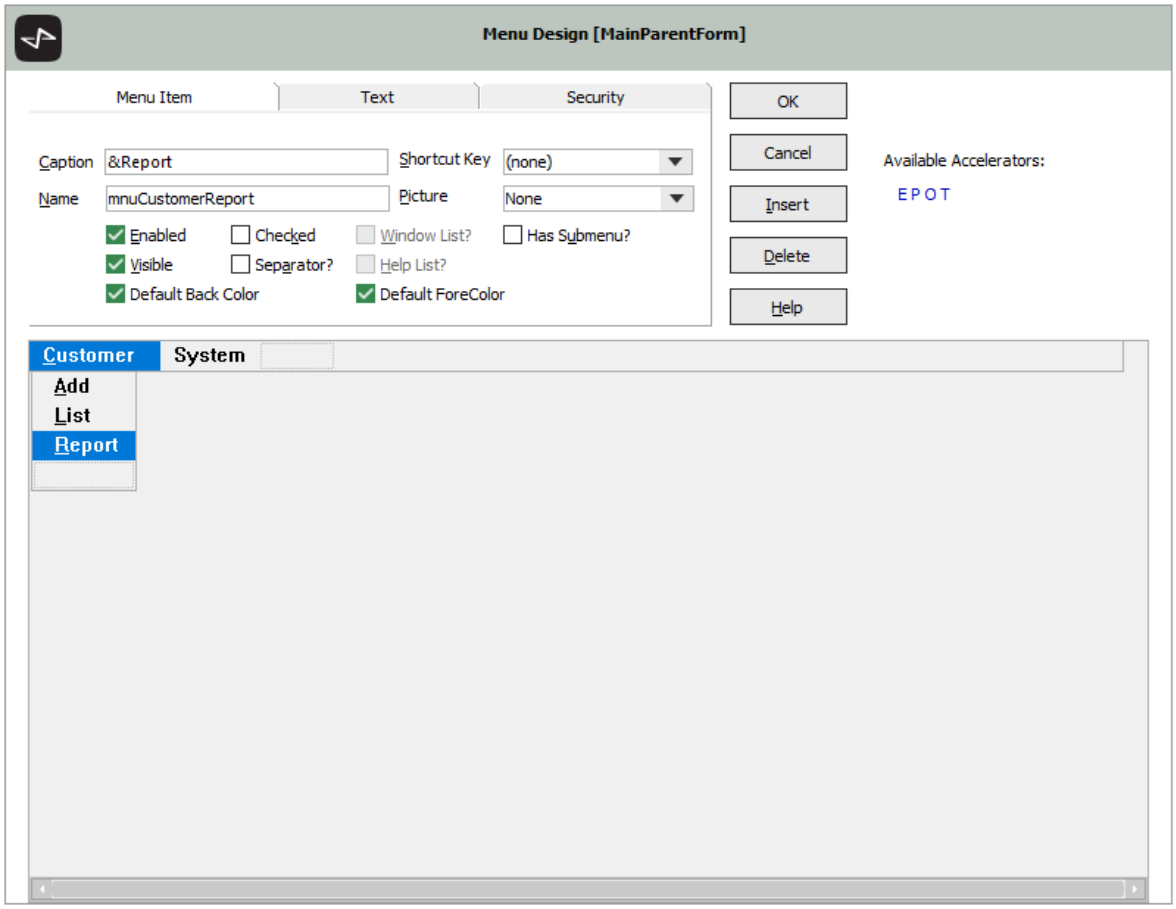
You will then add an option to the **Customer** menu on the **MainParentForm** form to print the **CustomerReport**.

1. In the **CustomerReportForm** class, add a method called **print**.
2. Code the **print** method as follows.

```
print();

vars
    customer : Customer;
    result : Integer;

begin
    app.printer.printPreview := true;
    app.printer.setMargins( Print_Portrait, 10, 10, 10, 10 );
    app.printer.setHeader( self.fraHeader );
    app.printer.setFooter( self.fraFooter );

    foreach customer in app.myBank.allCustomersByLastName do
        self.lblName.caption := customer.firstNames & " " & customer.lastName;
        self.lblAddress.caption := customer.address;

        result := app.printer.print( self.fraDetail );

        if result = Print_Cancelled then
            app.printer.abort();
            break;

        elseif result = Print_Stopped then
            app.printer.close();

        endif;

    endforeach;

    app.printer.close();
end;
```

3. Open the **MainParentForm** form in Painter.
4. Open the menu designer by selecting the File menu **Menu Design** command.

5. Select the empty menu item cell under the **Customer** menu and then enter **&Report** in the **Caption** field and **mnuCustomerReport** in the **Name** field.



6. Click the **OK** button to close the menu designer, and then save the form.

7. In the Class Browser, select the **mnuCustomerReport** menu item and then select the **click** method.

8. Code the method as follows.

```
mnuCustomerReport_click(menuItem: MenuItem input) updating;

vars
    customerReportForm : CustomerReportForm;

begin
    create customerReportForm transient;

    customerReportForm.print();

epilog
    // Clean up our transient to avoid a transient leak
    delete customerReportForm;
end;
```

9. Run the **Banking** application and then view the report.

Developer's Course
Evaluation Form

jade platform

Your feedback is important to our ongoing improvement.

Name

Company

Level

Too low ○ ○ ○ ○ ○ Too high

Pace

Too slow ○ ○ ○ ○ ○ Too fast

Relevance to your work

Low ○ ○ ○ ○ ○ High

Environment

Poor ○ ○ ○ ○ ○ Good

Notes

Poor ○ ○ ○ ○ ○ Good

Instructor

Poor ○ ○ ○ ○ ○ Good

›

Developer's Course
Evaluation Form

jade platform

Most useful topics

Least useful topics

Additional topic suggestions

Other comments

Thank you for providing us with your feedback.
We look forward to seeing you again soon.

Additional Modules

The modules additional to the legacy modules in the Developer's course are as follows.

- Audit Access
- Automated Test Code Generator
- Development Environment
- .NET Exposures
- Internationalization
- Jade Interfaces
- Jade Skins
- Logical Certifier
- Multithreading
- Report Writer
- REST Services
- Security
- Unit Testing
- Version Control
- Web Sockets
- XML in Jade

Audit Access

As Jade journals are designed for optimized performance and therefore not human-readable, the **JadeAuditAccess** class is provided as a tool for the extraction of useful information from Jade journals.

Note This module assumes you have completed the Jade Platform Developer's course and therefore have a copy of the **Banking** system and that you have used it, generating journals. If this is not the case, simply download it from https://github.com/jadesoftwarenz/JADE-Banking-Schema.

Jade Journals

Journals provide several functions necessary to the smooth and consistent running of a Jade database, including:

- Recovering the database to a consistent state after a crash.
- Undoing the effects of aborted transactions.
- Rolling-forwards after a restore from backup.
- Maintaining a redundant duplicate of the database when using the Synchronized Database Service (SDS).

As there are important performance concerns with journaling every database action, it is not safe to access the current journal, and journals are stored on disk in a manner that is not only not human readable, but it is not in any documented format. It is therefore not recommended that you try to read the journal files directly.

These journals, however, contain a wealth of useful information, including:

Information	Description
Audit	Which properties of which objects have changed, and what are their before and after states?
Data lifetimes	How often class properties change. Which properties change often, and which are mostly static?
Diagnostics	What exactly was the scope of a specific transaction; that is, what data did it change?
Extraction	The objects that have been modified in a specified period.
Performance	How long did transactions take to process, and how many are being processed?
Security	Who initiated specific transactions or any transaction that changed a specific object or property?

The journals can also be useful for the following.

Benefit	Description
Independence	Journals are a source that can be kept separate from the database, and therefore can provide access to information about the database without relying on application code.
Understanding	The information provided in journals can provide insight into data relationships and business procedures.

As such, there is benefit in the ability to extract this useful information out of the journals and present it in a human readable format. The **JadeAuditAccess** class of **RootSchema** provides this ability.

Description Files

To convert the journal files to a human readable format, a Database Description File (DDF) is used to interpret the structure of the class objects.

Whenever the format of a persistent object changes (for example, adding or removing a property of a class with at least one instantiated object), the structure of the journal changes. These changes also require a reorganization of the database itself, so an easy way to generate the required description files is to add the following parameter to your Jade initialization file.

```
[PersistentDb]
UseJournalDescriptions=true
```

Specifying this parameter with a value of true causes a new description file to automatically be generated after every reorganization.

You may sometimes need to generate a description file manually. For example, creating a new class in the database changes the journal structure but does not require a reorganization and therefore does not automatically generate a new description file. You can also generate an initial description file if the **UseJournalDescriptions** parameter has been set to **false**, by using the **JadeAuditAccess** class **generateDescription** method. This method requires no parameters and generates a description file in the default location (that is, the **journals** folder within the **system** directory of your Jade Platform installation).

```
makeAuditDescription();

vars
    auditAccess : JadeAuditAccess;
begin
    create auditAccess transient;

    beginTransaction;
    auditAccess.generateDescription();
    commitTransaction;
epilog
    delete auditAccess;
end;
```

Note As the **JadeAuditAccess** class **generateDescription** method updates the database and is audited, it therefore must be performed inside a transaction state.

Using the JadeAuditAccess Class to Read Journals

The main steps to read journals with Audit Access functionality are as follows.

1. Locate the first journal
2. Move between journals safely
3. Parse information out of each journal

The challenge with locating the first journal is that it cannot be guaranteed that the journals start at **1**, as they typically start in the range **20** through **23**, depending on the version of the Jade Platform you are using. However, in most cases, only a subset of journals is audited and, in this case, the first journal to be audited can simply be passed in the **pJournalNumber** parameter of the **JadeAuditAccess** class **getJournal** method.

The **getJournal** method parameters are as follows.

- **pDirectory**, which is a **String** value specifying the directory containing the journals. This can be manually specified, or it can be obtained from the **getCurrentJournalDirectory** method of the **JadeDatabaseAdmin** class.
- **pJournalNumber**, which is an **Integer** value specifying the number of the journal to open.
- **pRecordOffset**, which is an **Integer IO** value that is set to the position in the file after the header is skipped.

The **getJournal** method returns zero (**0**) if the journal is available, or exception 3125 (*A required transaction journal was not found*) is raised if it is not found.

Moving between journals is easy, as the **JadeAuditAccess** class **getNextJournal** method retrieves the next journal, although it does depend on a journal already being open (that is, the **getJournal** method must be called at least once).

The **getNextJournal** method returns zero (**0**) if an available next journal exists, or raises exception 3036 (*The database file being opened is required but was not found*) if there is no next journal. The most common strategy for this case is to assume that there will be a next journal, and simply handle the exception, if not.

When a journal is opened, the **JadeAuditAccess** class **getNextRecord** method can be used. This method requires several **output** parameters, and it update these parameters with the data from the journal as it is parsed.

Note **output** parameters are used to pass information back from the called method to the calling method, and as such, should not be initialized in the calling method. For more details, see "Parameters", in Chapter 1 of the *Developer's Reference*.

The parameters (which are all output parameters) of the **getNextRecord** method are as follows.

Name	Type	Description
pType	Integer	Represents the type of action recorded in the entry, using the **JadeAuditAccess** class constants; for example, **51** is **Jaa_Type_ BeginTransaction**.
pObjectType	Integer	Represents the type of object recorded in the entry, using the **JadeAuditAccess** class constants; for example, **9** is **Jaa_Object_ Collection**.
pRecordOffset	Integer	Current position in the journal file.
pTimestamp	TimeStamp	TimeStamp of when the entry was generated.
pSerialNumber	Decimal	Audit serial number of the entry.
pTransactionId	Decimal	Unique identifier for the transaction.
pOid	String	Object identifier (oid) of the object in the entry.
pClassNumber	Integer	Class number of the object in the entry.
pEdition	Integer	Update count of the object in the entry.

Exercise 1 – Creating a Description File

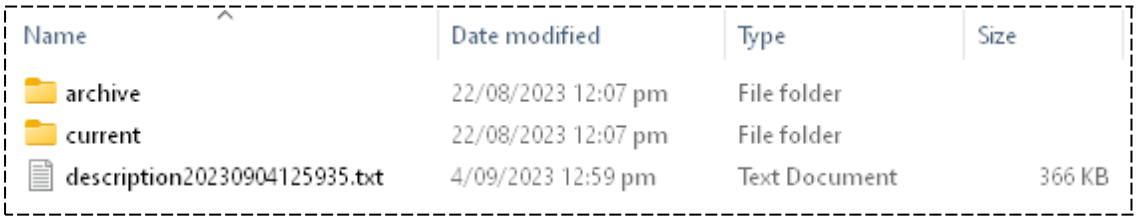
In this exercise, manually create a description file for your database's journals and set the Jade initialization file (**jade.ini**, by default) parameter required to have it automatically create description files on reorganization.

1. Create a schema called **AuditSchema** and open it in the Class Browser.
2. Add a JadeScript method called **createAuditDescription** and code it as follows. (Use the F4 shortcut key to find the **JadeScript** class in the Class Browser.)

```
createAuditDescription();

vars
    auditAccess : JadeAuditAccess;
begin
    create auditAccess transient;

    beginTransaction;
    auditAccess.generateDescription();
    commitTransaction;

epilog
    delete auditAccess;
end;
```

3. Run the method and then navigate to the journals directory (***install-dir*\system\journals**, where ***install-dir*** is the location of your Jade Platform installation).

You should see that a description file like the following has been generated.



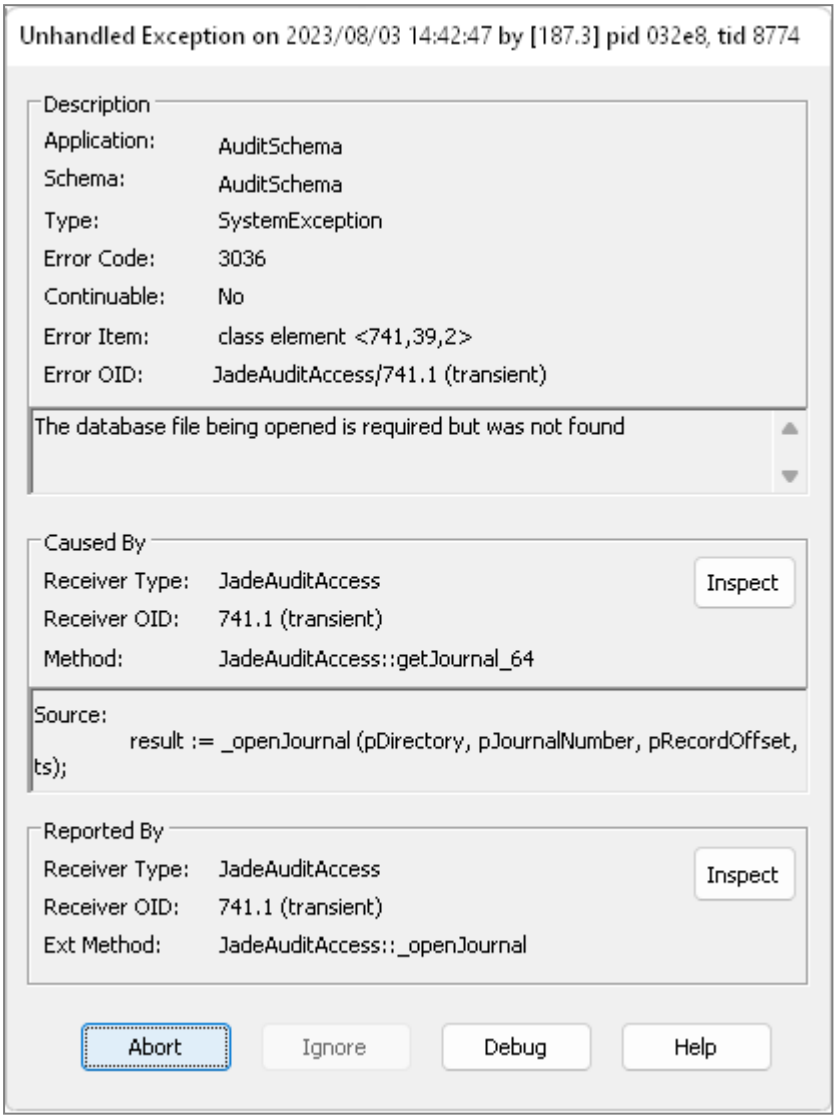
Exercise 2 – Opening a Journal

In this exercise, use the **JadeAuditAccess** class to open a journal and then retrieve various information about the journal.

1. Add a JadeScript method called **openJournal** and code it as follows.

```
openJournal();

constants
    FirstJournal = 1;
vars
    auditAccess : JadeAuditAccess;
    dbAdmin     : JadeDatabaseAdmin;
    offset      : Integer;
begin
    create auditAccess transient;
    create dbAdmin transient;

    auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, FirstJournal, offset);

    write auditAccess.getJournalName;
    write auditAccess.getJournalNumber;
    write auditAccess.getJournalPath;

epilog
    delete auditAccess;
    delete dbAdmin;
end;
```

2. Run the method, which should raise the following exception.



3. Navigate to your journal directory in your file system (***install-dir*/system/journals/current**).

4. Change the **FirstJournal** constant value in your **openJournal** method to the first journal in the directory. (In the following example, the first journal number is 128.)

5. Run the method again. The journal name, number, and location should be written to the Jade Interpreter Output Viewer.

Exercise 3 – Reading an Entry from a Journal

In this exercise, extend the **openJournal** method to read the first entry from the first journal.

1. Add the following to the variable declaration (**vars**) section of the **openJournal** method.

```
openJournal();

constants
    FirstJournal = 128;
vars
    auditAccess : JadeAuditAccess;
    dbAdmin     : JadeDatabaseAdmin;
    offset      : Integer;

    // output parameters for getNextRecord
    entryType           : Integer;
    entryObjectType     : Integer;
    entryTimeStamp      : TimeStamp;
    entrySerialNumber   : Decimal[12];
    entryTransactionID  : Decimal[12];
    entryOID            : String;
    entryClassNumber    : Integer;
    entryEdition        : Integer;
```

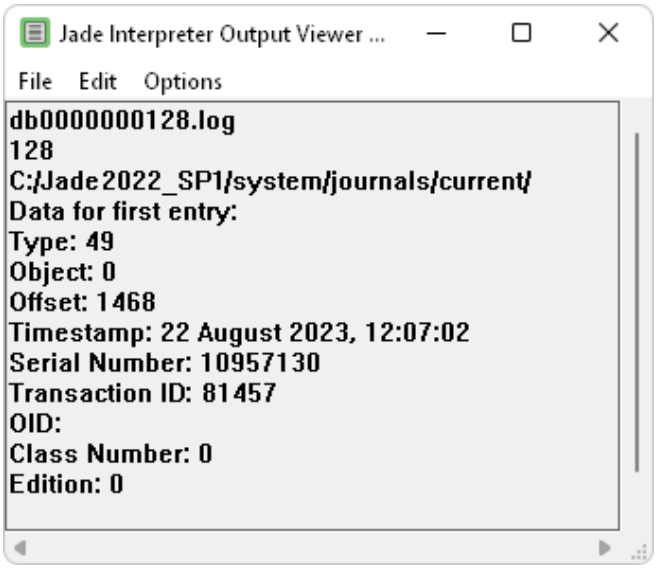
Tip When calling a method that requires a lot of parameters, it can be a good idea to put each parameter on a separate line to aid readability.

Jade uses semicolons to detect the end of a statement, so you can use line breaks as needed.

2. Modify the **openJournal** method, as follows.

```
begin
    create auditAccess transient;
    create dbAdmin transient;

    auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, FirstJournal, offset);

        write auditAccess.getJournalName;
        write auditAccess.getJournalNumber;
        write auditAccess.getJournalPath;

        auditAccess.getNextRecord(
                            entryType,
                            entryObjectType,
                            offset,
                            entryTimeStamp,
                            entrySerialNumber,
                            entryTransactionID,
                            entryOID,
                            entryClassNumber,
                            entryEdition
                            );

        write "Data for first entry:";
        write "Type: " & entryType.String;
        write "Object: " & entryObjectType.String;
        write "Offset: " & offset.String;
        write "Timestamp: " & entryTimeStamp.String;
        write "Serial Number: " & entrySerialNumber.String;
        write "Transaction ID: " & entryTransactionID.String;
        write "OID: " & entryOID;
        write "Class Number: " & entryClassNumber.String;
        write "Edition: " & entryEdition.String;

epilog
    delete auditAccess;
    delete dbAdmin;
```

3. Run the method. The Jade Interpreter Output Viewer should then look like the following.



Consulting the list of **JadeAuditAccess** class constants, we see that a Type of **49** is a **Database Open**. As such, there is no associated object.

Note For details, see the **JadeAuditAccess** class in the *Encyclopaedia of Classes (Volume 1)*, available at https://secure.jadeworld.com/developer-centre/Jade2025/OnlineDocumentation/.

Exercise 4 – Reading an Entire Journal

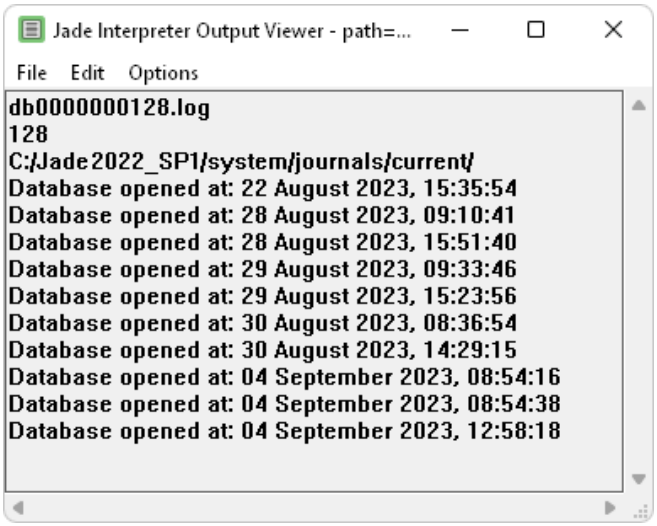
In this exercise, read an entire journal, displaying only the entries that relate to a database open or close action.

1. Modify the **openJournal** method as follows.

```
begin
    create auditAccess transient;
    create dbAdmin transient;

    auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, FirstJournal, offset);

    write auditAccess.getJournalName;
    write auditAccess.getJournalNumber;
    write auditAccess.getJournalPath;

    while auditAccess.getNextRecord(
                        entryType,
                        entryObjectType,
                        offset,
                        entryTimeStamp,
                        entrySerialNumber,
                        entryTransactionID,
                        entryOID,
                        entryClassNumber,
                        entryEdition
                        )do
        if entryType = auditAccess.Jaa_Type_DatabaseOpen then
            write "Database opened at: " & entryTimeStamp.String;
        elseif entryType = auditAccess.Jaa_Type_DatabaseClose then
            write "Database closed at: " & entryTimeStamp.String;
        endif;
    endwhile;
epilog
    delete auditAccess;
    delete dbAdmin;
end;
```

2. Run the method.

The Jade Interpreter Output Viewer should then look like the following.



3. Position the caret on the **getNextRecord** method and press the F1 shortcut key, to display the documentation about that method and use it to answer the following questions.

   a. What is the difference between the **getNextRecord** and **getNextRecordUTC** methods?

   b. Why does it work having the **getNextRecord** method as the condition for a while loop?

   Tip You may need to read the documentation for the **getNextRecordUTC** method, to obtain the answer.

Exercise 5 – Reading All Journals of a Database

In this exercise, extend the **openJournal** method to read all journals of the database rather than just the first one.

1. Modify the **openJournal** method as follows.

```
begin
    create auditAccess transient;
    create dbAdmin transient;

    auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, FirstJournal, offset);

    while true do
        write auditAccess.getJournalName;
        write auditAccess.getJournalNumber;
        write auditAccess.getJournalPath;

        while auditAccess.getNextRecord(
                            entryType,
                            entryObjectType,
                            offset,
                            entryTimeStamp,
                            entrySerialNumber,
                            entryTransactionID,
                            entryOID,
                            entryClassNumber,
                            entryEdition
                            )do
            if entryType = auditAccess.Jaa_Type_DatabaseOpen then
                write "Database opened at: " & entryTimeStamp.String;
            elseif entryType = auditAccess.Jaa_Type_DatabaseClose then
                write "Database closed at: " & entryTimeStamp.String;
            endif;
        endwhile;
        if not auditAccess.getNextJournal() = 0 then
            break;
        endif;
    endwhile;
epilog
    delete auditAccess;
    delete dbAdmin;
end;
```

2. Run the method.

   You will notice that it will successfully open each journal, one by one, and display any database open and close actions. However, when it reaches the end, an exception is raised.

3. To handle this exception, create a JadeScript method called **openJournalExceptionHandler** and code it as follows.

```
openJournalExceptionHandler(e: Exception; gotJournal: Boolean io): Integer;

begin
    if e.errorCode = JErr_DbFileNotFound then
        gotJournal:= false;
        return Ex_Resume_Next;
    else
        return Ex_Pass_Back;
    endif;
end;
```

4. Add a new **Boolean** called **gotJournal** to the **vars** list, as follows.

```
openJournal();

constants
    FirstJournal = 128;
vars
    auditAccess : JadeAuditAccess;
    dbAdmin     : JadeDatabaseAdmin;
    offset      : Integer;

    // output parameters for getNextRecord
    entryType           : Integer;
    entryObjectType     : Integer;
    entryTimeStamp      : TimeStamp;
    entrySerialNumber   : Decimal[12];
    entryTransactionID  : Decimal[12];
    entryOID            : String;
    entryClassNumber    : Integer;
    entryEdition        : Integer;

    gotJournal          : Boolean;
```

5. Initialize it to **true**, as follows.

```
begin
    gotJournal := true;

    create auditAccess transient;
    create dbAdmin transient;
```

6. Instead of using a *while true* loop, we can now use **while gotJournal**, relying on it being set to **false** by the exception handler when there are no more journals.

Modify **openJournal**, as follows.

```
begin
    gotJournal := true;

    create auditAccess transient;
    create dbAdmin transient;

    auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, FirstJournal, offset);

    on Exception do openJournalExceptionHandler(exception, gotJournal);
    while gotJournal do
        write auditAccess.getJournalName;
        write auditAccess.getJournalNumber;
        write auditAccess.getJournalPath;

        while auditAccess.getNextRecord(
                            entryType,
                            entryObjectType,
                            offset,
                            entryTimeStamp,
                            entrySerialNumber,
                            entryTransactionID,
                            entryOID,
                            entryClassNumber,
                            entryEdition
                            )do
            if entryType = auditAccess.Jaa_Type_DatabaseOpen then
                write "Database opened at: " & entryTimeStamp.String;
            elseif entryType = auditAccess.Jaa_Type_DatabaseClose then
                write "Database closed at: " & entryTimeStamp.String;
            endif;
        endwhile;
        auditAccess.getNextJournal();
    endwhile;
epilog
    delete auditAccess;
    delete dbAdmin;
end;
```

7. Run the method. It should no longer raise an exception when finishing.

Bonus Exercise – Finding the First Journal

In the previous exercises, we hard-coded the number of the first journal.

In this bonus exercise, attempt to find the first journal in a general way. This exercise is optional and as such, less instruction will be given than in normal exercises.

1. Create a JadeScript method called **findFirstJournal** and code it as follows (for now).

```
findFirstJournal() : Integer;

vars

begin
    return 128;
end;
```

2. Modify the **openJournal** method as follows.

```
begin
    gotJournal := true;

    create auditAccess transient;
    create dbAdmin transient;

    //auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, FirstJournal, offset);
    auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, findFirstJournal, offset);
```

This image is the first portion only of the method.

3. Your challenge is now to find an implementation of the **findFirstJournal** method that finds and returns the number of the first journal (which can vary from system to system).

Tips

You could:

- Try calling the **JadeAuditAccess** class **getJournal** method in a loop, trying journal numbers and handling the exceptions on misses
- Use the **FileFolder** class **files** method to look at the files in the default journal directory

Journal Filtering

Journal filtering is a powerful tool that extracts only the relevant data out of a set of journals.

The **JadeAuditAccess** class provides for several methods that enable you to set up filters that will then skip non-matching results when calling the **getNextRecord** method. These methods include the following.

- **registerFilterClass**, which takes a class number as a parameter and causes the **getNextRecord** method to return only events relating to instances of the specified class number.
- **registerFilterClassName**, which takes a schema name and a class name as parameters and causes the **getNextRecord** method to return only events relating to instances of the specified class within the specified schema.

  As this method relies on the journal's description file, it must be called after the **getJournal** method is successfully called.
- **registerFilterCollection**, which takes the number of a collection class and the number of a parent class.

If the parent class number is set to zero (**0**), any event relating to the specified collection class is returned by the **getNextRecord** method; otherwise, only events relating to the specified collection of the specified parent class are returned.

- **registerFilterCollectionName**, which takes three parameters: a schema name, a class name, and a reference property name.

  It causes only those events that relate to the collection specified in the reference property of the specified class of the specified schema to be returned by the **getNextRecord** method.

  As this method relies on the journal's description file, it must be called after the **getJournal** method is successfully called.

- **registerFilterTimeRange**, which takes two timestamp parameters: one for a start time and one for an end time. Only the events between the specified start time and end time are returned by the **getNextRecord** method.

  If the start time or end time parameter is not required, a **null** value accepts all events before or after the time, respectively.

  If the start time is to be used (that is, it is set to a non-null value), the **registerFilterTimeRange** method must be called before the **getJournal** method.

- **registerFilterTimeRangeUTC**, which behaves the same as the **registerFilterTimeRange** method except that the timestamps are in Coordinated Universal Time (UTC).

Any of these methods other than the time range methods can be reversed by calling the **JadeAuditAccess** class **setFilterExcludes** method.

The **setFilterExcludes** method takes a **Boolean** value as a parameter, and if passed true, changes the behavior of the filter to cause the **getNextRecord** method to return everything except the filtered events rather than only the filtered events. This can be reversed back to the normal behavior by calling it again, passing it a **false** value.

Note When filtering by class or collection, a filter must be set for both collection and class.

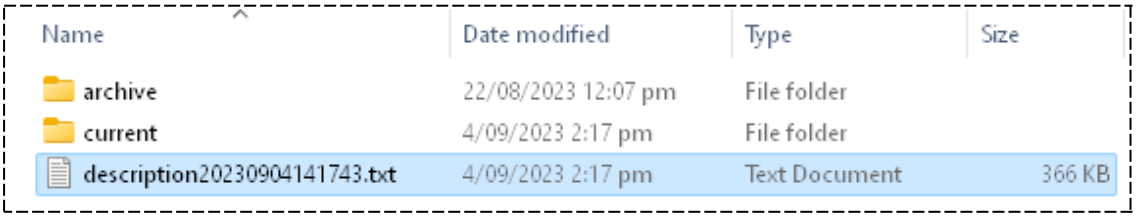
To filter out all non-collection classes or all collections, simply use a **RootSchema** class such as the **Schema** class (that is, class number **1**).

Exercise 6 – Filtering by Class

In this exercise, create a filter to show all events relating to the **Customer** class of **BankingModelSchema** that have been logged.

1. Navigate to your journal directory to find the timestamp of your most-recent description file (in this example, **20230904141743**). This number is based on the time and date when the description was generated.

   The following image shows one generated on September 4th, 2023 at 14:17:43).



2. In **AuditSchema**, create a JadeScript method called **filterByCustomer** and code it as follows, replacing **<timestamp>** in the **loadDescriptionByName** call with the located timestamp.

```
filterByCustomer();

constants
    FirstJournal = 128;

vars
    auditAccess : JadeAuditAccess;
    dbAdmin     : JadeDatabaseAdmin;
    offset      : Integer;
    gotJournal  : Boolean;
    // output parameters for getNextRecord
    // output parameters for getNextRecord
    entryType          : Integer;
    entryObjectType    : Integer;
    entryTimeStamp     : TimeStamp;
    entrySerialNumber  : Decimal[12];
    entryTransactionID : Decimal[12];
    entryOID           : String;
    entryClassNumber   : Integer;
    entryEdition       : Integer;
begin
    gotJournal := true;

    create auditAccess transient;
    create dbAdmin transient;

    auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, FirstJournal, offset);
    auditAccess.loadDescriptionByName("description<timestamp>.txt");
    auditAccess.registerFilterClassName("BankingModelSchema", "Customer");
    auditAccess.registerFilterCollection(1, 1);

    on Exception do openJournalExceptionHandler(exception, gotJournal);
    while gotJournal do
        write auditAccess.getJournalName;
        while auditAccess.getNextRecord(
                            entryType,
                            entryObjectType,
                            offset,
                            entryTimeStamp,
                            entrySerialNumber,
                            entryTransactionID,
                            entryOID,
                            entryClassNumber,
                            entryEdition
                            )do
            if not auditAccess.getClassName(entryClassNumber) = "" then
                write auditAccess.getClassName(entryClassNumber) & " was modified at " & entryTimeStamp.String;
            endif;
        endwhile;
        auditAccess.getNextJournal();
    endwhile;
epilog
    delete auditAccess;
    delete dbAdmin;
end;
```

3. Run the method.

   You will see that only the events relating to the **Customer** class of **BankingModelSchema** are displayed.

4. Modify the method to register a collection rather than a class, as shown in the following code fragment.

```
    auditAccess.getJournal(dbAdmin.getCurrentJournalDirectory, FirstJournal, offset);
    auditAccess.loadDescriptionByName("description<timestamp>.txt");
//  auditAccess.registerFilterClassName("BankingModelSchema", "Customer");
//  auditAccess.registerFilterCollection(1, 1);
    auditAccess.registerFilterClass(1);
    auditAccess.registerFilterCollectionName("BankingModelSchema", "Bank", "allBankAccounts");
```

5. Run the method.

You will see that only the events relating to the **allBankAccounts** collection of the **Bank** class are now displayed.

Automated Test Code Generator

The Automatic Test Code Generator (ATCG) is a Jade schema that enables you to record and replay GUI actions in Jade applications. It does this by capturing the execution of GUI event methods and generating code to replay those actions.

As the test code generated by ATCG is Jade code, you can modify it to meet your requirements.

Loading ATCG

ATCG is available on the **JadeSoftwareNZ** GitHub and you can load it directly from the Jade Platform development environment using the Jade Git integration.

Git works by *cloning*, to create a local copy of a remote repository and then providing tools to manage synchronization and versioning between your local copy and the remote repository. For this module, we use only the Git clone function to obtain a copy of the ATCG system from the **JadeSoftwareNZ** GitHub repository onto your local machine.

When the schema (.**scm**) and form and data definition (.**ddx**) files have been cloned, you can load them into your Jade database using a standard schema load.

Note DDX files, implemented in Jade 2018.0.01, serve the same purpose as the legacy DDB files except that they are human-readable and are therefore more appropriate for the Git technology, which relies on the ability to compare the differences in files.

Method Recording

ATCG generates its test code by tracking method calls (including click events on buttons, and so on) as a user performs GUI actions.

It then generates code fragments (snippets) that call those methods and bundles them into a class (.**cls**) file that is loaded into the database.

When the ATCG test is run, it simply calls the methods that were recorded into its .**cls** file, which has the same effect as performing the GUI actions manually. However, as it does not actually perform any click or mouse actions, the location of elements changing would not affect the test but the naming of the elements changing would.

Exercise 1 – Loading ATCG

In this exercise, use Jade's Git integration to load the ATCG schema into your database.

Note The application we will be testing is the Banking application developed during the Jade Platform Developer's course. If you do not already have a copy, a model answer can be found on the **JadeSoftwareNZ** GitHub repository.

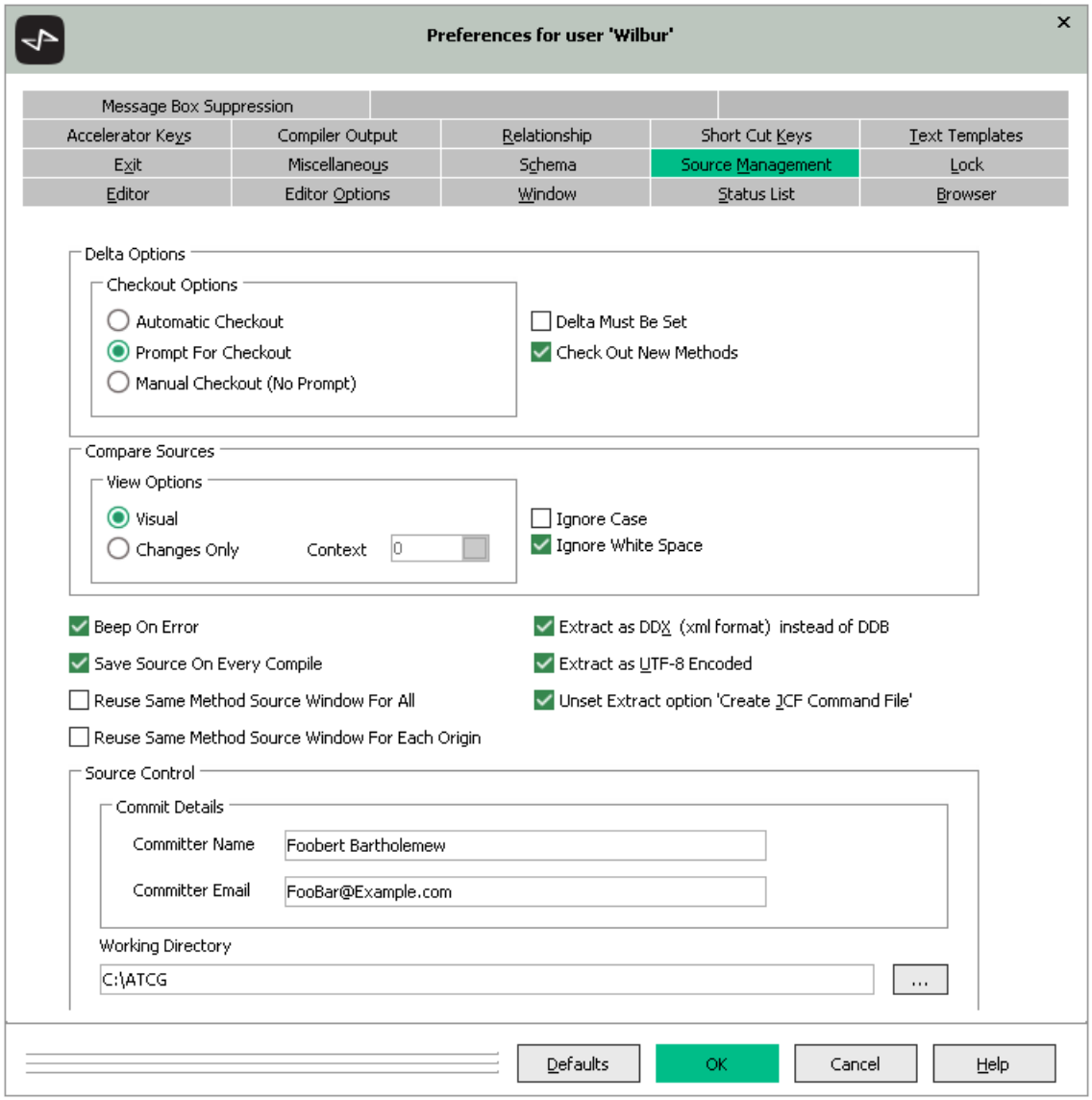
---

1. Select the **Preferences** command from the Options menu and in the controls in the Source Control group box on the **Source Management** sheet, add:

   a. Your name

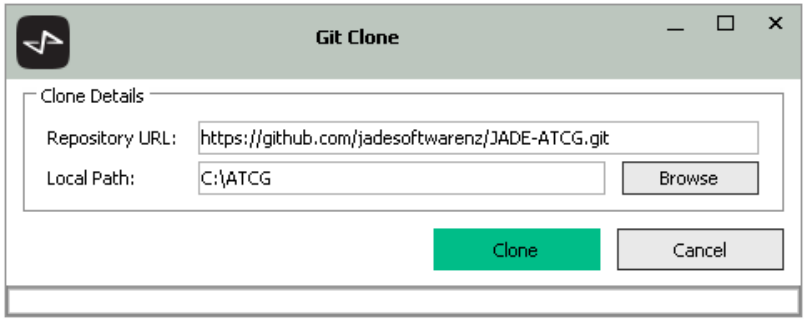
   b. Your email address

   c. A valid directory



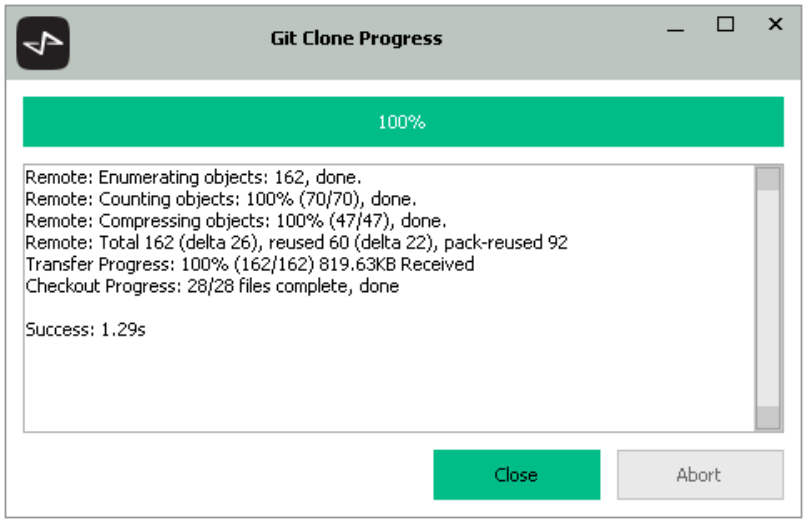
Note The specified working directory will become the folder used for keeping a local copy of the ATCG schemas. This folder should be empty, but it can be called whatever you like, as long as the location is memorable.

2. Select the **Clone** command from the Git Source Control Client submenu of the Browse menu, filling out the details as follows.



Note Replace **C:\ATCG** with the working directory you specified in step 1 of this instruction, if required.

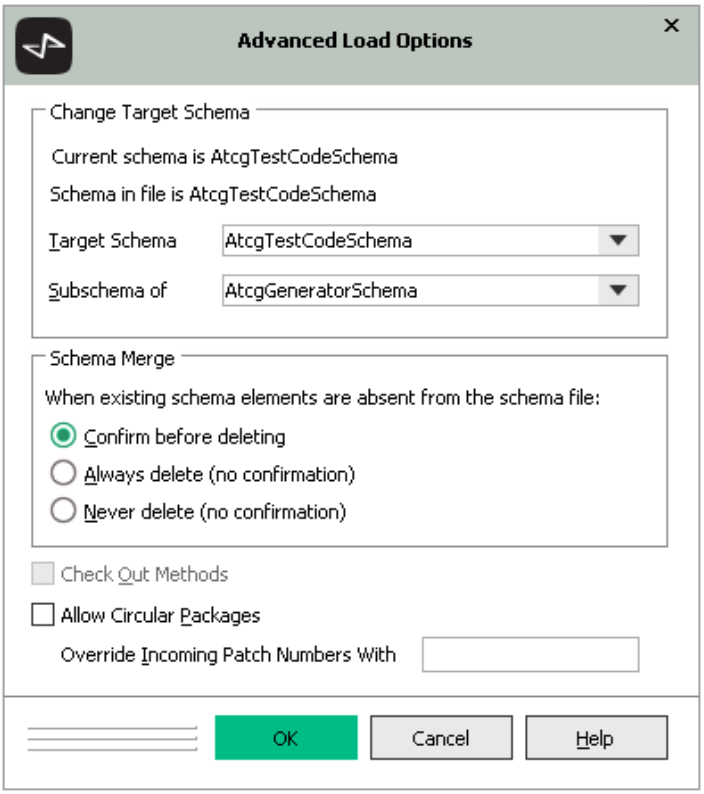
3. The Git Clone Progress dialog displays the clone progress as the files are copied to your working directory, and when complete, enables the **Close** button.



Note Load the **AtcgGeneratorSchema** first (as a subschema of **BankingViewSchema**), and then load the **AtcgTestCodeSchema** as a subschema of the **AtcgGeneratorSchema**.

4. Select the **Load** command from the Schema menu. Navigate to your specified working directory, open its **ATCG** subfolder, and then select **AtcgGeneratorSchema.scm**.
5. Click the **Advanced** button at the right of the **Load Style** combo box and then fill out the Advanced Load Options dialog, as follows.
   a. Enter **AtcgGeneratorSchema** as the name of the target schema.
   b. Select **BankingViewSchema** as the superschema.
6. Click **OK** on the Advanced Options dialog and then click **OK** on the Load Options dialog.
7. Select the **Load** command from the Schema menu. Navigate to your specified working directory, open its **ATCG** subfolder, and then select **AtcgTestCodeSchema.scm**.

8. Click the **Advanced** button at the right of the **Load Style** combo box and then fill out the Advanced Load Options dialog, as follows.

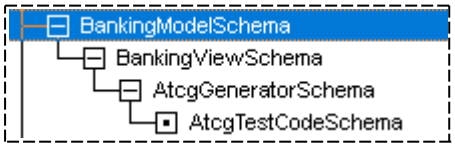


   a. Enter **AtcgTestCodeSchema** as the name of the target schema.

   b. Select **AtcgGeneratorSchema** as the superschema.

9. Click **OK** on the Advanced Options dialog and then click **OK** on the Load Options dialog.

Your schema should look like the following example.



Exercise 2 – Configuring ATCG for the Banking System

In this exercise, configure ATCG to track GUI events in the **Banking** system so that the GUI actions can be replayed.

1. By default, Jade blocks the tracking of user actions. To enable method tracking, add the **MethodTrackingEnabled=true** parameter to the [JadeSecurity] section of your Jade initialization (**jade.ini**) file.

```
[JadeSecurity]
MethodTrackingEnabled=true
```

2. In the **AtcgTestCodeSchema**, find the **XxxProfile** class, which is a subclass of **AtcgProfile**, and rename it to **BankingProfile**. This is the class from which all generated test classes are subclassed.

3. In the **AtcgTestCodeSchemaApp** subclass of **Application**, find and replace all instances of **XxxProfile** with **BankingProfile** (that is, within the **atcgGetControlOptions** and **atcgRecordAppInit** methods).

4. In the **atcgGetControlOptions** method, replace **XxxSchema** with **BankingViewSchema**.

```
// list of schemas to be tracked during recording
targetSchemas.add("AtcgTestCodeSchema");
targetSchemas.add("BankingViewSchema");
```

5. To make testing with ATCG easier, temporarily disable the **getAndValidateSecurity** method in the **GBankingViewSchema** subclass of **Global** in the **BankingViewSchema**, by adding **return true;** as the first line, as follows.

```
getAndValidateUser(usercode: String output; password: String output): Boolean;

vars
    form: Logon;
begin
    return true;
```

6. Navigate to the **BankingProfile** class under **AtcgProfile** in the **AtcgTestCodeSchema**, and then add a new reference property called **form** of type **MainMenu**.

7. Modify the **startup** method of the **BankingProfile** class, as follows.

```
startup() updating;
//  This method will start EVERY profile.
//  It creates the app's mainForm and logs in
vars
    mm : MainMenu;

begin
    app.atcgLogMessageTC("===========");
    app.atcgLogMessageTC("Starting profile");
    app.atcgLogMessageTC("===========");

    // create and show main form, and logon
    create self.form transient;
    self.form.show();

    app.atcgLogMessageTC(method.qualifiedName&" finished");
end;
```

8. Modify the **stop** method of the **BankingProfile** class, as follows.

```
stop() updating;

begin
    app.atcgLogMessageTC("============");
    app.atcgLogMessageTC("Stopping profile");
    app.atcgLogMessageTC("============");

    // log out, close all forms, close main form
    delete self.form;

    app.atcgLogMessageTC("Stopped");
end;
```

9. Run the **AtcgControlApp** application in **AtcgTestCodeSchema**, to ensure it opens correctly.

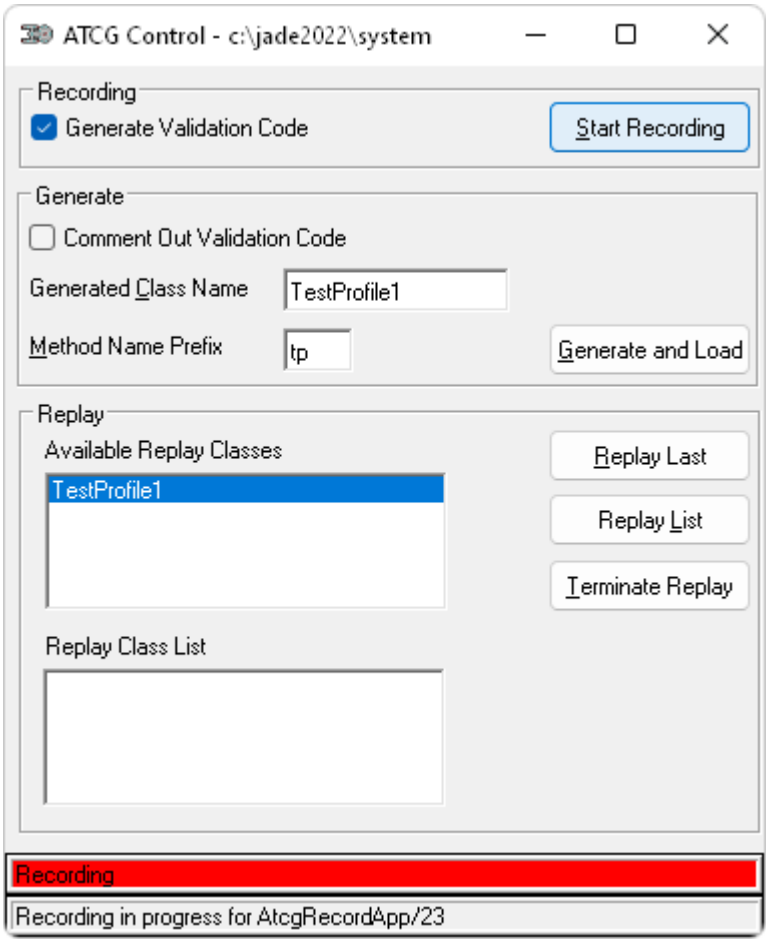
Exercise 3 – Recording a Test Run

In this exercise, use ATCG to record the adding of a customer with the GUI form, creating a test case with the recording, and then replaying it multiple times.

Note Although this is not an efficient way to add many customers, it a good demonstration of ATCG's ability to automate GUI processes.

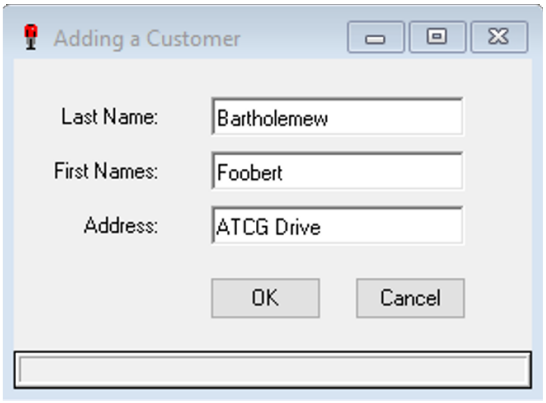
1. Before you record the adding of customers, run the **JadeScript** class **removeTestData** method in the **BankingModelSchema**.
2. Ensure your database is running as a standard (fat) client, as ATCG will not function correctly as a presentation (thin) client or in single user mode.
3. Start the ATCG Control application by performing the following actions.
   a. Select the **AtcgTestCodSchema**.
   b. Click the **Run Application** toolbar button.
   c. Select **AtcgControlApp** in the **Application Name** combo box and then click the **OK** button.

4. Click **Start Recording** on the ATCG Control dialog.



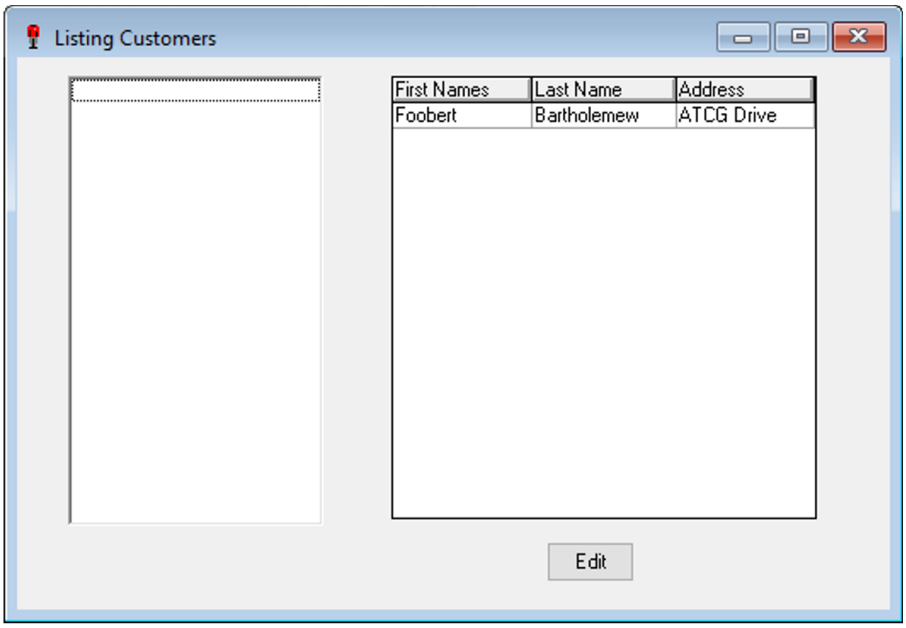
5. The **Banking** system then opens. Perform the following actions.

   a. Select the **Add** command from the Customer menu, and then fill out the dialog as follows.



   b. Click **OK**.

c. Select the **List** command from the Customer menu. You should see one copy of **Foobert Bartholemew** in the list of customers at the right of the dialog.

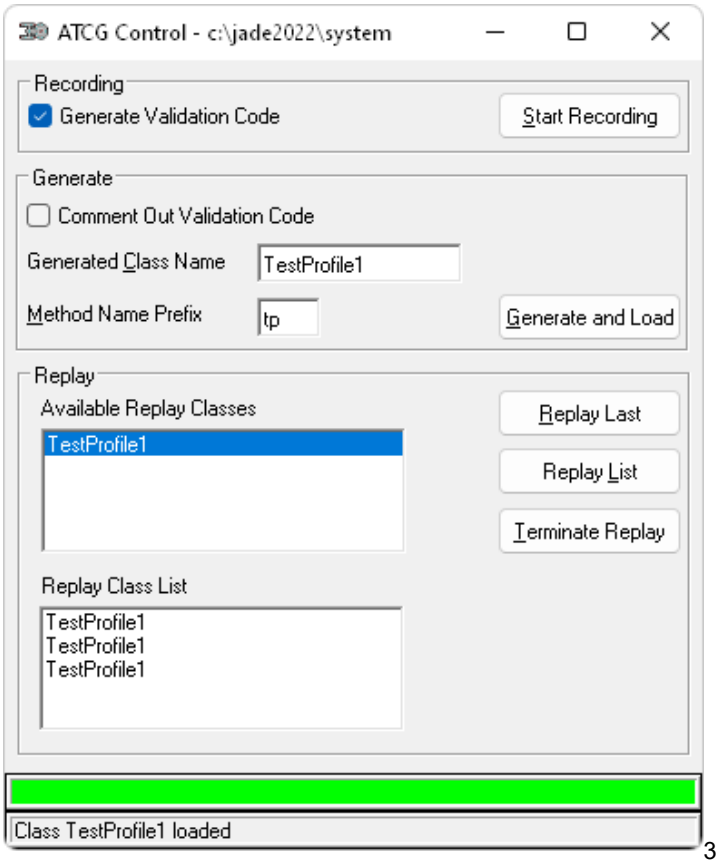


d. Close the **Banking** system form.

6. Click **Generate and Load** on the ATCG Control dialog.

7. When the generation is complete, double-click **TestProfile1** in the **Available Replay Classes** list box to add it to the **Replay Class List** list box.

8. Do this three times, so that there are three copies of **TestProfile1** in the **Replay Class List** list box, as follows.



3

9. Click **Replay List**.

ATCG will repeat the same actions you manually performed three times. You should see four copies of **Foobert Bartholemew** in the list of customers: one from the manual run and three from the automated runs.

Generated Code

Whenever a test profile is generated in ATCG, a class with the name of the replay class is added as a subclass of the **AtcgProfile** subclass (which defaults to **XxxProfile** and is renamed **BankingProfile** for the **Banking** system).

This class has a reference for each form that was used during the test run and a method for each set of actions. In addition, it always has the following three methods.

- **runTest**, which runs each of the methods of the class in sequence
- **startup**, which ensures the preconditions are met
- **tp999_tidyUp**, which ensures the postconditions are met

Although you will not normally want to change these three methods, it can often be useful to manually change the other generated methods to make small changes to the test's operation without the need of a full re-recording.

The manual changing of methods can also be useful to eliminate superfluous actions from the test code. For example, when filling out the Customer Add dialog in the **Banking** system, you will likely see the generated code look like the following.

```
tp004_CA_btnOK_click():Integer updating;

vars
    keyCode:Integer;
begin
    myCustomerAdd_1:=app.getForm('CustomerAdd').CustomerAdd;
    if unexpected(2, 'myCustomerAdd_1', '', 'FormNotNull', null, myCustomerAdd_1, null, method.qualifiedName) then return 2; endif;
    keyCode:=16;
    myCustomerAdd_1.txtLastName.keyDown(myCustomerAdd_1.txtLastName, keyCode, 1);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtLastName.firstChange(myCustomerAdd_1.txtLastName);
    keyCode:=16;
    myCustomerAdd_1.txtLastName.keyUp(myCustomerAdd_1.txtLastName, keyCode, 0);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtLastName.text:='Bartholemew';
    myCustomerAdd_1.txtLastName.lostFocus(myCustomerAdd_1.txtLastName);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtLastName.change(myCustomerAdd_1.txtLastName);
    keyCode:=16;
    myCustomerAdd_1.txtFirstNames.keyDown(myCustomerAdd_1.txtFirstNames, keyCode, 1);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtFirstNames.firstChange(myCustomerAdd_1.txtFirstNames);
    keyCode:=16;
    myCustomerAdd_1.txtFirstNames.keyUp(myCustomerAdd_1.txtFirstNames, keyCode, 0);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtFirstNames.text:='Foobert';
    myCustomerAdd_1.txtFirstNames.lostFocus(myCustomerAdd_1.txtFirstNames);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtFirstNames.change(myCustomerAdd_1.txtFirstNames);
    keyCode:=16;
    myCustomerAdd_1.txtAddress.keyDown(myCustomerAdd_1.txtAddress, keyCode, 1);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtAddress.firstChange(myCustomerAdd_1.txtAddress);
    keyCode:=16;
    myCustomerAdd_1.txtAddress.keyUp(myCustomerAdd_1.txtAddress, keyCode, 0);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.btnOK.mouseDown(myCustomerAdd_1.btnOK, 1, 0, 23, 9);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtAddress.text:='ATCG Drive';
    myCustomerAdd_1.txtAddress.lostFocus(myCustomerAdd_1.txtAddress);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtAddress.change(myCustomerAdd_1.txtAddress);
    myCustomerAdd_1.btnOK.mouseUp(myCustomerAdd_1.btnOK, 1, 0, 23, 9);
    app.doWindowEvents(shortPause);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.btnOK.click(myCustomerAdd_1.btnOK);  // 16:02:47
    return 0;
end;
```

Most of this code is used to reproduce the subtleties of how the form was navigated; for example, tabbing between fields and the handling of the toggling of the Shift key as text was entered. None of these are required for the ATCG to reproduce the actions, so the method could be stripped to the following.

```
tp004_CA_btnOK_click():Integer updating;

vars
    keyCode:Integer;

begin
    myCustomerAdd_1 := app.getForm('CustomerAdd').CustomerAdd;

    myCustomerAdd_1.txtLastName.text    := 'Bartholemew';
    myCustomerAdd_1.txtFirstNames.text  := 'Foobert';
    myCustomerAdd_1.txtAddress.text     := 'ATCG Drive';
    myCustomerAdd_1.btnOK.click(myCustomerAdd_1.btnOK);

    return 0;
end;
```

Note This method also has the **app.doWindowsEvents** calls (pauses) removed so that it replays faster than a human can see.

ATCG and Persistent Data

A special consideration when using ATCG to automate the testing of Jade GUI applications is that any changes to the database while performing an ATCG test can change the behavior of the GUI application and therefore cause future ATCG runs to fail. For example, if using ATCG to automate the testing of the adding of a customer and the customer name has a uniqueness constraint, the second run of the test will fail.

There are several strategies that can be useful to mitigate this risk and ensure the smooth running of many iterations of ATCG testing.

- Start from a fresh database each run; that is, clear all data from the database before or after each run.
- Perform only actions in an ATCG run that do not impact the database.
- Reverse any persistent data changes within the same ATCG test run; for example, if you add a customer, delete it afterwards.

The nature of the database and the goals of the testing determine which of these strategies is most appropriate.

Recording Strategies

As it can often be time-consuming recording ATCG test runs, and the manual running of applications is prone to human error, you should limit the scope of each test run to as small as possible.

This has several benefits, as follows.

- It is easier to re-record if a major mistake is made
- When changing the code manually, it is easier to find the specific place in the test code at which a minor mistake was made
- If the behavior of the application being tested changes, it is easier to isolate the change into a single small test, rather than having to re-record all the functionality

The code generated when recording ATCG test runs does not attempt to discriminate between what is relevant and what is irrelevant (for example, resizing or moving forms or unnecessary clicks) to the test you are trying to run.

It is therefore preferable to perform only actions that are strictly necessary and to avoid superfluous actions that generate extra code in the test (making it harder to manage and slower to rerun during replay).

Exercise 4 – Editing Generated Code

In this exercise, modify the code generated in the previous exercise to remove superfluous actions and change the name of the customer to be added.

1. In the **tp004_CA_btnOK_click** method of the **TestProfile1** class (a subclass of the **BankingProfile** class), remove code until the following remains.

```
tp004_CA_btnOK_click():Integer updating;

vars
    keyCode:Integer;
begin
    myCustomerAdd_1:=app.getForm('CustomerAdd').CustomerAdd;
    if unexpected(2, 'myCustomerAdd_1', '', 'FormNotNull', null, myCustomerAdd_1, null, method.qualifiedName) then return 2; endif;

    myCustomerAdd_1.txtLastName.text:='Bartholemew';
    app.doWindowEvents(shortPause);

    myCustomerAdd_1.txtFirstNames.text:='Foobert';
    app.doWindowEvents(shortPause);

    myCustomerAdd_1.txtAddress.text:='ATCG Drive';
    app.doWindowEvents(shortPause);

    myCustomerAdd_1.btnOK.click(myCustomerAdd_1.btnOK);  // 16:02:47
    return 0;
end;
```

Note As you don't need to add any new code yet, just remove the extra lines of code such as those for **keyDown** events.

2. Modify the string values being assigned to the text boxes, as follows.

```
tp004_CA_btnOK_click():Integer updating;

vars
    keyCode:Integer;
begin
    myCustomerAdd_1:=app.getForm('CustomerAdd').CustomerAdd;
    if unexpected(2, 'myCustomerAdd_1', '', 'FormNotNull', null, myCustomerAdd_1, null, method.qualifiedName) then return 2; endif;

    // myCustomerAdd_1.txtLastName.text:='Bartholemew';
    myCustomerAdd_1.txtLastName.text:='Foolish';
    app.doWindowEvents(shortPause);

    // myCustomerAdd_1.txtFirstNames.text:='Foobert';
    myCustomerAdd_1.txtFirstNames.text:='Barry';
    app.doWindowEvents(shortPause);

    // myCustomerAdd_1.txtAddress.text:='ATCG Drive';
    myCustomerAdd_1.txtAddress.text:='Manual Entry Place';
    app.doWindowEvents(shortPause);

    myCustomerAdd_1.btnOK.click(myCustomerAdd_1.btnOK);  // 16:02:47
    return 0;
end;
```

3. Save the method, then re-run the **Replay Class List** from the ATCG Control dialog. The automated runs will now create **Barry Foolish** rather than **Foobert Bartholemew**.

Limitations

While ATCG is a powerful tool, there are some limitations that need to be accounted for when using it to create automated GUI tests.

- Exception handling - ATCG does not support the replaying of **Ex_Abort_Action**. Any exception handler that uses this will cause the ATCG recorder to terminate recording at this point.
- Although the **AtcgRecordApp** application must be run as a standard (fat) client, you can run the **AtcgReplayApp** application from a fat client or a thin client.
- You can run one **AtcgRecordApp** application only at a time. However, you can run any number of other applications simultaneously.
- ATCG cannot correctly validate dates or times, as they will vary from recording to replaying. It automatically attempts to detect the presence of a time or date on a form and give a warning, rather than failing with an error. For more details, see "Validation Warnings for Date and Time Fields" in the *Automated Test Code Generator (ATCG) Reference*.
- While calling an event method is usually the same as performing the corresponding GUI action, this can sometimes not be the case. For example, clicking on a table performs the following as well as generating the **click** event.
  - Sets focus to the table
  - Sets the value of the **selected** property
  - Changes the values of the **row** and **column** properties
- The **Form** class **showModal** method blocks execution until the form is unloaded. This prevents ATCG from replaying actions onto that form and it therefore generates a **handleShowModal** method instead (which uses the **show** method rather than the **showModal** method).

Exercise 5 – Troubleshooting Exceptions

This exercise assumes that the state of exception handling in the **Banking** system is as it was at the end of the Jade Platform Developer's course; that is:

- The **initialize** method arms a global exception handler.

```
initialize() updating;

begin
    on Exception do self.genericExceptionHandler(exception) global;
    self.myBank := Bank.firstInstance();
    if self.myBank = null then
        beginTransaction;
        create myBank persistent;
        commitTransaction;
    endif;
end;
```

- The global exception handler returns **Ex_Abort_Action**.

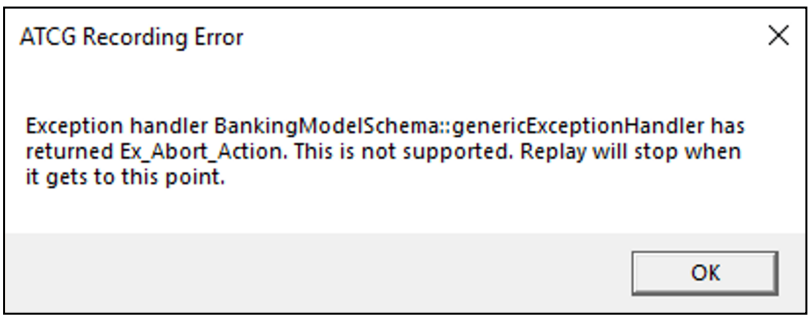
```
genericExceptionHandler(exObj: Exception): Integer;

begin
    abortTransaction;
    exObj.logSelf("errors.log");
    app.msgBox("Unexpected error occurred", "Application Error", MsgBox_OK_Only);
    return Ex_Abort_Action;
end;
```

- The **click** event of the **btnCancel** button on the Add Customer form raises an exception.

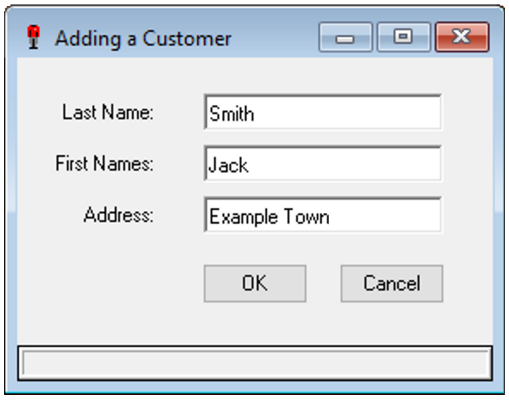
```
btnCancel_click(btn: Button input) updating;

begin
    write 42/0;
    self.unloadForm();
end;
```

In this exercise, you will troubleshoot the handling of exceptions while recording an ATCG test profile. (Note that the end of lines are clipped in some images in this exercise because of long line lengths.)

1. Open the ATCG Control dialog, select **TestProfile2** as the generated class name, and then click **Start Recording**.
2. Open the Customer Add dialog, by selecting the **Add** command from the Customer menu.
3. Click **Cancel**. A message box is displayed, stating that an unexpected error occurred. Click **OK**, and the following should be displayed, because **Ex_Abort_Action** is not supported by ATCG.



4. Close the ATCG Recording Error message box and then add a customer, as follows.



5. Close the **Banking** system form and then click **Generate and Load** on the ATCG Control dialog.

6. Run the newly created test profile by clicking **Replay Last**. You will see that the test aborts and fails as soon as it gets to the place where the exception was raised.

Tip To fix this, we can either circumnavigate the code paths that generate such an exception while recording or add a local exception handler to the **Test Profile** method that calls the offending code after generation.

7. Open **AtcgTestCodeSchema** in the Class Browser and navigate to **TestProfile3** under **BankingProfile**.

8. Add a method to this class called **suppressException**, coded as follows.

```
suppressException(e : Exception) : Integer;

begin
    write "Exception suppressed: " & e.text;
    return Ex_Resume_Next;
end;
```

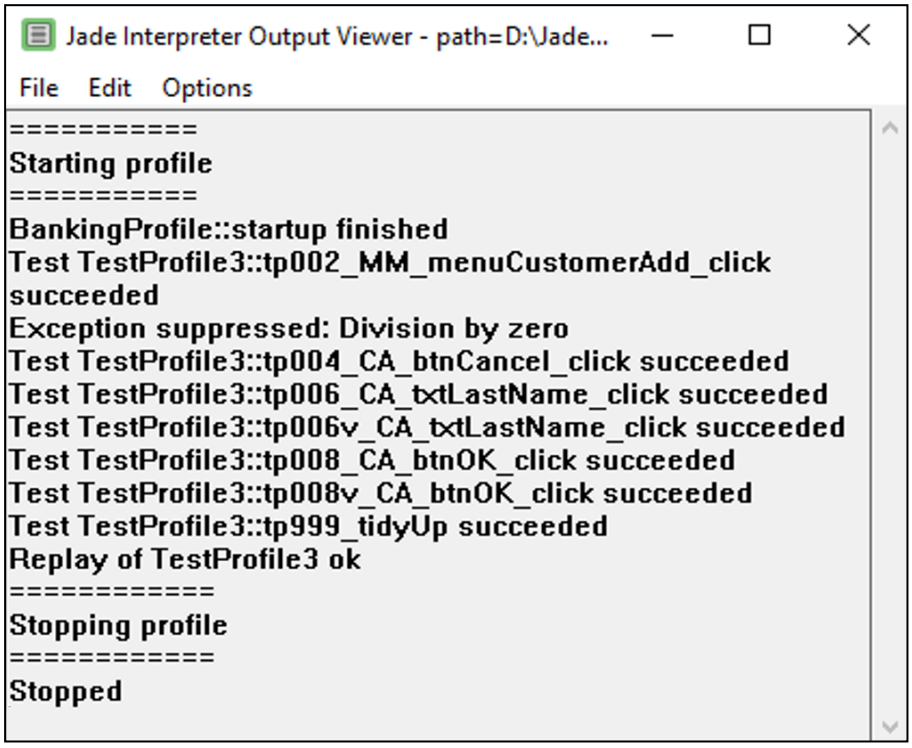
9. Add the following to the **tp004_CA_btnCancel_click** method.

```
tp004_CA_btnCancel_click():Integer updating;

begin
    on Exception do suppressException(exception);

    myCustomerAdd_1:=app.getForm('CustomerAdd').CustomerAdd;
    if unexpected(2, 'myCustomerAdd_1', '', 'FormNotNull', null, myCustomerAdd_1,
    myCustomerAdd_1.btnCancel.mouseDown(myCustomerAdd_1.btnCancel, 1, 0, 24, 7);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtLastName.text:='';
    myCustomerAdd_1.txtLastName.lostFocus(myCustomerAdd_1.txtLastName);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.txtLastName.change(myCustomerAdd_1.txtLastName);
    myCustomerAdd_1.btnCancel.mouseUp(myCustomerAdd_1.btnCancel, 1, 0, 24, 7);
    app.doWindowEvents(shortPause);
    app.doWindowEvents(shortPause);
    myCustomerAdd_1.btnCancel.click(myCustomerAdd_1.btnCancel);  // 13:01:36
    return 0;
end;
```

10. In the **runTest** method, comment out the following.

```
runTest():Integer updating;

vars
    sa:StringArray;
begin
    create sa transient;
    sa.add('tp002_MM_menuCustomerAdd_click');
    sa.add('tp004_CA_btnCancel_click');
    sa.add('tp006_CA_txtLastName_click');
    sa.add('tp006v_CA_txtLastName_click');
    sa.add('tp008_CA_btnOK_click');
    sa.add('tp008v_CA_btnOK_click');
    sa.add('tp999_tidyUp');
//  app.atcgSetMsgBoxReturn('Unexpected error occurred', 'Application Error', 0, 1);
//  app.atcgSetMsgBoxReturn('Exception handler BankingModelSchema::genericExceptionHand

    runMethods(sa, 6, "v_");

    app.atcgDetectMissedMsgBoxes;
    return retCode;
epilog
    delete sa;
end;
```

11. Run the **TestProfile3** again. It will now replay the entire test profile successfully.

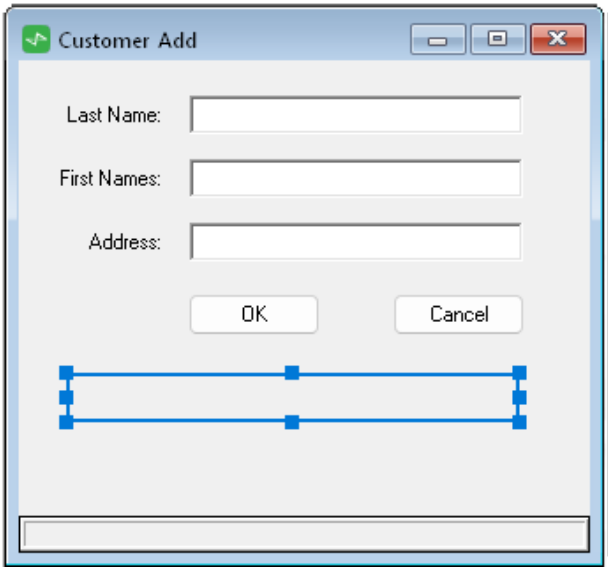


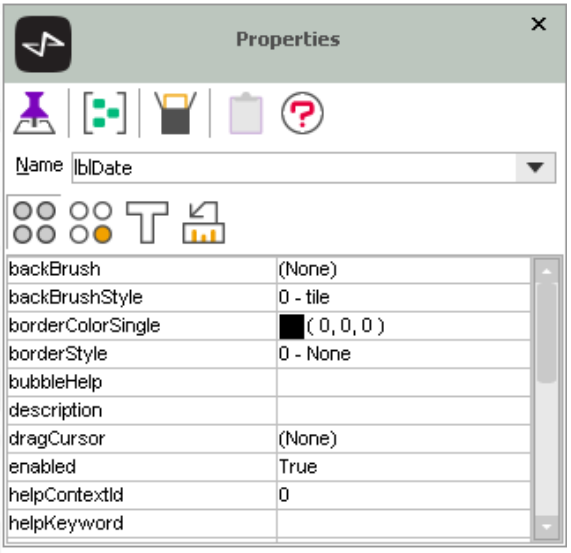
Exercise 6 – Troubleshooting Dates and Times

In this exercise, troubleshoot the handling of dates and times in ATCG, which may not remain static over multiple runs of the application.

1. Select **BankingViewSchema** in the Schema Browser and then open the Jade Painter (for example, by using the Ctrl+P shortcut keys).

2. Edit the CustomerAdd form by adding a **Label** control called **lblDate** at the bottom of the form.





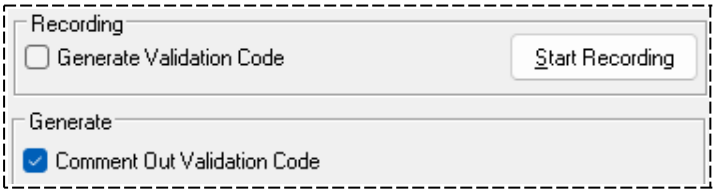
3. Open the **BankingViewSchema** in the Class Browser and modify the **load** method on the **Form Events** sheet of the **CustomerAdd** Form class, as follows.

```
load() updating;

begin
    lblDate.caption := app.actualTime.String;
    txtLastName.maxLength := 0;
end;
```

4. Run the **ATCG Control** from **AtcgTestCodeSchema** and click the **Start Recording** button to begin a new recording.
5. Open the Add Customer dialog and click on any field. (As soon as you click on an element in a form, ATCG validates it. Forms that you don't touch are not validated.)
6. Close the **Banking** system.
7. Change the generate class name to **TestProfile3**.
8. Click **Generate and Load**, to create a new test profile.
9. Click **Replay Last**, to replay the most-recently created test profile. You will notice that the replay fails when it validates and the time part of the **lblDate** doesn't match.

Tip We don't want to validate the time displayed on a form, so will suppress the part of the validation that is checking that element.

10. The simplest, but also least-precise way of suppressing the validation, is to uncheck **Generate Validation Code** to not validate at recording time or to check **Comment Out Validation Code**, to disable all validation at generation time.



You can still go through the code and uncomment the validation, if required.

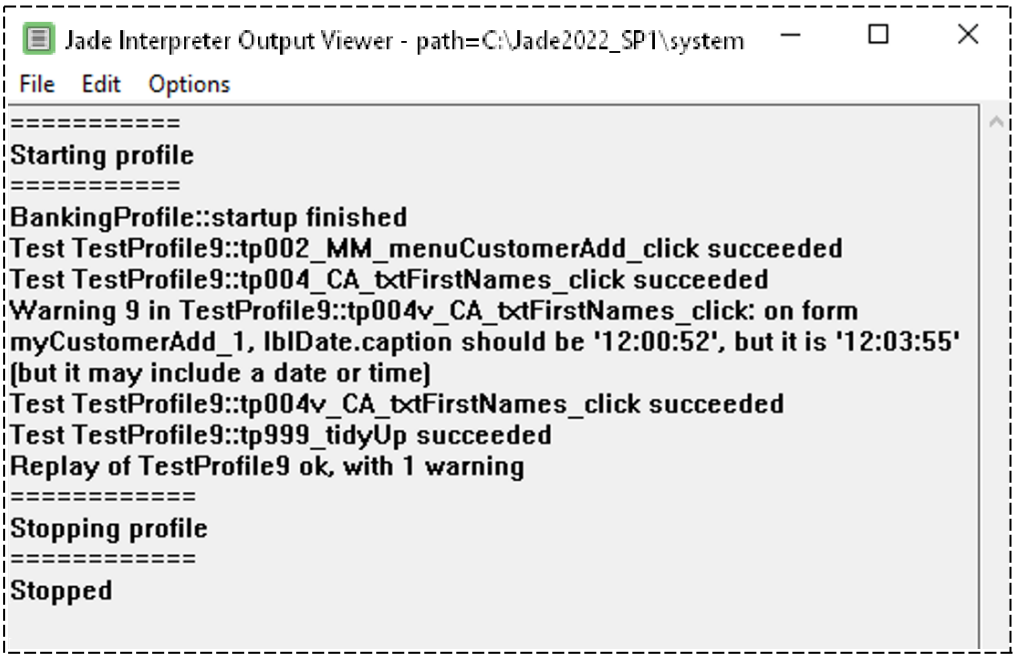
11. The more-precise way of suppressing the validation is to manually find and comment out or delete the offending code.

```
tp004v_CA_txtFirstNames_click():Integer updating;

begin
    if not app.isValidObject(myCustomerAdd_1) then
        return 0;  // presumably the form has been closed by the previous clic
    endif;
    if unexpected(3, 'myCustomerAdd_1', 'txtAddress', 'text', '', myCustomerAd
    if unexpected(4, 'myCustomerAdd_1', 'label3', 'caption', 'Address:', myCus
    if unexpected(5, 'myCustomerAdd_1', 'txtFirstNames', 'text', '', myCustome
    if unexpected(6, 'myCustomerAdd_1', 'label2', 'caption', 'First Names:', m
    if unexpected(7, 'myCustomerAdd_1', 'txtLastName', 'text', '', myCustomerA
    if unexpected(8, 'myCustomerAd_1', 'label1', 'caption', 'Last Name:', myC
//  if unexpected(9, 'myCustomerAdd_1', 'lblDate', 'caption', '11:13:41', myCu
    return 0;
end;
```

12. To enable ATCG to try to auto-detect dates and times, add the following section and parameters to your **jade.ini** file.

```
[ATCG]
ValNoWarningsOverride=false
ValDateTimeWarningOnly=true
```

Running a test profile with a time in it now gives a warning rather than an error and it does not fail the whole test.



Note ATCG's definition for containing a date or time is any string that contains a number followed by a slash or a colon, followed by a second number. This can give both false positives and false negatives, so use them with caution.

Development Environment

The Jade Platform development environment (or IDE) is full of features that are useful for navigating, maintaining, and bug-fixing in large Jade systems.

In this module, you will use the various features of the development environment to locate and repair defects in an intentionally broken version of the **Erewhon** demonstration system. The **Erewhon** system is a demonstration system that showcases some of the features of the Jade Platform, and as such, is large enough for the development environment tools to be useful, while small enough to be reasonable to navigate without requiring a great deal of familiarity with the system.

User Preferences

The Preferences dialog is accessed from the **Preferences** command from the Options menu. It allows you to specify a variety of look-and-feel options for the Jade Platform development environment.

Options set in the Preferences dialog persist between sessions. However, they impact the user who set them; any other users of the same database are unaffected.

The Preferences dialog has the following tabs that access sheets containing the types of options you can set.

Sheet	Purpose
Accelerator Keys	Maps accelerator keys that insert text into the Jade editor; for example, Ctrl+Shift+A inserts **abortTransaction;**
Browser	Customizes behavior related to how hierarchy browsers (including the Schema Browser and Class Browser) function; for example, the Mdi group box specifies whether they are floating MDI windows or tabs are displayed above the MDI client window
Compiler Output	Modifies the behavior of the Compiler Output Viewer
Editor	Customizes syntax highlighting colors; for example, Jade keywords are a dark blue color by default, but you can change them to a color of your choice
Editor Options	Customizes the way in which the editor pane is displayed; for example, whether line numbers are displayed
Exit	Customizes the behavior when exiting from the development environment; for example, whether to confirm before closing
Lock	Modifies the default lock exception handling
Message Box Suppression	Allows for the suppression of specified message boxes and retention of your answers for your future work sessions
Miscellaneous	Modifies settings that you are unlikely to want to change; default file suffixes (although it is best not to change the schema file suffix of .**scm**), the base locale, and interface and versioning options
Relationship	Customizes the display of the relationships, which are covered later in this module
Schema	Changes the default access to classes, references, and attributes in schemas as well as the persistence and type of classes
Shortcut Keys	Changes development environment and keyboard shortcuts, and swaps to the default use of F11 and F12 keys and F5 and F9 keys; that is, Jade by default has the F11 and F12 keys and the F5 and F9 keys swapped relative to Visual Studio (or Visual Studio to Jade, depending on your perspective)
Source Management	Customizes Jade version control functionality; for example, deltas and Git integration
Status List	Allows for the display of all methods and class constants, instead of the default display of only methods and class constants that are in error or have not been compiled (covered later in this module)

Sheet	Purpose
Text Templates	Customizes the default text template; for example, so that you can add a header to every method for documentation purposes
Window	Customizes the colors of windows (for example, the background color of the Class Browser) and the default font used in Jade windows

Macros and Regular Expressions

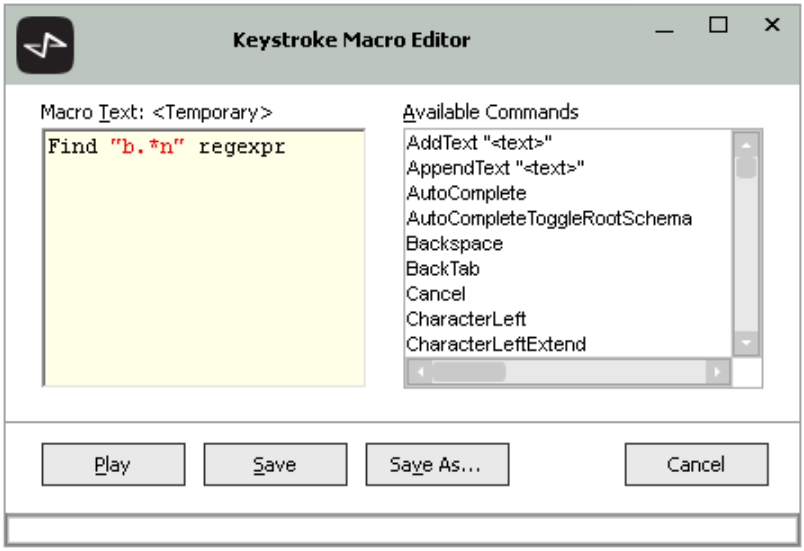
Jade macros enable you to record and replay keystrokes in the editor pane; for example, to comment or uncomment a selection of text, or to delete all text on the current line.

The two ways to create a macro are as follows.

- Use the **Start Macro Record** command from the Macro submenu of the Edit menu to start recording, perform a sequence of keystrokes, and then select the **Stop Macro Record** command from the Macro submenu of the Edit menu.
- Select the **Edit Temp Macro** command from the Macro submenu of the Edit menu to access the Keystroke Macro Editor dialog, select the required actions from the list box at the right (or you can type them in the **Macro Text** text box at the left), and then click **Save**.

When creating macros, you can use regular expressions.

The Keystroke Macro Editor supports the use of regular expressions (regex) with the **Find**, **ReplaceFind**, and **ReplaceAll** commands. For example, you can use the **Find** command as follows.



In this example, **regexpr** causes the search string to be treated as a regular expression rather than a string literal. As such, **"b.*n"** translates to a **b**, then any number of any characters, and then an **n**.

You can use the following regular expression characters.

Character	Description
.	Matches any character.
\(	Marks the start of a region for tagging a match.

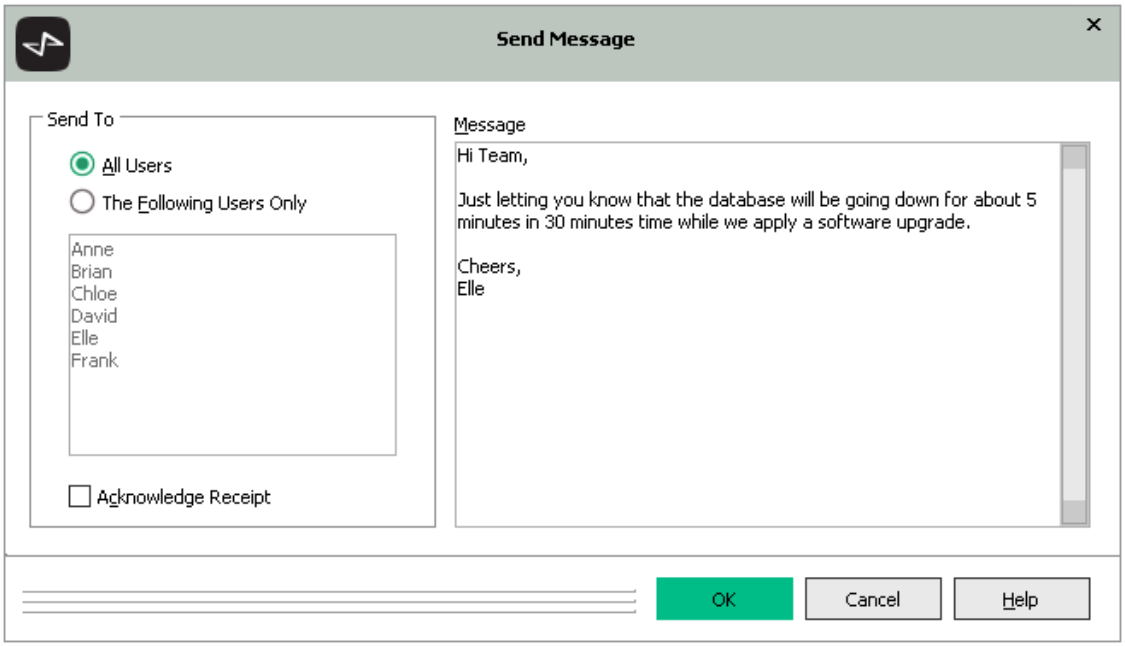
Character	Description
\)	Marks the end of a tagged region.
\n	The *n* parameter is a digit in the range 1 through 9, which refers to the first through ninth tagged region when replacing; for example, if the search string was **Fred\([1-9]\)XXX** and the replace string was **Sam\1YYY**, this would generate **Sam2YYY** when applied to **Fred2XXX**.
\<	Matches the start of a word. A word is defined to be a character string beginning or ending, or both beginning and ending, with characters in the ranges A through Z, a through z, 0 through 9, and an underscore. In addition, it must be preceded or followed, or both preceded and followed, by any character outside those mentioned.
\>	Matches the end of a word.
\x	Enables you to use a character x that would otherwise have a special meaning; for example, **\[** would be interpreted as **[** and not as the start of a character set.
[...]	Indicates a set of characters; for example, **[abc]** means any of the characters a, b, or c. You can also use ranges; for example, **[a-z]** for any lowercase character.
[^...]	Complement of the characters in the set; for example, **[^A-Za-z]** means any character except an alphabetic character.
^	Matches the start of a line (unless it is used inside a set).
$	Matches the end of a line.
*	Matches zero (**0**) or more times; for example, **Sa*m** matches **Sm**, **Sam**, **Saam**, **Saaam**, and so on.
+	Matches one or more times; for example, **Sa+m** matches **Sam**, **Saam**, **Saaam**, and so on.

When you have saved a macro, you can replay it by selecting the **Play Temp Macro** command from the Macro submenu of the Edit menu. You can also make a temporary macro (whether generated from recording keystrokes or from the Keystroke Macro Editor dialog) permanent, by clicking **Save As** on the Keystroke Macro Editor dialog.

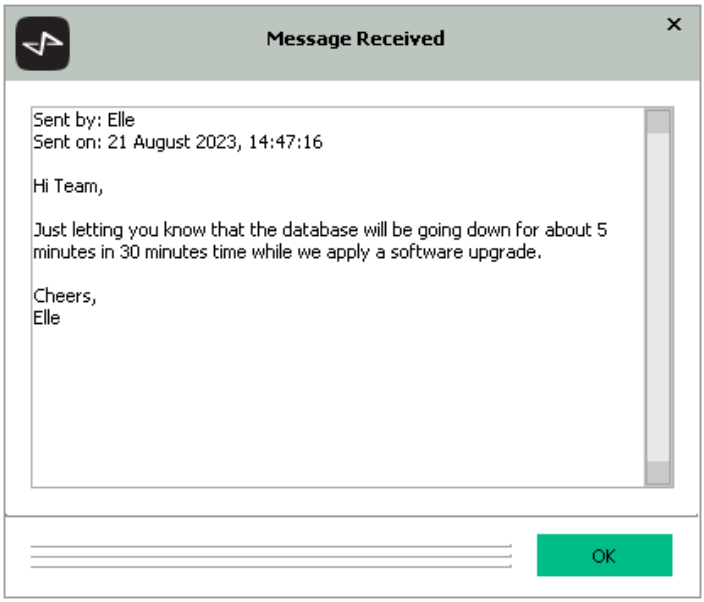
Sending Messages to other Developers

When using the Jade Platform development environment in a multiuser system, it can sometimes be useful to see who else is connected to the Jade server and to send them messages; for example, to let all currently connected developers know when the database is going to be taken down or to check with a developer before terminating their process with the Jade Monitor.

The **Send Message** command of the File menu accesses the Send Message dialog, which displays a list of all users who are currently connected to the database.



You can send messages to all users by selecting the **All Users** option button, or to a specific user or users by selecting the **The Following Users Only** option button and then selecting the users to which the message should be sent.



The sender's name and the timestamp automatically generated for all messages are displayed above the message.

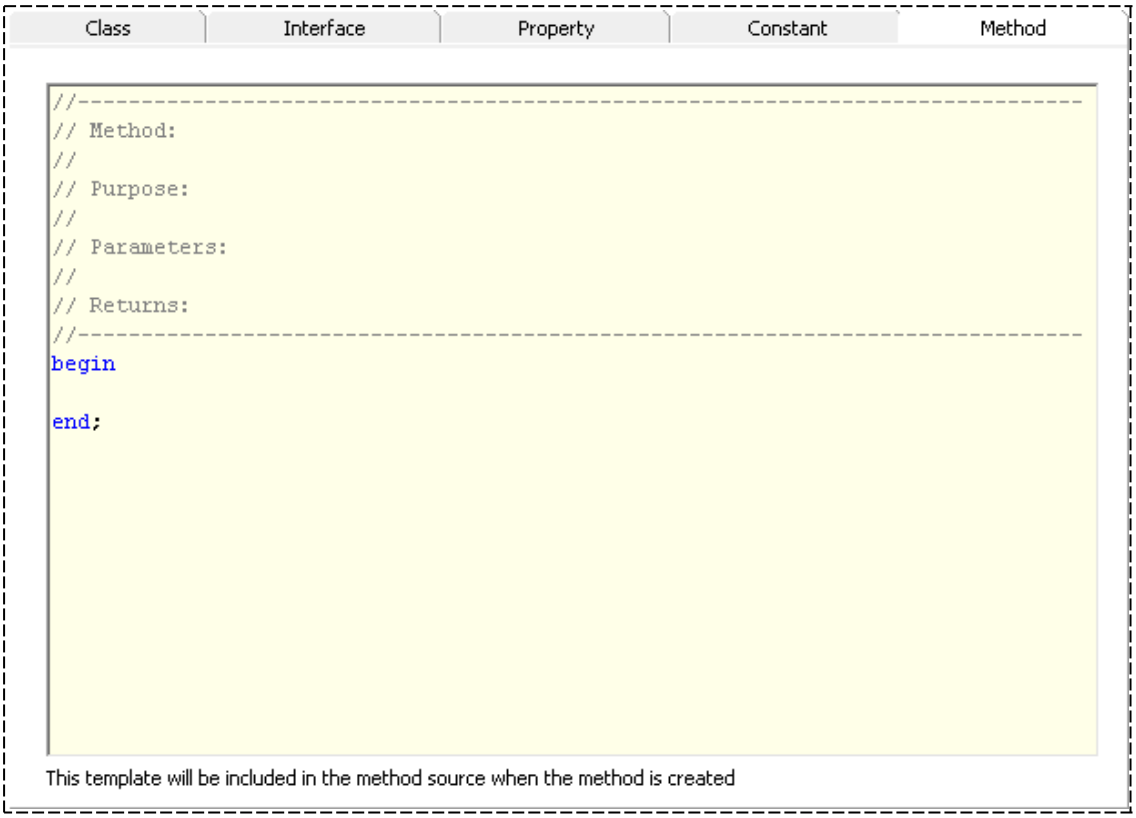
When the Message Received dialog is closed, if you checked the **Acknowledge Receipt** check box on the Send Message dialog, an acknowledgement message is sent back to you.

Exercise 1 – Customizing your Development Environment View

In this exercise, configure the look and feel of your development environment view. Although the steps in this exercise suggest a configuration setting, you can select whatever values you prefer.

1. From the Options menu, select the **Preferences** command.
2. Click on the **Accelerator Keys** tab to display the **Accelerator Keys** sheet and then change the **E** accelerator key from **endforeach;** to **epilog**. This will cause pressing Ctrl+Shift+E to insert **epilog** into the editor, at the caret.
3. On the **Browser** sheet, check the **Show Inherited** check box and then select the **Use Tabs Only** option button in the Mdi group box.
4. On the **Editor** sheet, change the **Background** from light yellow to white.
5. On the **Editor Options** sheet, check the **Insert Parentheses for Method with no parameters** check box in the Auto Complete group box and the **View Line Numbers** check box in the Display Options group box.
6. On the **Exit** sheet, uncheck the **Exit Confirmation** check box and check the **Save Windows** check box.
7. On the **Lock** sheet, change the **Number of times to Retry** to **15**.
8. On the **Miscellaneous** sheet, change the alternative Jade Skin in the Versioning Options group box to **JADE Sumner**.
9. On the **Relationship** sheet, uncheck the **Show Detail** check box and set the **Target Class** color to white.
10. On the **Source Management** sheet, check the:
    - **Extract as DDX (xml format) instead of DDB** check box.
    - **Reuse Same Method Source Window For All** check box.
11. On the **Short Cut Keys** sheet, select the **MacroPlayTemp** shortcut, click in the **Key Combination** text box, and then press Shift+Space to add that as the shortcut key combination.
12. On the **Status List** sheet, check the **Compiled Methods** check box and uncheck the **Uncompiled Methods** check box.

13. On the **Text Templates** sheet, add the following to the **Method** folder.

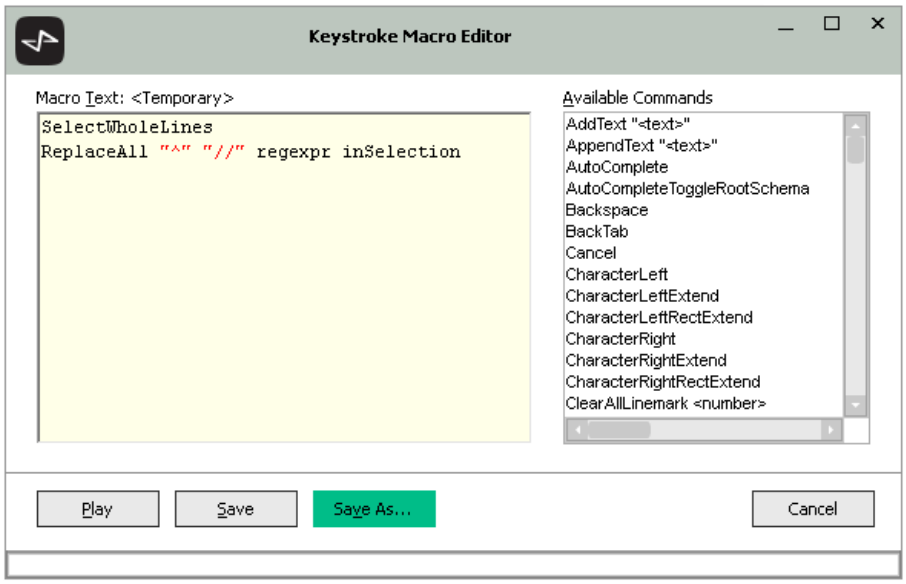


14. On the **Window** sheet, check the **Show Alternating Row BackColor** check box and then select **JADE New Brighton** from the **Select Jade Skin** combo box.

Exercise 2 – Creating a Jade Macro

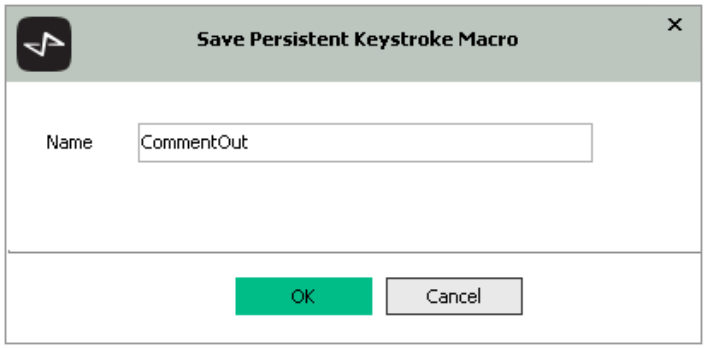
In this exercise, create macros to comment on or remove a comment from the selected text by generating or removing // at the beginning of each selected line.

1. From the Macro submenu of the Edit menu, select the **Edit Temp Macro** command.
2. Code the macro as follows, and then click **Save As**.

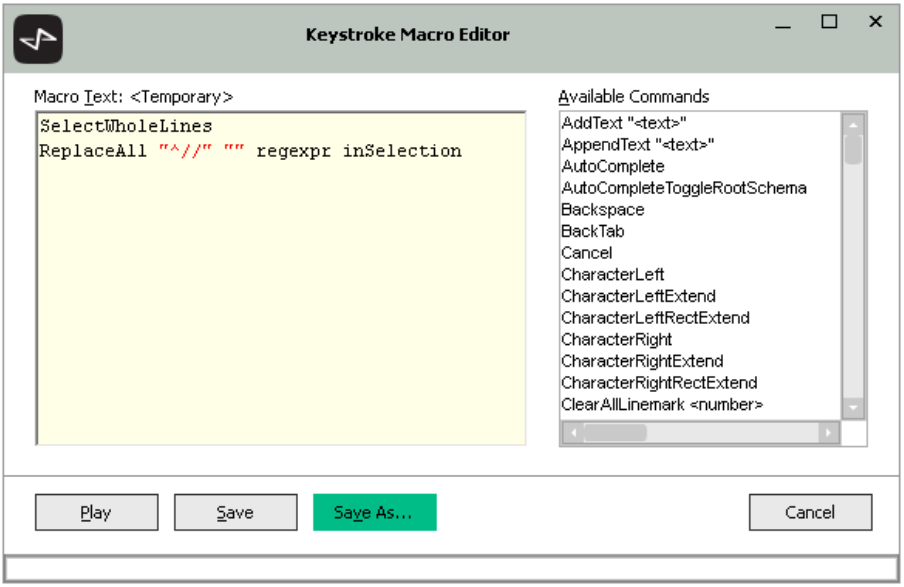


This macro finds the beginning of each selected line with a regular expression and then adds // to the start of each line.

Enter **CommentOut** as the name of the macro and then click **OK**.



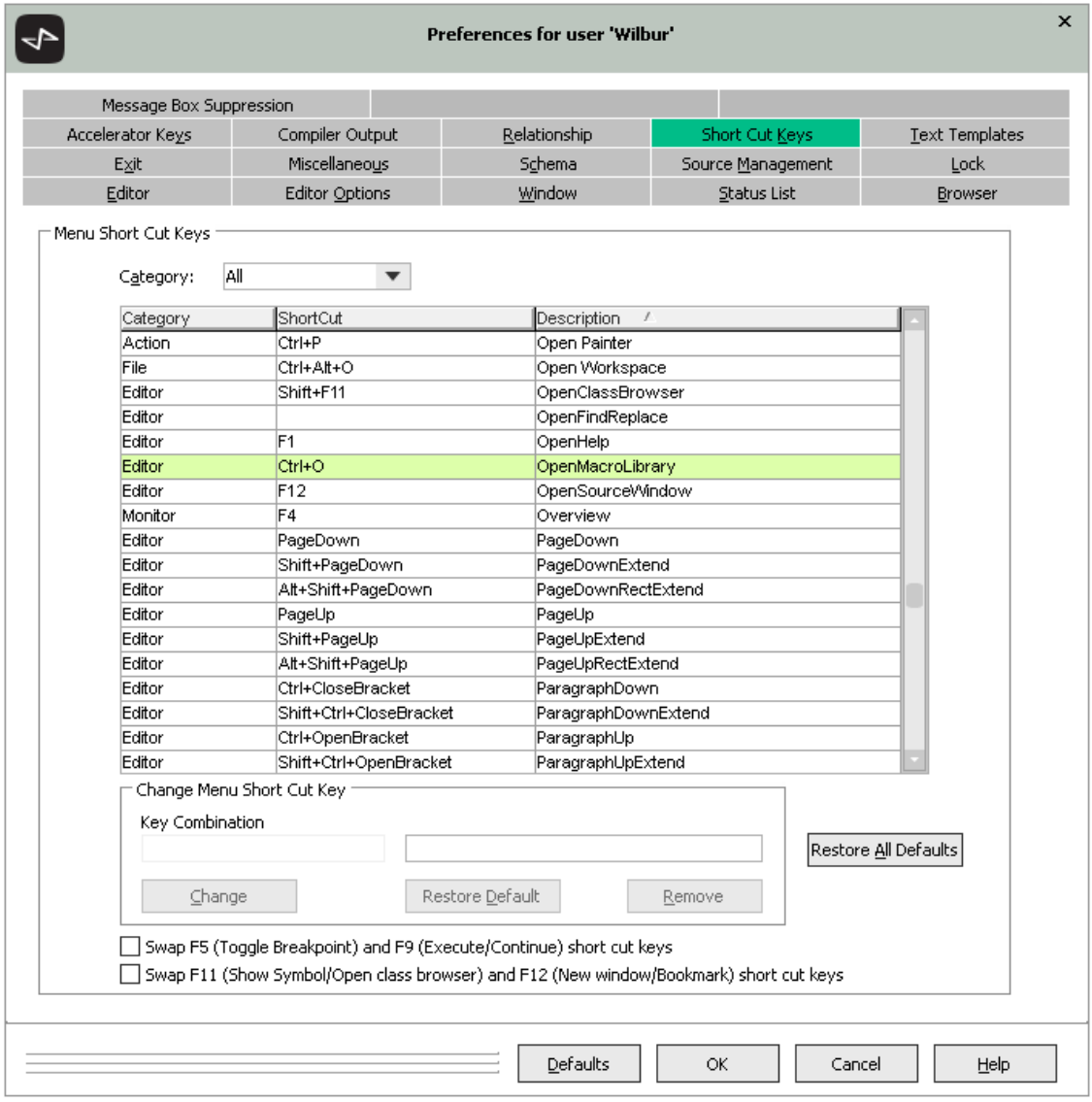
3. Reopen the Keystroke Macro Editor dialog and code a new macro as follows. Save it as **Uncomment**.



This macro finds the beginning of each selected line that begins with **//** and replaces the **//** with an empty string, which deletes the **//** from the line.

4. Open the Preferences dialog (from the **Preferences** command from the Options menu).

5. Add a Ctrl+O shortcut to the **OpenMacroLibrary** command.



6. Create a new JadeScript method called **m1** (in any schema, or create a new schema named **S1**, if needed) and then select the **begin** and **end;** instructions, as follows.

```
1   m1();
2   //---------------------------------------------------------------------
3   // Method:
4   //
5   // Purpose:
6   //
7   // Parameters:
8   //
9   // Returns:
10  //---------------------------------------------------------------------
11  begin
12
13  end;
```

7. Press Ctrl+O to open the Keystroke Macro Editor dialog, select the **CommentOut** command in the list box at the right of the dialog, and then click **Play**.

   The **begin** and **end;** instructions should be commented out, along with the blank line between them.

```
ml();
//------------------------------------------------------------------------------
// Method:
//
// Purpose:
//
// Parameters:
//
// Returns:
//------------------------------------------------------------------------------
//begin
//
//end;
```

8. With the **begin** and **end**; instructions still highlighted, press Ctrl+O to reopen the Keystroke Macro Editor dialog and then select the **Uncomment** command in the macro list at the right. When you click **Play**, the highlighted text should no longer be commented out.

```
 1  ml();
 2  //------------------------------------------------------------------------------
 3  // Method:
 4  //
 5  // Purpose:
 6  //
 7  // Parameters:
 8  //
 9  // Returns:
10  //------------------------------------------------------------------------------
11  begin
12
13  end;
```

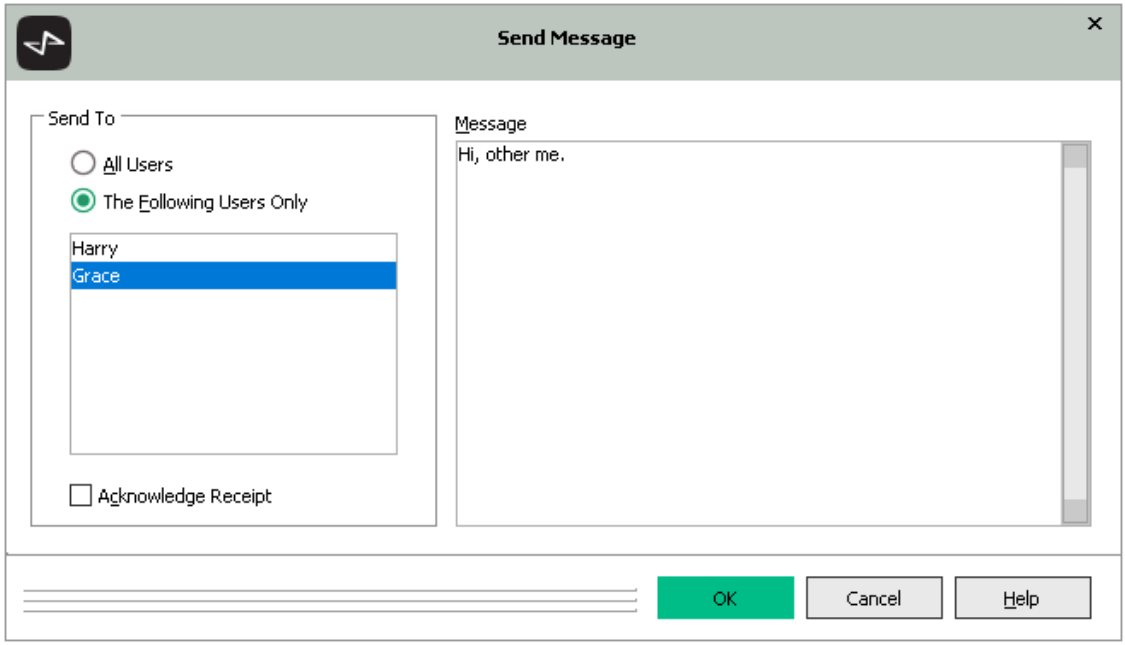
9. Delete the **m1** method (and the **S1** schema, if you created it).

Exercise 3 – Sending a Message in a Multiuser System

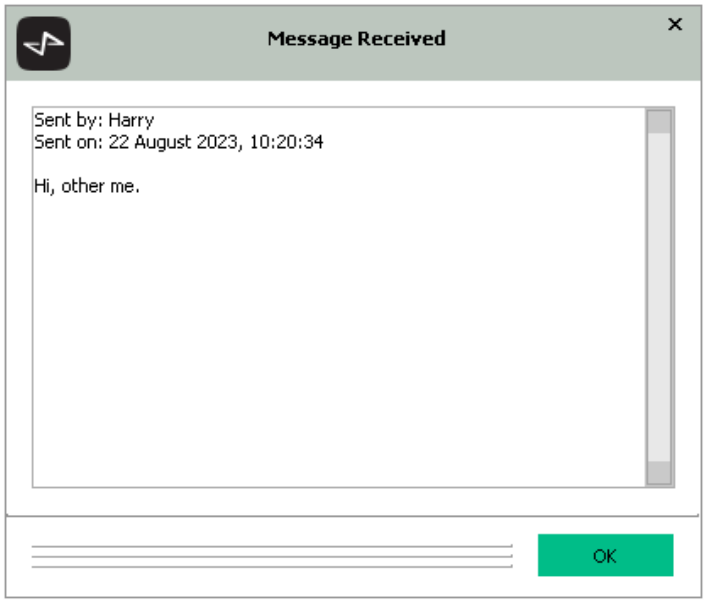
In this exercise, open Jade in multiuser mode and then send a message from one client to another.

1. If your Jade system is open, close it.
2. Start the Jade database server for your database.
3. Start two Jade fat (standard) client nodes for your database, signing on to each with a different name.
4. From one of the client nodes, select the **Send Message** command from the File menu.
5. Select the **The Following Users Only** option button and the user name that you selected for the other client node.

6. Enter a message in the **Message** text box and then click **OK**.



7. The message should appear on the other client node.



Show Symbol Command and Source Windows

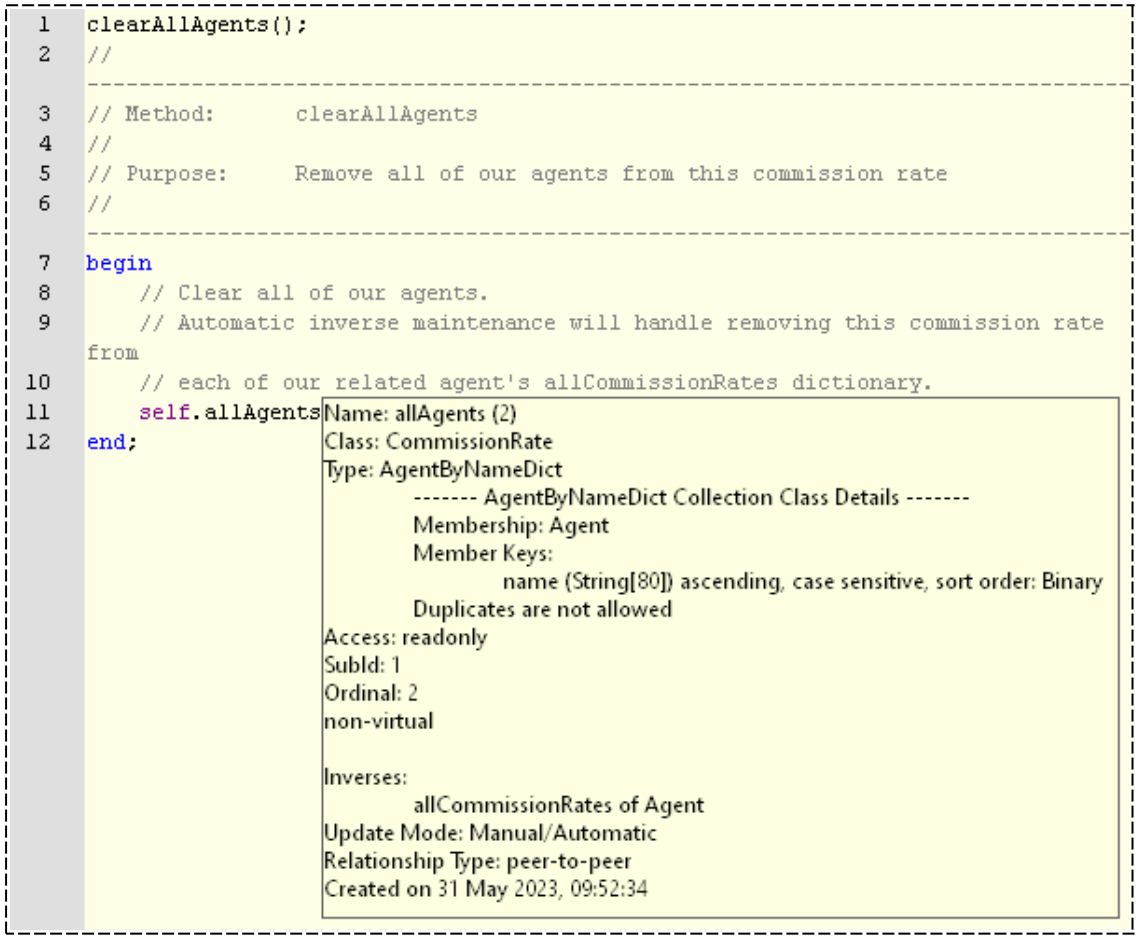
When navigating around a larger system, it is often important to be able to see the implementation of an identifier when encountering it in the code base. An identifier can be a method name, a local variable, a property, a class, a constant, or an interface.

It is considered best practice to use descriptive names for these identifiers. However, when maintaining another person's code, you may need to peek at what an implementor describes.

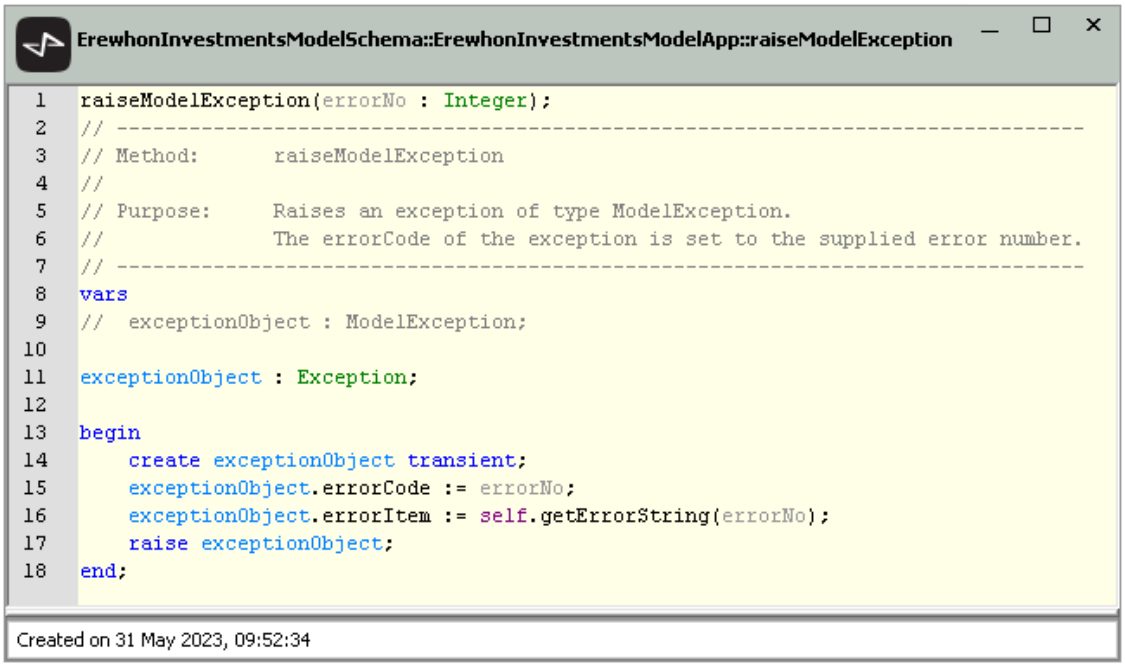
The **Show Symbol** command, accessed by the F11 key, shows the implementation of the identifier under the caret. It will behave slightly differently, depending on the identifier that is under the caret and the values of the **Reuse Same Method Source Window** check boxes on the **Source Management** sheet of the Preferences dialog.

The behavior of the F11 key is as follows.

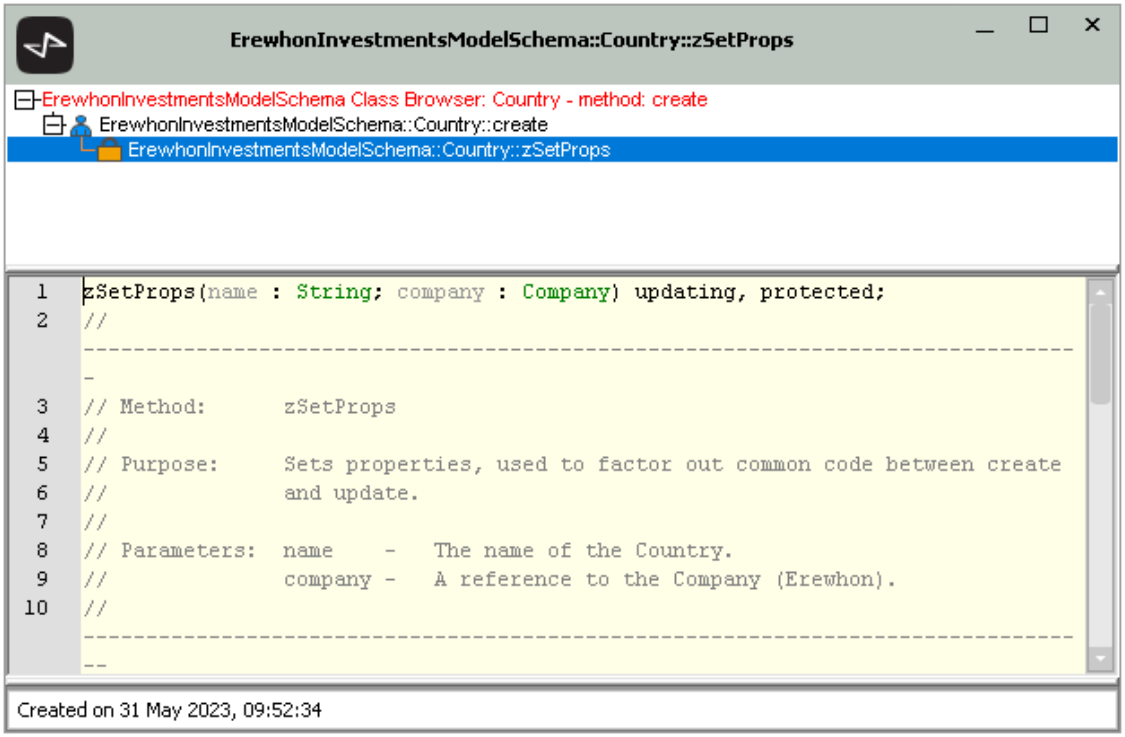
- For variables, properties, classes, constants, and interfaces, it creates a dialog with a description of the identifier; for example, the following will be displayed for a property.



- For methods when the **Reuse Same Method Source Window** check boxes on the **Source Management** sheet are unchecked, the method is opened in a new window.



- For methods when the **Reuse Same Method Source Window For All** check box on the **Source Management** sheet is checked, the method is added to a common method source form, shared amongst all opened methods.



- If the **Reuse Same Method Source Window For Each Origin** check box is enabled on the **Source Management** sheet, the method is added to a method source window that is common to methods opened from the same Class Browser.

References and Implementors

The Jade Platform development environment enables you to list all:

- References (the methods that call a specific method or that read or update a specific property) for a method or property
- Implementors (other methods with the same name) for a method

Find references of a property by right-clicking a property or from the Properties menu while the property is selected, and then selecting any of the following.

- References

  Shows all references to the property, and whether they read the property or update it

- Read References

  Shows only the references to the property that read the property

- Update References

  Shows only the references to the property that update the property

To find the references to a method, you can search for all references or for local references only.

To search for all references across all classes both within the current schema and any subschemas, right-click on the method and then select **References** or select the **References** command from the Methods menu when the method is selected.

Alternatively, you can search for the local references only; that is, references to the method that are within the method's class or one of its subclasses, and in the current schema only. To search for local references only, right-click on the method and select **Local References** or select the **Local References** command from the Methods menu when the method is selected.

Implementors of a method are most commonly used for finding a method's reimplementations. As such, all reimplementations of the method within the class hierarchy, but including subschemas, can be found using the **Local Implementors** command. Access this command by right-clicking on a method and then selecting **Local Implementors** or by selecting the **Local Implementors** command from the Methods menu when the method is selected.

Alternatively, you can locate all methods within the schema with the same name as a method; for example, to see which classes have a **toString** method implemented. The **Implementors** command finds all methods that share a name with a method across all classes of the schema, but it does not search subschemas. To search for implementors of a selected method, right-click on the method and then select **Implementors** or select the **Implementors** command from the Methods menu when the method is selected.

Renaming Identifiers

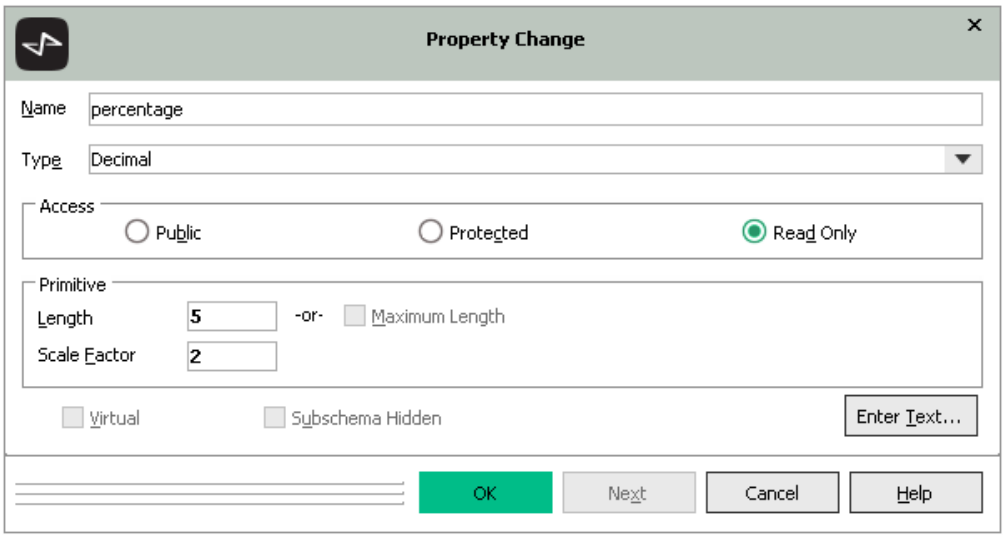
When renaming an identifier, special care must be taken to ensure that the references to the identifier refer to the new name after it changes.

The Jade Platform development environment provides the ability to perform this task automatically, avoiding the risk of human error creating compilation errors.

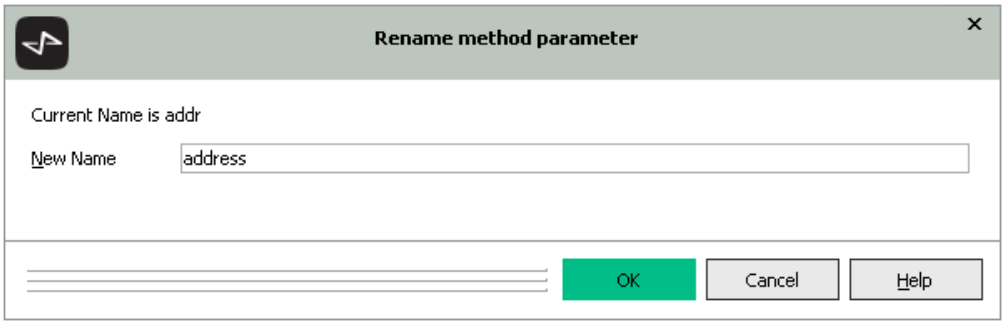
The steps required to safely rename an identifier vary, based on the type of identifier, as follows.

- For classes, properties, and references, right-click on the property and then select **Change**.

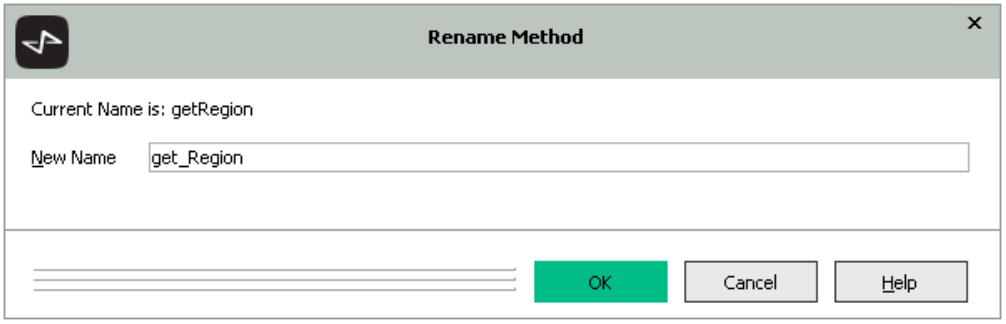
  Modify the **Name** value as required, so that all references to the new name are updated automatically when you click **OK**.



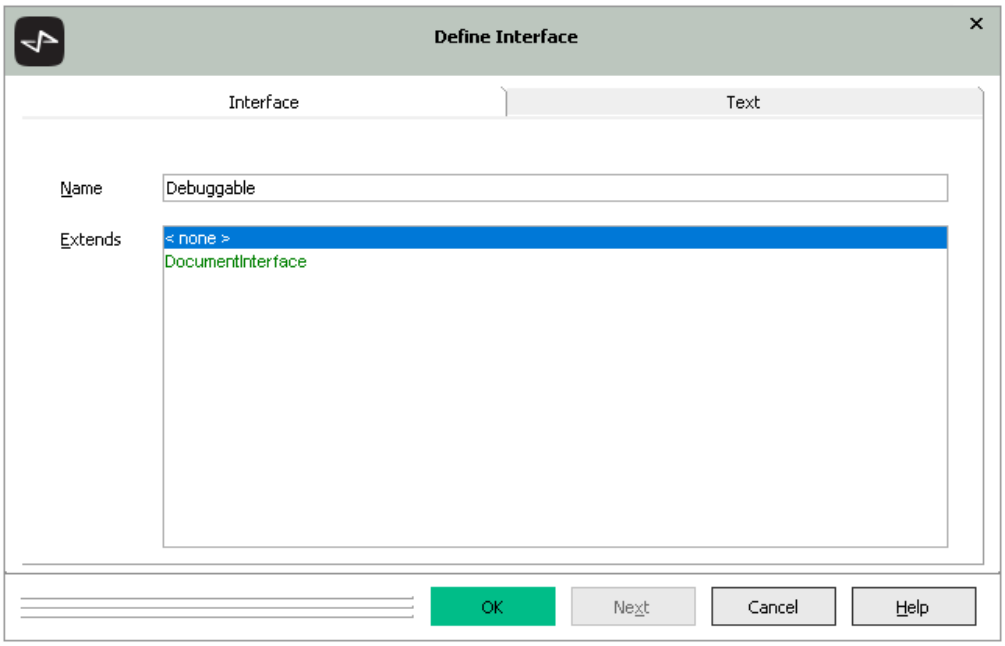
- For method parameters and local variables, right-click on the parameter or variable and then select the **Rename / Change** command from the Refactor submenu.



- For method names, right-click on the method and then select **Rename**, or select the **Rename** command from the Methods menu while the method is selected.



- For interfaces, open the Interfaces Browser (Ctrl+N), right-click on the interface, and then select **Change**. Modify the **Name** value as required, so that all references to the new name are updated automatically when you click **OK**.



Caution Schemas cannot be renamed once they have been created, so select the name carefully.

Exercise 4 – Finding and Fixing Bugs

In this exercise, locate and resolve several defects that have been intentionally introduced into the Erewhon demonstration system, by using the **Show Symbol**, **References**, and **Implementors** commands to navigate around the system and find the methods that are causing the provided unit tests to fail.

1. Load the Erewhon system provided in the **BrokenErewhon** folder of the additional modules Zip file, by selecting the **Load** command of the Schema menu, checking the **Load Multiple Schemas** check box, and then selecting **ErewhonInvestments.mul** as the schema file name.
2. Open **ErewhonInvestmentsModelSchema** in the Class Browser by selecting it in the Schema Browser and then using the Ctrl+B keyboard shortcut.
3. Select the **JadeTestCase** class and then press F9 to run the provided tests. You should see that the following tests fail.
   - TestTransactionAgent::testCreateCommRateOutsideTrx
   - TestTransactionAgent::testCreateCommissionRate
   - TestTransactionAgent::testUpdateCountry
   - TestClient::testGetAllSales
   - TestCountry::testInvalidName
   - TestCountry::testUpdate
   - TestSale::testGetDate

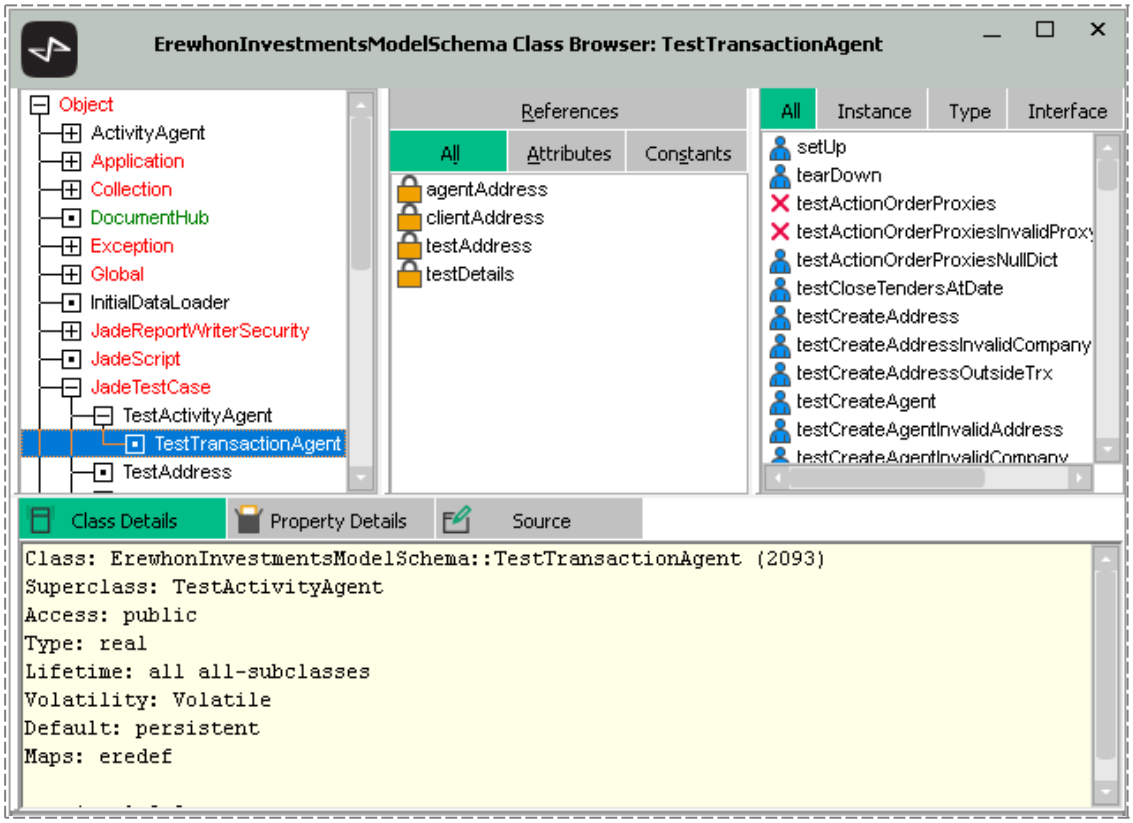
- TestRetailSale::testGetDate
- TestRetailSaleItem::testGetDebugString
- TestTenderSaleItem::testGetDebugString
- TestSaleItemCategory::testCreateCommissionRate

4. For each of these test failures, find the failing test in the Class Browser and use the F11 key to show the implementation of the method that is being tested by the unit test. Find the defect in the tested method and re-run the tests, to check that it is now fixed.

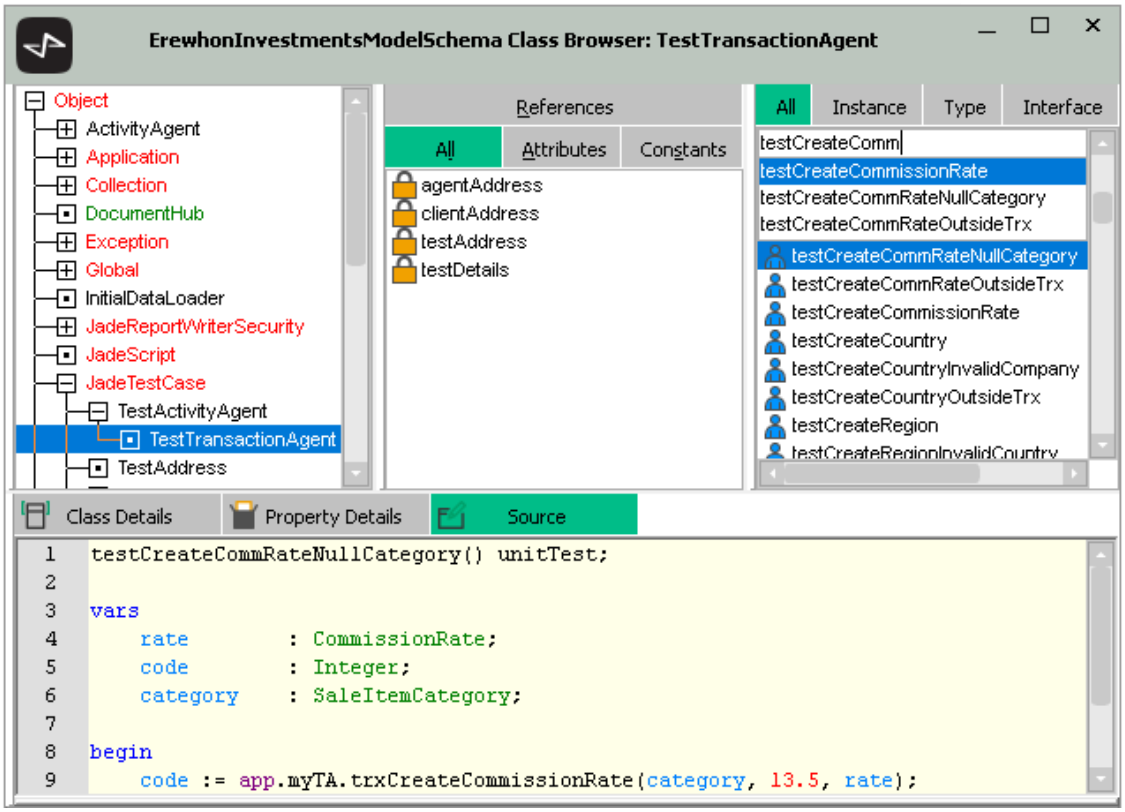
   To get you started, the fix for the first failing test is given as an example.

   a. The failing test is **TestTransactionAgent::testCreateCommRateOutsideTrx**, and the test failed with a 1090 error on the assert; that is, the created object was null when we tried to read it. We can therefore deduce that the method completed without error but did not create the object that it was supposed to.

   b. Find the **TestTransactionAgent** class in the Class Browser.

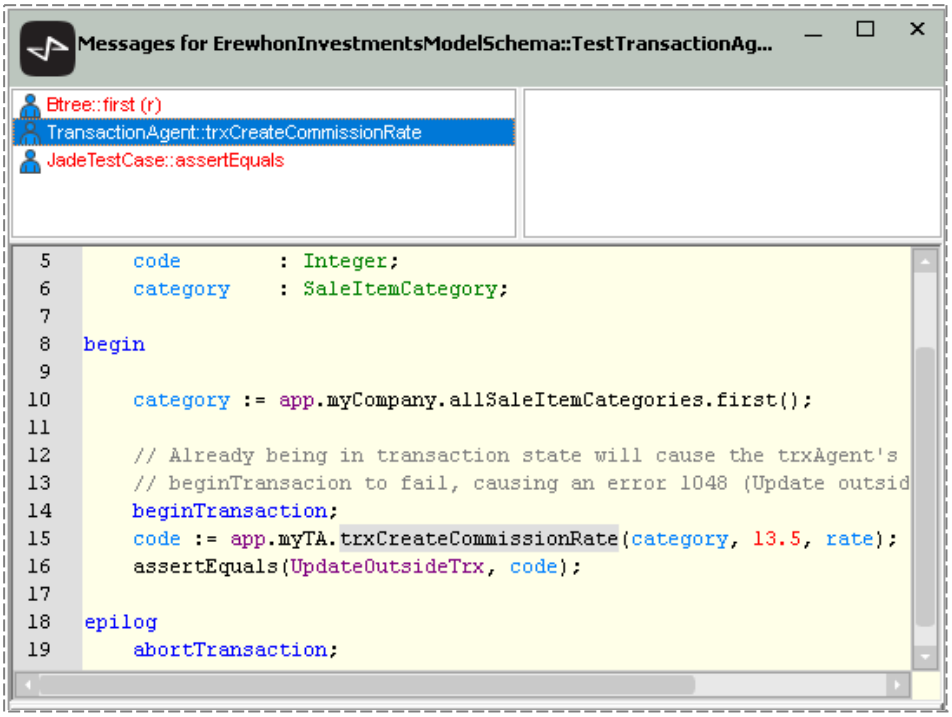


c. Use the Ctrl+7 key to display the method search text box at the top of the Methods List, and search for **testCreateCommRateOutsideTrx**.



d. Click the **testCreateCommRateOutsideTrx** method and then select the **Messages** command from the Methods menu. In this window, the two messages in red are calls to methods of system classes, which are not going to be the subject of user-defined unit tests.

As such, we can see that the one user method, **trxCreateCommissionRate**, must be the one being tested. Clicking on this method in the messages list will highlight it in the method source.



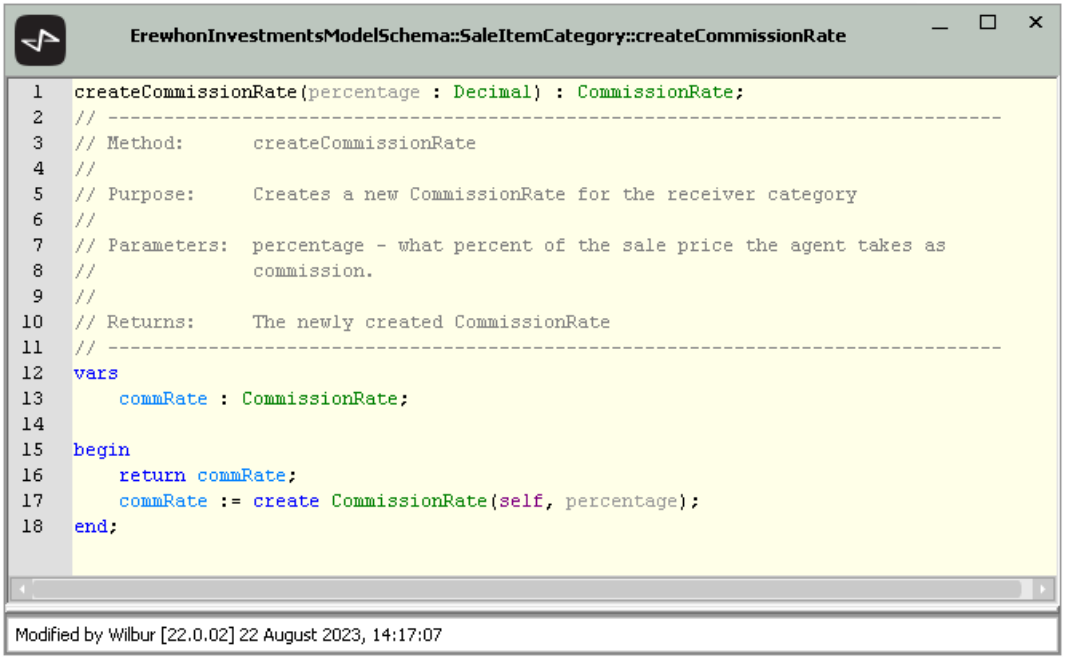
e. Clicking on the highlighted method and then pressing F11 will navigate to the implementation of the method, which is as follows.

ErewhonInvestmentsModelSchema::TransactionAgent::trxCreateCommissionRate

```
1  trxCreateCommissionRate(
2          saleItemCategory : SaleItemCategory;
3          percentage       : Decimal;
4          commissionRate   : CommissionRate output) : Integer;
5  // ------------------------------------------------------------------------------
6  // Method:       trxCreateCommissionRate
7  //
8  // Purpose:      Transaction method to create a CommissionRate
9  //
10 // Parameters:   Parameters correspond to each of the properties of a CommissionRate.
11 //               The newly created rate is returned in the output parameter.
12 //
13 // Returns:      0 if transaction successful, otherwise a model error number (refer
14 //               to the global constants in this schema for a list of error numbers)
15 // ------------------------------------------------------------------------------
16 begin
17     // Always initialize the activity first
18     self.zInitializeActivity;
19
20     // Arm our base exception handlers
21     on Exception do self.zExceptionHandler(exception);
22     on LockException do self.zLockExceptionHandler(exception);
23
24     if saleItemCategory = null then
25         return self.zSetErrorCode(InvalidSaleItemCategory);
26     endif;
27
28     // If saleItemCategory is invalid, our handler will return InvalidSaleItemCategory
29     self.zRegisterObjectAndErrorCode(saleItemCategory, InvalidSaleItemCategory);
30
31     // Start the transaction
32     beginTransaction;
33
34     // Perform the operation
35     commissionRate := saleItemCategory.createCommissionRate(percentage);
36
37     // If "createCommissionRate" raises an exception, our handler will resume to here
38     if app.noErrors then
39         // No errors, so commit
40         commitTransaction;
41     else
42         // We got an error so abort the transaction. Although TransactionAgent
43         // exception handlers abort the transaction before resuming, we do so
44         // here as well for completeness and symmetry with the beginTransaction.
45         // An abortTransaction when not in transaction state does nothing.
46         abortTransaction;
47     endif;
48
49 epilog
50     // Always return the error code to the caller (which will be zero if no errors)
51     return app.getLastError;
```

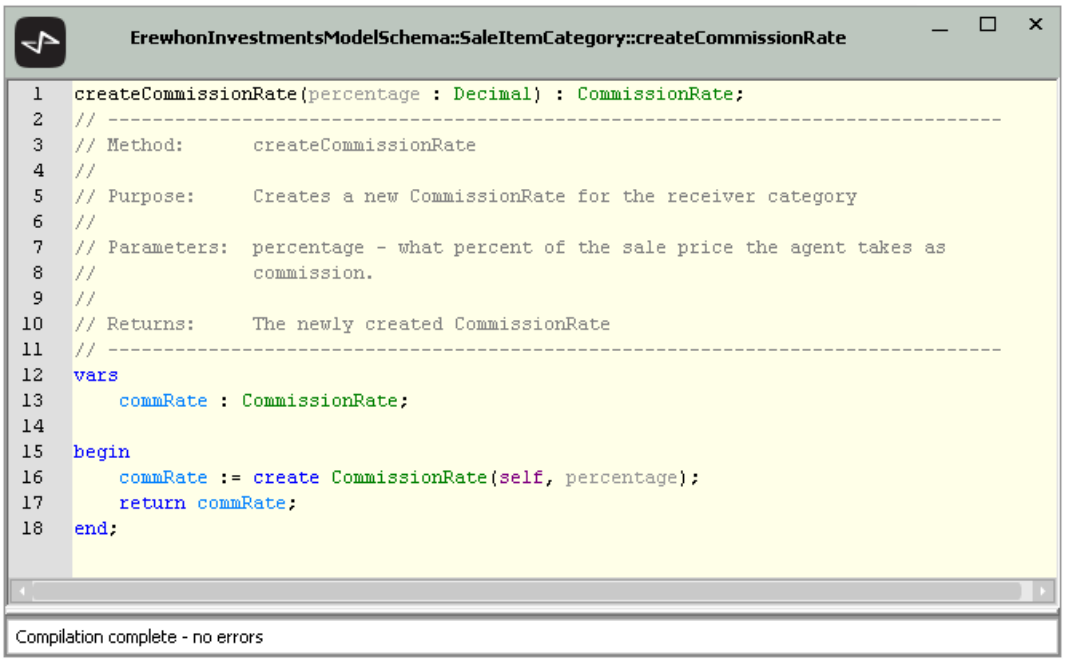
f. This method appears to be a wrapper method for another method, **createCommissionRate**.

To see the **createCommissionRate** method, select **createCommissionRate** and then press F11.



g. In this method, we can see that the **CommissionRate** is being returned before it is created.

To fix this, we simply reverse the order of the two statements and recompile (using the F8 key).



h. If we re-run the tests, we see that the following tests are no longer failing.

- TestTransactionAgent::testCreateCommRateOutsideTrx
- TestTransactionAgent::testCreateCommissionRate
- TestSaleItemCategory::testCreateCommissionRate

i. To see why these are no longer failing, we can view the references to the **createCommissionRate** method. With focus on the **createCommissionRate** method, select the **References** command from the Methods menu.

j. There are four test methods included in the references: the three that are no longer failing and the **TestTransactionAgent::testCreateCommissionRateNullCategory** method, which is an exception-flow test that verifies that the method rejects invalid inputs.

5. The following failing tests remain. Identify and remove the errors so that all tests pass.

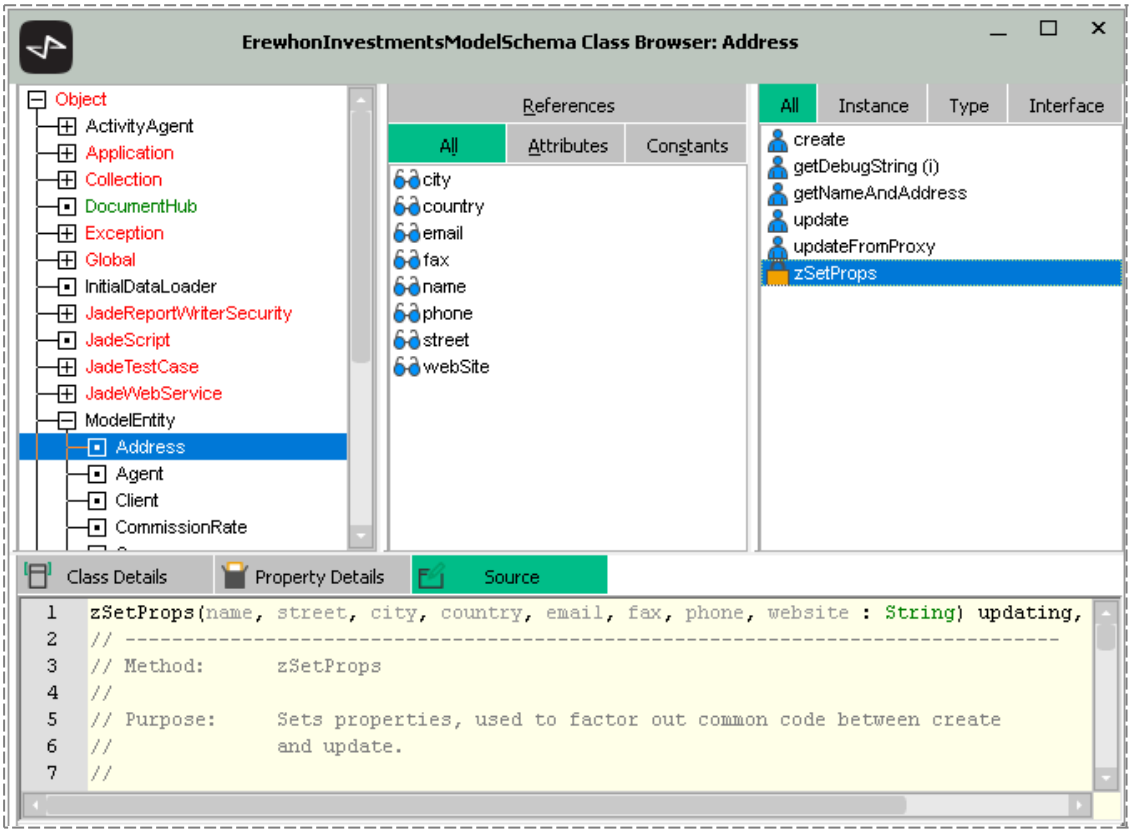
- TestTransactionAgent::testUpdateCountry
- TestClient::testGetAllSales
- TestCountry::testInvalidName
- TestCountry::testUpdate
- TestSale::testGetDate
- TestRetailSale::testGetDate
- TestRetailSaleItem::testGetDebugString
- TestTenderSaleItem::testGetDebugString

Tip Some fixes will solve multiple failing tests.

Exercise 5 – Refactoring Names

In this exercise, rename various entities and use the **References** and **Implementors** commands to verify that the renaming has correctly modified the entities' references and implementors.

1. In the Class Browser for the **ErewhonInvestmentsModelSchema**, navigate to the **Address** class, which is a subclass of **ModelEntity**, and find the **zSetProps** method.



Note The **z** prefix on private methods is an old naming convention that was used to distinguish between public and private methods before the functionality to add a visual aid to methods in the Class Browser was implemented.

2. With the **zSetProps** method highlighted, select the **Implementors** command from the Methods menu to see all classes in the current schema that implement this method.

3. For each method found, perform the following actions.

   a. With the method highlighted, select the **References** command from the Methods menu to see which methods refer to that specific **zSetProps** method.

   b. Select the **Rename** command from the Methods menu and rename the method to **setProperties**.

   c. Check the references list for the method to ensure that all references have been updated to **setProperties**.

Note You will see that the method header comment at the top of the changed method still includes **Method: zSetProps**. Comments are unaffected by the renaming of identifiers, which is one disadvantage to the use of method headers, as they can become out of date with the actual names and behaviors within the method.

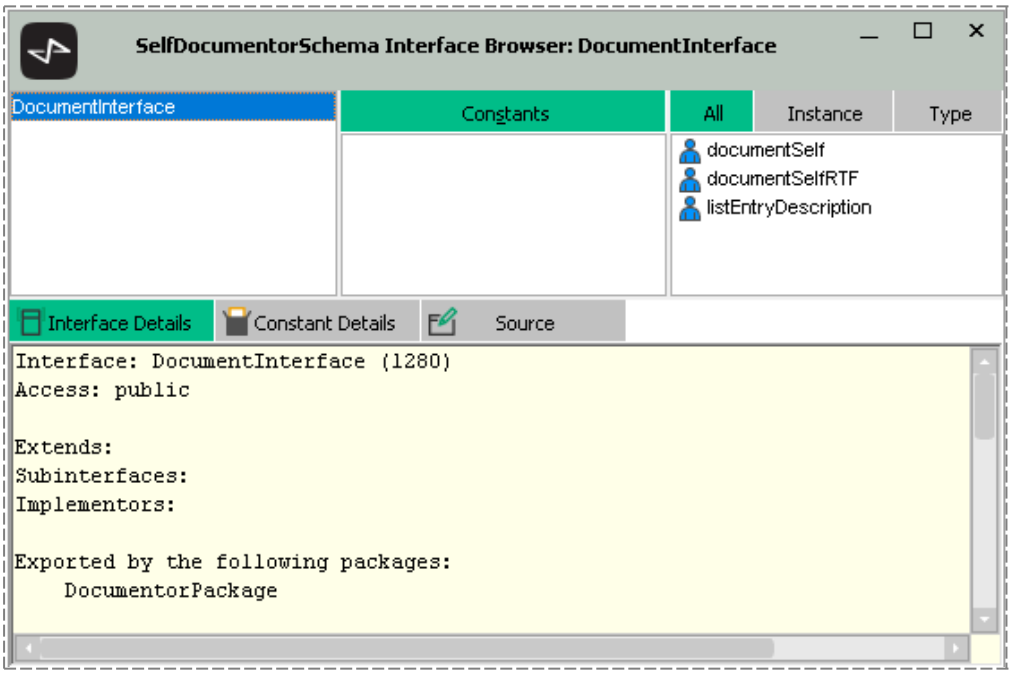
4. In the Class Browser for **ErewhonInvestmentsModelSchema**, navigate to the **InitialDataLoader** class and then find the **zGetNextToken** method.

5. Rename the following parameters and variables by right-clicking on them and then selecting the **Rename / Change** command from the Refactor submenu.

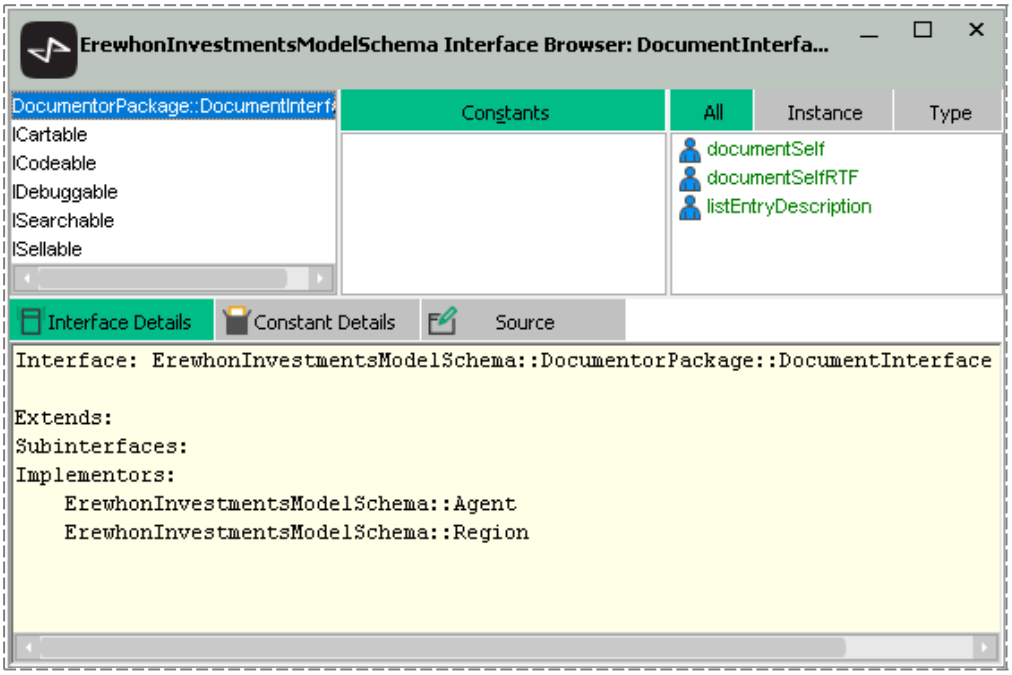
Old Name	New Name	Identifier Type
str	targetString	Parameter
pos	currentPosition	Parameter
len	stringLength	Variable
idx	currentIndex	Variable

Note You can remove the end-of-line comments for the variables, as it is better practice to use descriptive variable names rather than abbreviations with comments describing what they represent.

6. Select **SelfDocumentorSchema** in the Schema Browser and then press Ctrl+N to open the Interface Browser. You will see that this schema contains one interface: **DocumentInterface**.



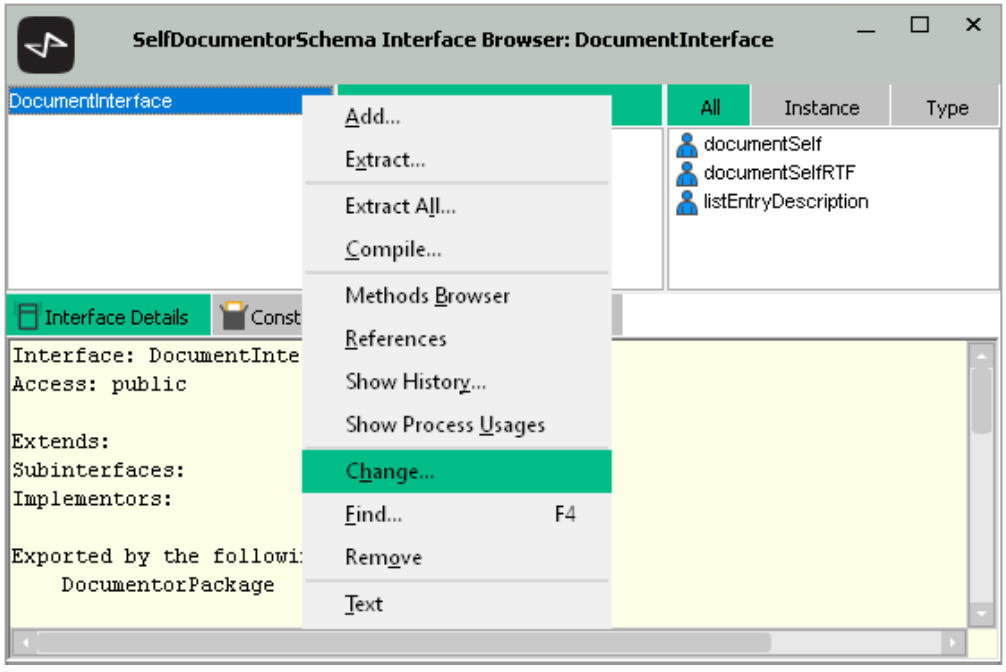
7. With the Interface Browser still open, select **ErewhonInvestmentsModelSchema** in the Schema Browser and then press Ctrl+N to open another Interface Browser.



In this Interface Browser, you can see that all Erewhon interfaces begin with a capital **I** prefix and are of the form I*verbable*. This is one of the two common interface naming conventions, along with the form *responsibility*IF, which is used in the RootSchema interfaces.

However, **DocumentInterface** does not conform to either style, so we will rename it.

8. From the **SelfDocumentorSchema** Interface Browser, right-click **DocumentInterface** and then select the **Change** command.



9. Rename the interface to **IDocumentable** and then click **OK**, to save the change.

Unused Variables

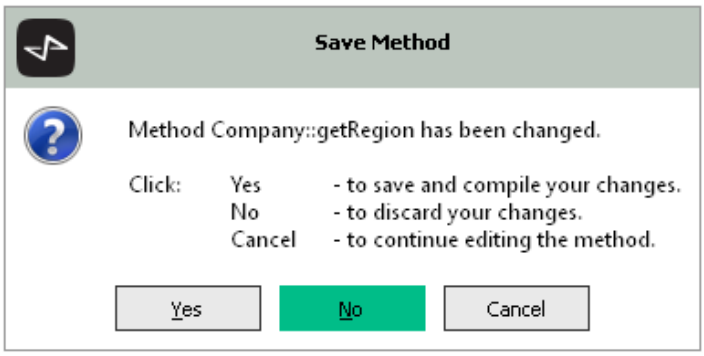
In the Jade language, you must always declare a variable before you use it; that is, attempting to use an undeclared variable results in compiler error 6027 (*Unknown identifier*).

Although no error is generated when declaring a variable and never using it in the method, it is undesirable behavior because unused variables make the code more cluttered and less-readable.

The Jade Platform development environment includes the ability to find and optionally remove these unused variables. To remove unused variables for a specific method, right-click on the method and then select the **Unused Local Variables** command. The Find Unused Variables dialog is then displayed, advising you that the specified local variable is unused. Buttons enable you to click:

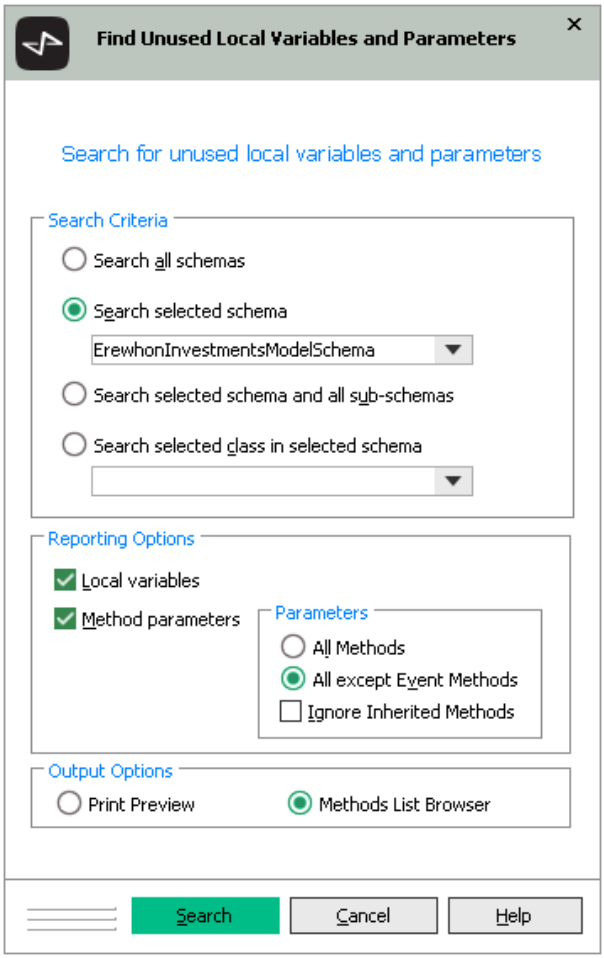
- **Find Next**, to skip the current unused variable and go to the next one
- **Remove**, to remove the current unused variable
- **Remove All**, to remove all unused variables from the method, including those that were previously skipped
- **Cancel**, to close the window as well as removing any previously removed unused variables

After removing unused variables, the method will not automatically save or compile. If you want to compile it, locking in the removal, press F8 or select the **Compile** command from the Methods menu.



Alternatively, if you navigate away from the method without saving or compiling it and then click **No** on the Save Method dialog, the unused variables are returned to the method.

You can also search for all unused variables in classes, schemas, or even the whole database by selecting the **Find Unused Local Variables/Parameters** command from the Schema menu when the Schema Browser has focus.



In the Find Unused Local Variables and Parameters dialog, specify the scope of the search by selecting one of the option buttons in the Search Criteria group box. To search:

- All schemas in the database, select **Search all schemas**
- Within a specific schema, select **Search selected schema** and then select the required schema from the associated combo box
- Within a specific schema and any subschemas of that schema, select **Search selected schema and all sub-schemas** and then select the required schema from the associated combo box

  Note The **Search selected schema**, **Search selected schema and all sub-schemas**, and **Search selected class in selected schema** options share the schema selection combo box (that is, the first one in the Search Criteria group box).

- Within a specific class of a specific schema, select the **Search selected class in selected schema** option and then select the schema from the first combo box and the class from the second combo box

You can also specify the reporting options, by selecting options within the Reporting Options group box as follows.

- To skip local variables and search only for method parameters, uncheck the **Local variables** check box.
- To skip method parameters and search only for local variables, uncheck the **Method parameters** check box.

Note You must select at least one of these check boxes.

- If the **Method parameters** check box is checked, you can select the **All Methods** or **All except Event Methods** option button in the Parameters group box. Many event methods will have an unused parameter, so typically the **All except Event Methods** default option button should be selected. However, you can select the **All Methods** option button to include them.

In the Output Options group box, the default **Methods List Browser** option button causes the results of the search to display in a methods list to make them easier to resolve. However, if you want to print them instead, select the **Print Preview** option button.

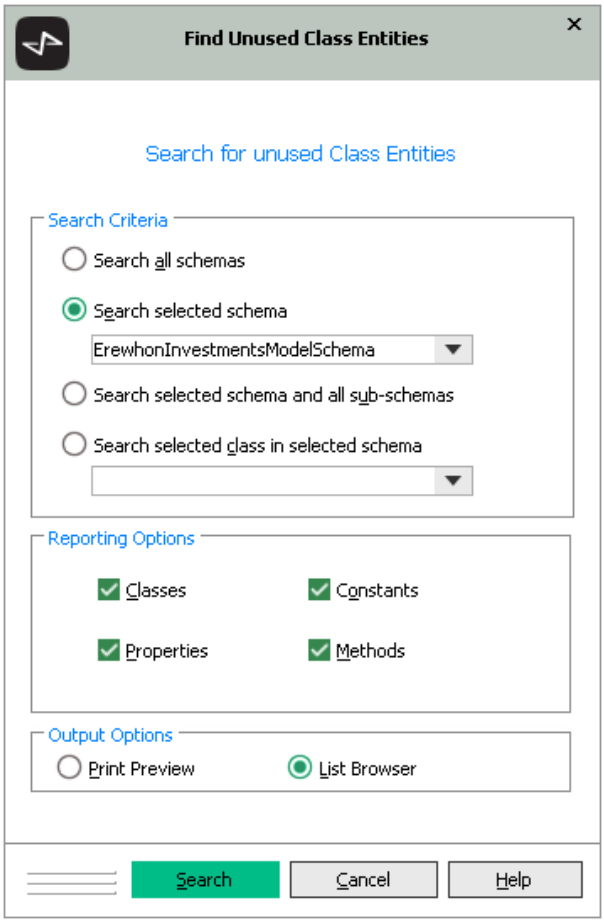
Unused Class Entities

As a code base grows and is maintained, it is common for dead code to arise; that is, code that is never called and therefore provides no value to the system but makes the code base larger and therefore harder to maintain.

The Jade Platform development environment provides the ability to locate and delete unused class entities to combat the accumulation of dead code.

While the Find Unused Class Entities dialog is a fast and powerful way to find and remove dead code, it will err on the side of avoiding the removal of possibly useful code so there may still be dead code remaining after its use.

To open the Find Unused Class Entities dialog, select the **Find Unused Class Entities** command from the Schema menu in the Schema Browser.



In this dialog, you can specify the scope of the search by selecting one of the option buttons in the Search Criteria group box. To search:

- All schemas in the database, select **Search all schemas**
- Within a specific schema, select **Search selected schema** and then select the required schema from the associated combo box
- Within a specific schema and any subschemas of that schema, select **Search selected schema and all sub-schemas** and then select the required schema from the associated combo box

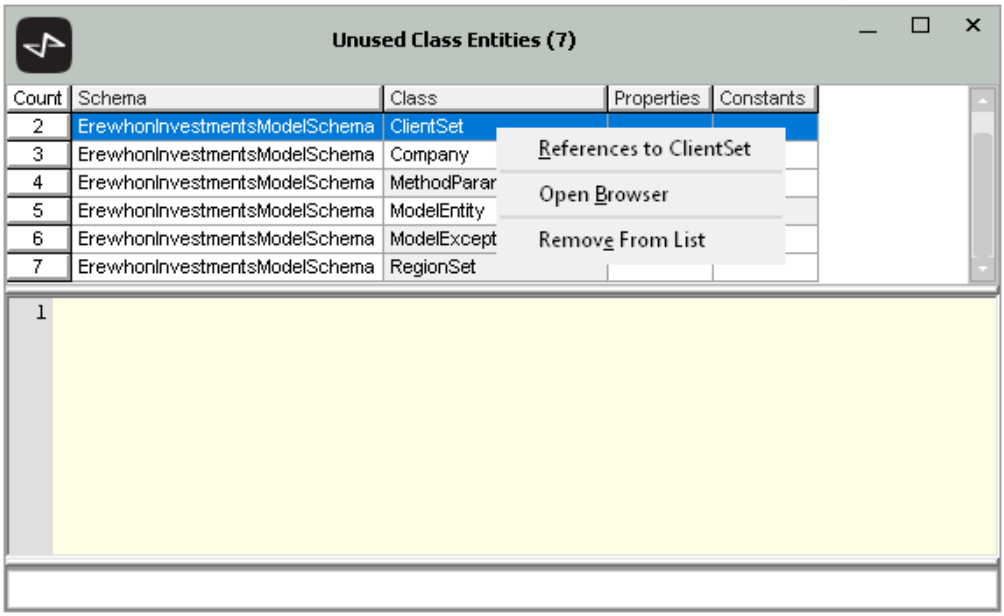
  Note The **Search selected schema**, **Search selected schema and all sub-schemas**, and **Search selected class in selected schema options** share the schema selection combo box (that is, the first one in the Search Criteria group box).

- Within a specific class of a specific schema, select the **Search selected class in selected schema** option and then select the schema from the first combo box and the class from the second combo box

In the Reporting Options group box, check the check box of each type of class entity you want find.

In the Output Options group box, the default **List Browser** option button causes the results of the search to display in a methods list to make them easier to resolve. However, if you want to print them instead, select the **Print Preview** option button.

After clicking **Search**, the Unused Class Entities dialog lists any unused entities that match your selected criteria.



Select an unused entity in the table and then right-click to access a popup (context) menu with the following commands.

- **References to *entity-name*** displays any references to the entity. While this will usually be none, a class can have a method referencing itself while still being unused.
- **Open Browser** displays the entity in a new Class Browser.
- **Remove From List** removes the entity from the current Unused Class Entities dialog but it does not remove it from its schema or prevent it from being located the next time an unused class entities search is performed.

Exercise 6 – Locating and Removing Unused Variables

In this exercise, locate and remove unused variables, first in a specific method and then in an entire schema.

1. In the **ErewhonInvestmentsModelSchema**, navigate to the **InitialDataLoader** class **zLoadAgents** method.

   You will see that there are many declared variables in the method.

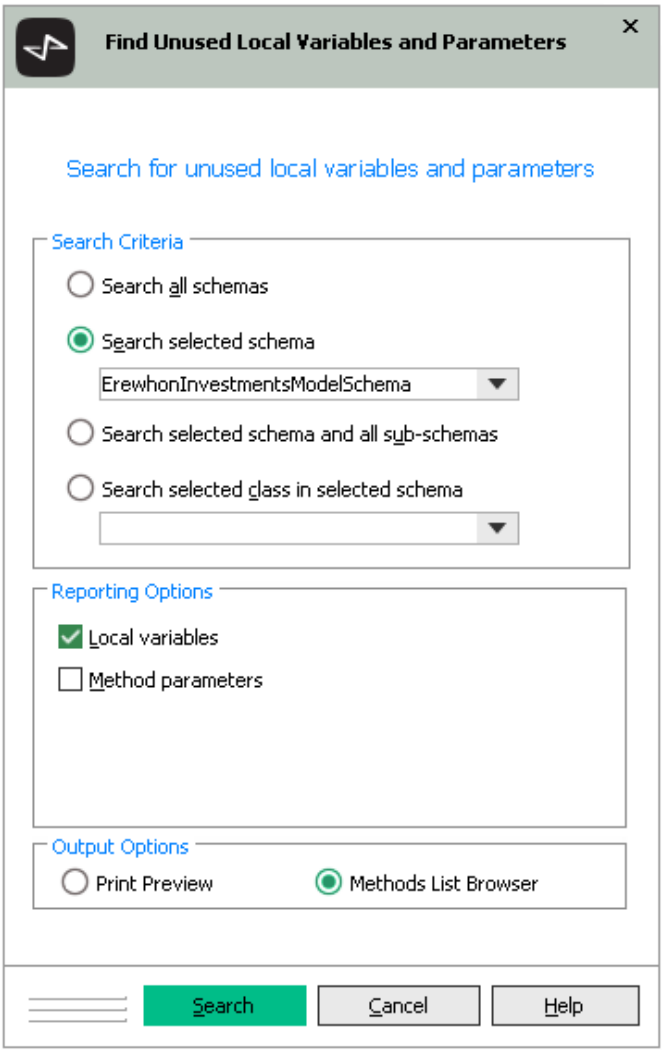
```
vars
    inputFile       : File;
    agent           : Agent;
    agentAddress    : Address;
    fileName        : String;
    line            : String;
    agentName       : String;
    street          : String;
    address         : String;
    address2        : String;
    address3        : String;
    email           : String;
    phone           : String;
    fax             : String;
    web             : String;
    pos             : Integer;
    startClock      : Integer;
```

2. Find the three unused variables by selecting the **Unused Local Variables** command from the Methods menu and then clicking **Find Next** until the following message box is displayed.

3. Reopen the Find Unused Variables dialog and then click **Remove All**. The three unused variables are then removed from the method.

4. Press F8 to compile the method.

5. Select **ErewhonInvestmentsModelSchema** in the Schema Browser and then select the **Find Unused Local Variables/Parameters** command from the Schema menu.



6. Uncheck the **Method parameters** check box and then click **Search**. This narrows the search to unused local variables only.

7. For each unused variable that is found, double-click it in the Unused Local Variable and Parameters dialog. The unused variable is then removed from the method, the method is automatically compiled, and the entry is removed from the list.

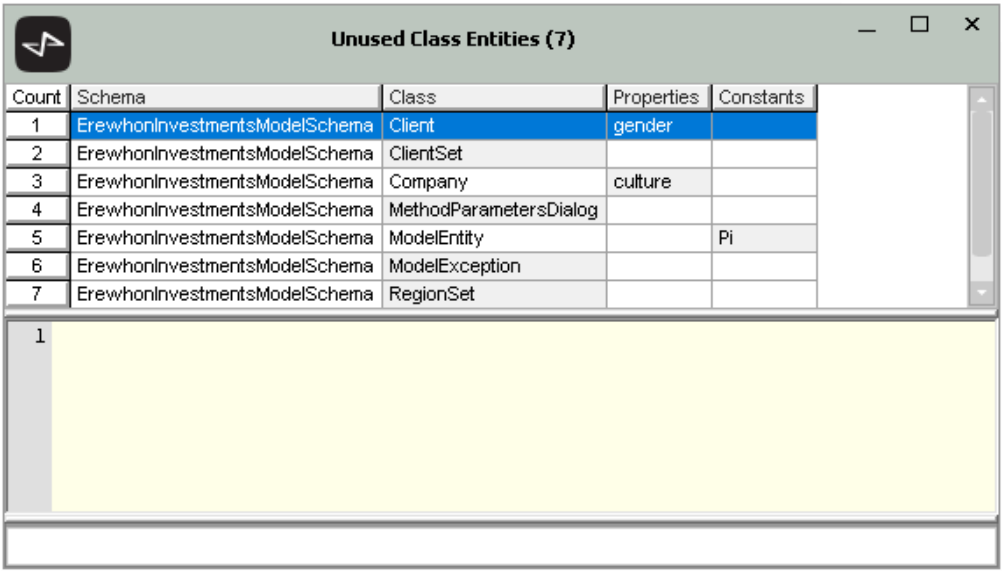
> **Note** You can use the double-click shortcut on unused variables only. If a method has an unused parameter, more care must be taken as the removal of that parameter will change the method signature, which may affect other methods.

Exercise 7 – Locating and Removing Unused Class Entities

In this exercise, locate and remove several unused classes, two unused properties, and an unused constant by using the **Find Unused Class Entities** command.

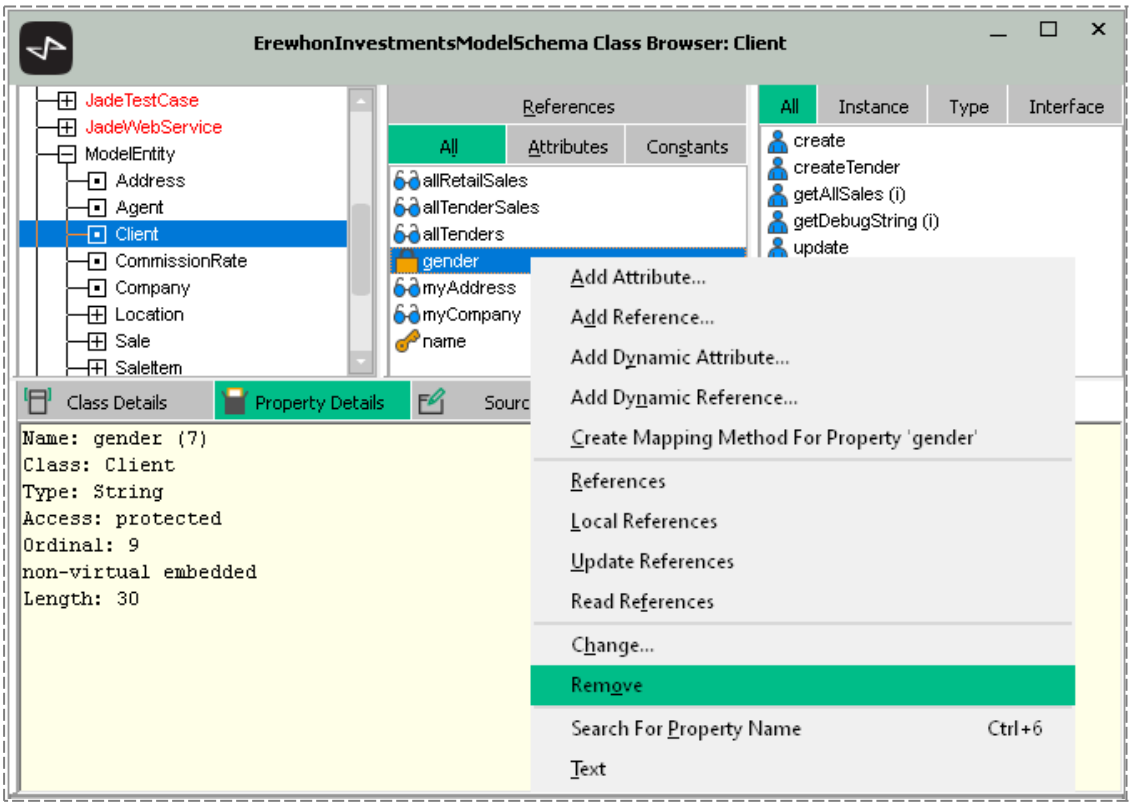
1. Select **ErewhonInvestmentsModelSchema** in the Schema Browser and then select the **Find Unused Class Entities** command from the Schema menu.

The Unused Class Entities list is then displayed.



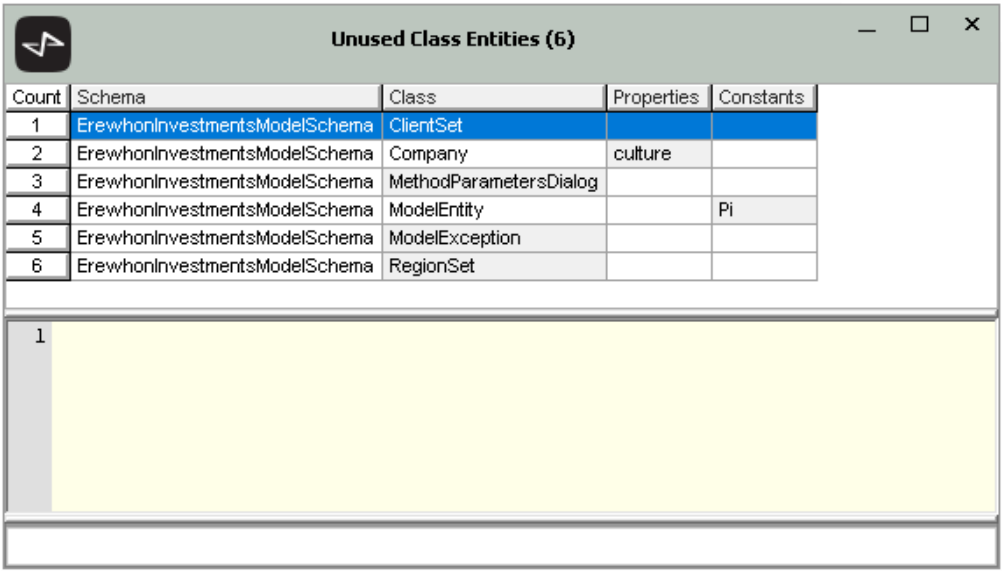
2. For each of the found entities, select the entity and then right-click it and select the **Open Browser** command.

3. A new Class Browser is opened, with the selected entity highlighted. From this Class Browser, right-click on the entity and then select the **Remove** command.



4. In the Confirm Delete message box, click **OK**.

5. The entity is then automatically removed from the Unused Class Entities list. Repeat these steps for each unused class entity until the list is empty.



Transient Leaks

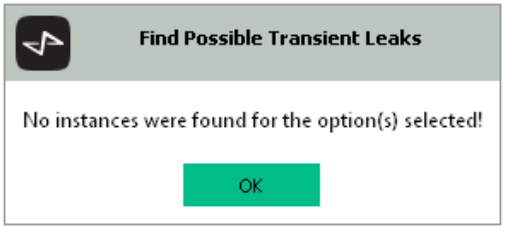
A transient leak is when a transient object is created but not deleted after use. In large, long-running applications, transient leaks can have a significant impact on performance as the transient cache has finite capacity and once filled, transients must be saved to a transient database, which is much slower than the transient cache.

The Jade Platform development environment provides the ability to identify methods in which a transient is created but not deleted and it is a useful tool to identify transient leaks; however, the identified transients may not necessarily be leaks if they are deleted elsewhere in the code.

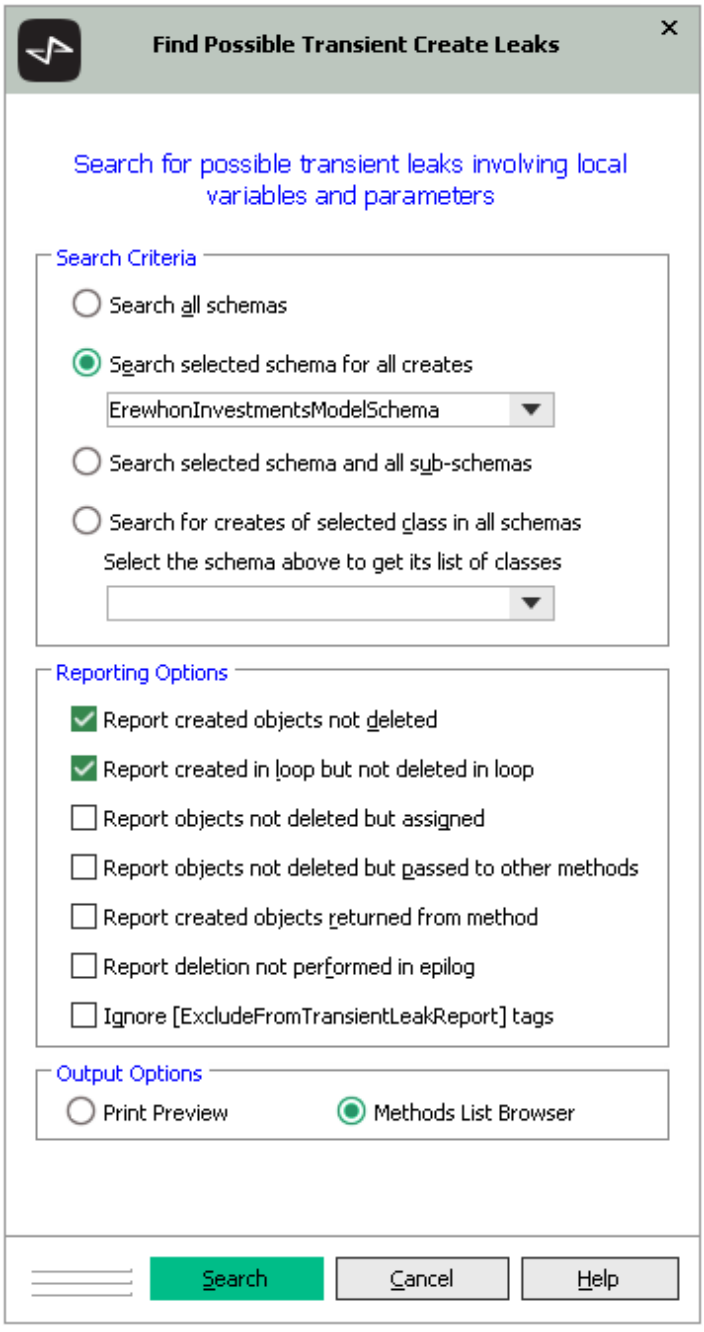
When creating a transient object that you are confident will be deleted but that may generate a false positive for a transient leak, you can add **[ExcludeFromTransientLeakReport]** into an end-of-line comment after the **create** instruction, to ignore it. For example, the following **JadeScript** method will leak the transient **Company** named **leak**.

```
leakExample();

vars
    leak : Company;
begin
    create leak transient; // [ExcludeFromTransientLeakReport]
end;
```

However, by including **[ExcludeFromTransientLeakReport]**, the method will not appear in a transient leak search.



To open the Find Possible Transient Create Leaks dialog, select the **Find Possible Transient Leaks** command from the Schema menu.



From this dialog, you can specify the scope of the search by selecting one of the options in the Search Criteria group box. To search:

- All schemas in the database, select **Search all schemas**.
- Within a specific schema, select **Search selected schema** and then select the required schema from the associated combo box.
- Within a specific schema and any subschemas of that schema, select **Search selected schema and all sub-schemas** and then select the required schema from the associated combo box.

  Note The **Search selected schema**, **Search selected schema and all sub-schemas**, and **Search selected class in selected schema** options share the schema selection combo box (that is, the first one in the Search Criteria group box).

- Within a specific class of all schemas, select **Search selected class in all schemas** and then select the class from the second combo box. The selected class is then searched for across all schemas in which it exists.

The Reporting Options group box is used to specify the possible leak conditions to be searched for, as follows. The:

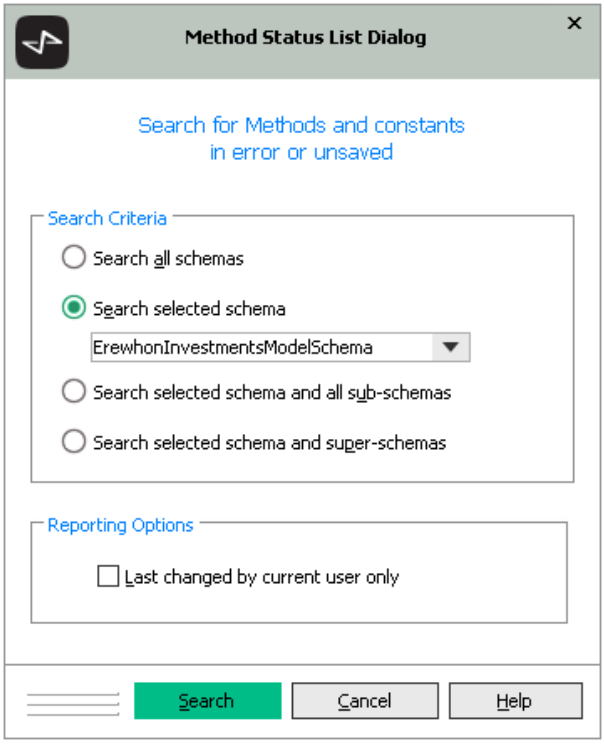
- **Report created objects not deleted** check box finds transients that are created but not deleted within the method, nor passed to another method, nor returned in a **return** instruction. These objects will usually result in leaks.
- **Report created in loop but not deleted in loop** check box finds transients that are created in a loop but not deleted inside that loop, but may be deleted outside the loop.
- **Report objects not deleted but assigned** check box finds transients that are created and not deleted, but are assigned to another property or variable that may or may not be deleted later.
- **Report objects not deleted but passed to other methods** check box finds transients that are created but not deleted within the method, but are passed to another method as a parameter where they may or may not be deleted.
- **Report created objects returned from method** check box finds transients that are created but not deleted within the method, but are returned in the **return** instruction of the method. These may or may not be leaks, depending on what the calling method does with the object.
- **Report deletion not performed in epilog** check box finds transients that are not deleted in the epilog. Such objects may leak if an exception occurs in the method between when the object is created and when it is deleted, returned, or passed to another method.
- **Ignore [ExcludeFromTransientLeakReport] tags** check box finds transients that could generate a leak according to the other options selected, even if they are marked with a **[ExcludeFromTransientLeakReport]** tag.

In the Output Options group box, the default **Methods List Browser** option button causes the results of the search to display in a methods list to make them easier to resolve. However, if you want to print them instead, select the **Print Preview** option button.

Methods Status List

The Jade Platform development environment provides the ability to quickly find all methods that are unsaved or have errors in a specific schema or schemas.

Open the Method Status List dialog by selecting the **Status List** command from the Browse menu or with the Ctrl+Shift+C shortcut keys.

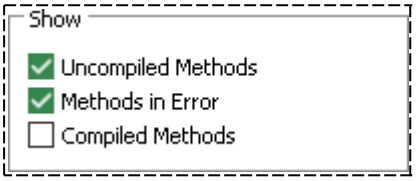


From this dialog, you can specify the scope of the search by selecting one of the options in the Search Criteria group box. To search:

- All schemas in the database, select **Search all schemas**.
- Within a specific schema, select **Search selected schema** and then select the required schema from the associated combo box.
- Within a specific schema and any subschemas of that schema, select **Search selected schema and all sub-schemas** and then select the required schema from the associated combo box.
- Within a specific schema and any superschemas of that schema, select **Search selected schema and all super-schemas** and then select the required schema from the associated combo box.

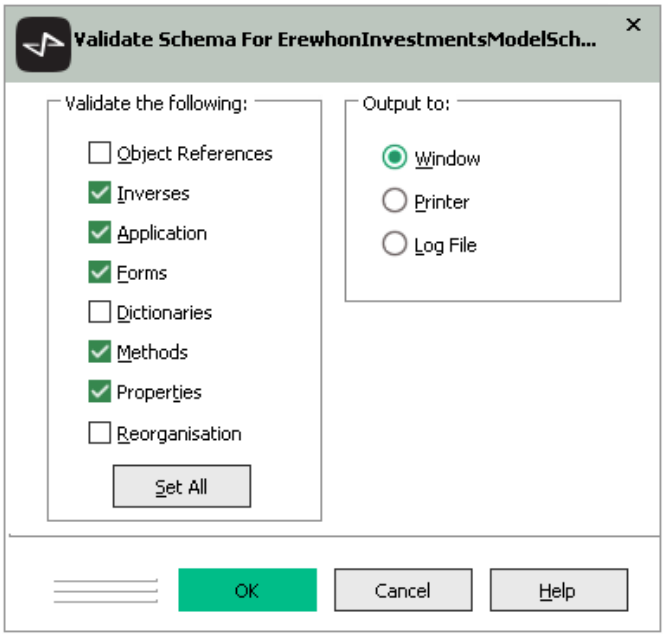
If you want to see only methods that were last changed by you, select the **Last changed by current user only** check box in the Reporting Options group box.

Once you have selected your required options, click **Search** to perform the search. By default, the Method Status List finds methods in error and uncompiled methods. However, this can be customized if needed, by selecting options in the Show group box on the **Status List** sheet of the Preferences dialog.



Schema Validation

The Validate Schema dialog provides validation for a variety of elements in the current schema. Access it by selecting the **Validate** command from the Schema list of the Schema Browser.



From the Validate the following group box, check the check boxes of the elements to be validated, as follows. The:

- **Object References** option finds and lists any objects that are referenced but no longer exist. This should normally never occur, so these errors suggest that the schema is corrupted.
- **Inverses** option finds any invalid inverse reference definitions.
- **Application** option finds any invalid application definitions.
- **Forms** option finds any invalid form definitions; for example, a form that contains a control with no corresponding property.
- **Dictionaries** option lists any invalid dictionary definitions. This should normally never occur, so these errors suggest that the schema is corrupted.
- **Methods** option outputs the same list as that generated by the Method Status List dialog.
- **Properties** option outputs a list of all properties in a class that have duplicate feature numbers.

- **Reorganisation** option validates and reports on the following, both of which suggest corruption of the schema.
  - If a schema has been marked as versioned but there are no classes versioned.
  - If there are classes versioned but the schema has not been marked as versioned.

The Validate Schema dialog provides the following output options that are selected from the Output to group box.

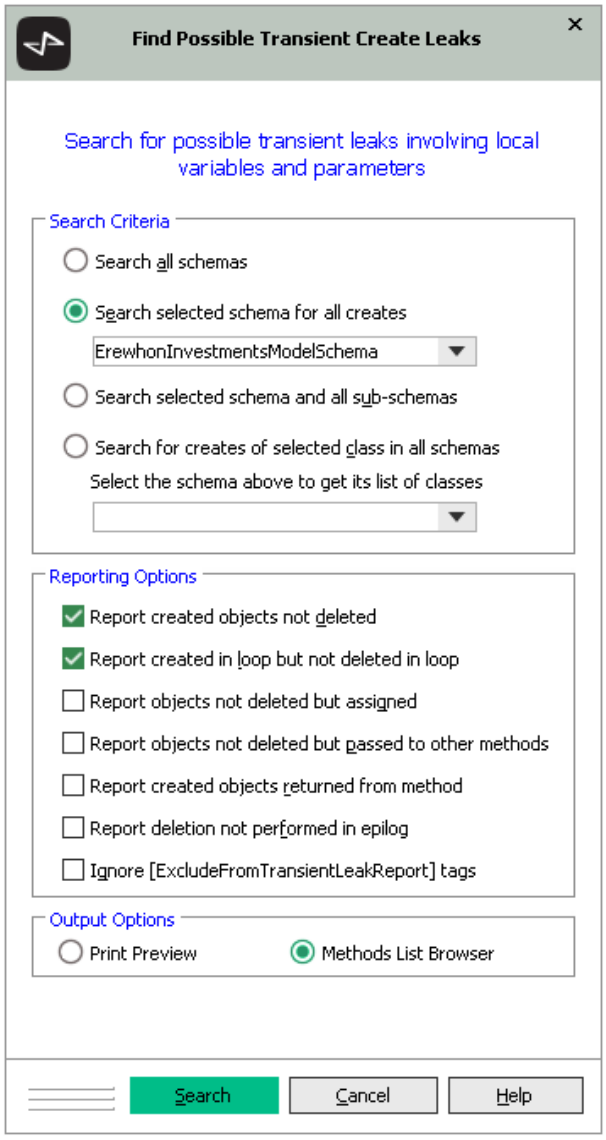
- **Window** (the default), which displays the results of the validation to the Jade Interpreter Output Viewer.

- **Printer**, which saves the results to a PDF file called **Jade.pdf** in your **Documents** directory of your Jade Platform installation.
- **Log File**, which saves the results to the **valscm.log** file in the **bin** directory of your Jade Platform installation.

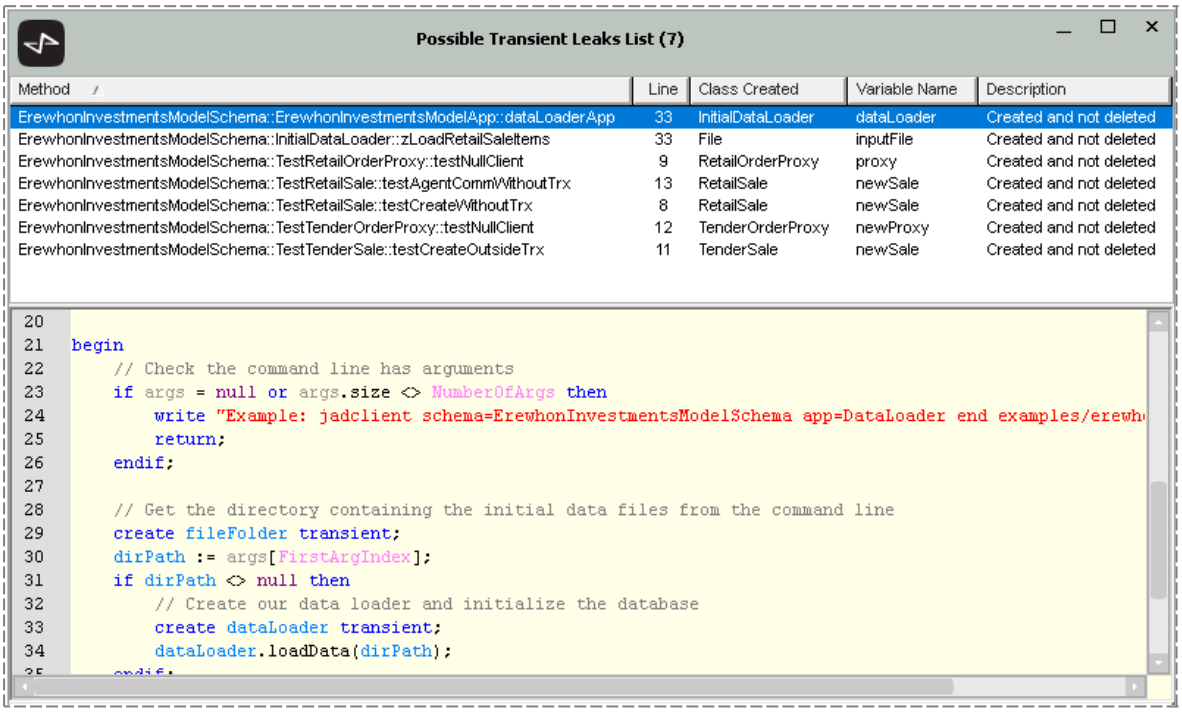
Exercise 8 – Finding and Removing Transient Leaks

In this exercise, locate and remove transient leaks from **ErewhonInvestmentsModelSchema**, using the **Find Possible Transient Leaks** command.

1. With **ErewhonInvestmentsModelSchema** selected in the Schema Browser, select the **Find Possible Transient Leaks** command from the Schema menu.



2. Click **Search**, to display the list of possible transient leaks.



You should see that the following methods have potential transient leaks.

- ErewhonInvestmentsModelApp::dataLoaderApp
- InitialDataLoader::zLoadRetailSaleItems
- TestRetailOrderProxy::testNullClient
- TestRetailSale::testAgentCommWithoutTrx
- TestRetailSale::testCreateWithoutTrx
- TestTenderOrderProxy::testNullClient
- TestTenderSale::testCreateOutsideTrx

3. For each method, delete the created transient in the epilog section of the method. For example, for the **ErewhonInvestmentsModelApp** class **dataLoaderApp** method, the epilog section should be as follows.

```
epilog
    delete fileFolder; // does nothing if fileFolder is null
    delete dataLoader;
    terminate;
end;
```

4. When you have added the **delete** instructions to the epilog, use the **Find Possible Transient Leaks** command again, to verify that the transient leaks have been resolved.

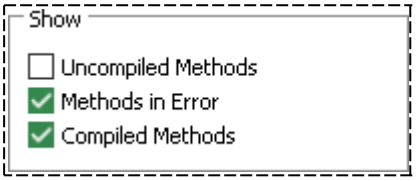
Exercise 9 – Finding Changed Methods

In this exercise, explore the use of the Method Status List dialog.

You will first use the Method Status List dialog to find all methods that you have changed throughout this module.

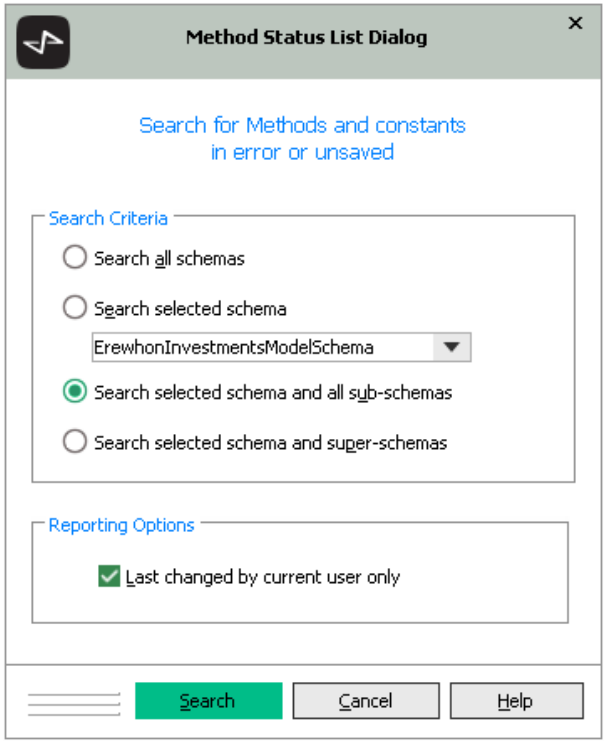
You will then use the Method Status List Browser to find and resolve compiler errors by first intentionally breaking some methods and then using the Method Status List Browser to repair the changes.

1. In the first exercise in this module, you configured some user preferences. Open the Preferences dialog and ensure that the **Status List** sheet has the following settings in the Show group box.



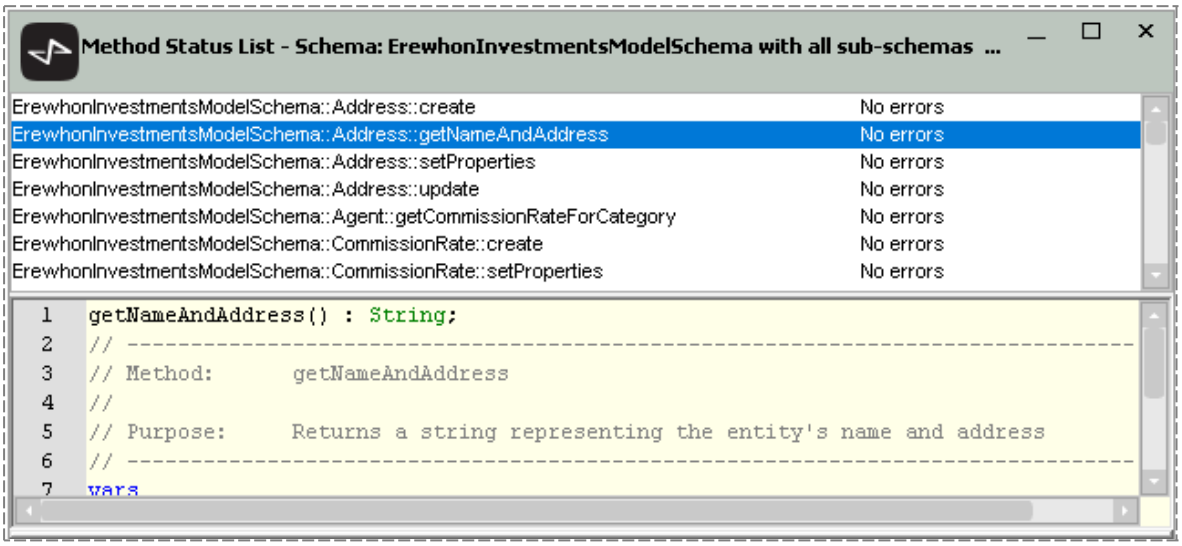
2. Select **ErewhonInvestmentsModelSchema** in the Schema Browser and then open the Method Status List dialog using the Ctrl+Shift+C shortcut keys.

3. In the Search Criteria group box, select the **Search selected schema and all sub-schemas** option button and in the Reporting Options group box, check the **Last changed by current user only** check box.



4. Click **Search**.

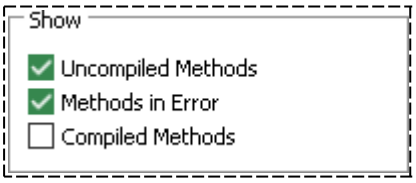
The Method Status List Browser then displays a list of all methods that you have changed.



Note Your list will show different methods from those in this example.

You should see that none of the methods are in error. You will now intentionally break some methods so that the Method Status List has something to find.

5. Open the Preferences dialog and set the following preferences on the **Status List** sheet.



6. Navigate to the **Address** class **getNameAndAddress** method, which is a subclass of **ModelEntity**.

7. Delete the **end;** instruction on line 12, as follows.

```
getNameAndAddress() : String;
// ----------
// Method:       getNameAndAddress
//
// Purpose:      Returns a string representing the entity's name and address
// ----------
vars
    name    : String;
begin
    return name & ", " & street & ", " & city & ", " & country;
```

8. Compile the method (press F8). A compiler error will be raised.

9. Navigate to the **Tender** class **getDate** method, which is also a subclass of **ModelEntity**.

10. Comment out the return statement on line 8 by adding // to the beginning of the line, as follows.

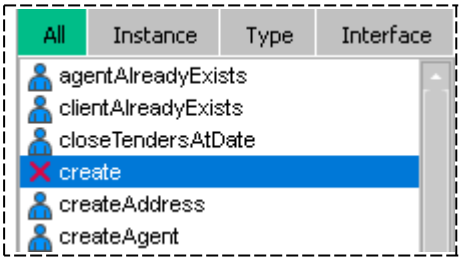
```
getDate() : Date;
// ------------------------------------------------------------------------------
// Method:       getDate
//
// Returns:      The offer date of the tender
// ------------------------------------------------------------------------------
begin
//  return timeStamp.date;
end;
```

11. Compile the method. Compiler error **6114 - Method does not return a value** will be raised.

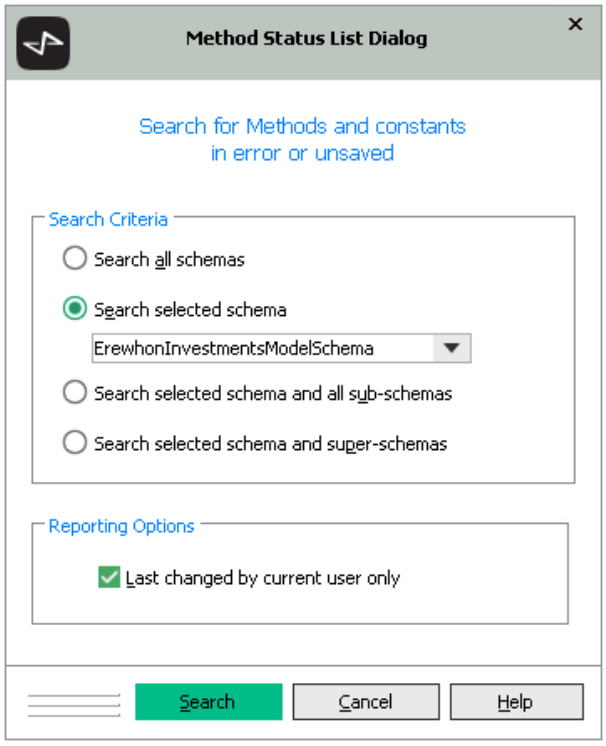
12. Navigate to the **Company** class **create** method.

13. Delete the commented-out line on line 11 (immediately after the **begin** instruction).

14. Press F2 to save the method without compiling it. You should see that it now has an **X** to the left of the method name in the Class Browser.



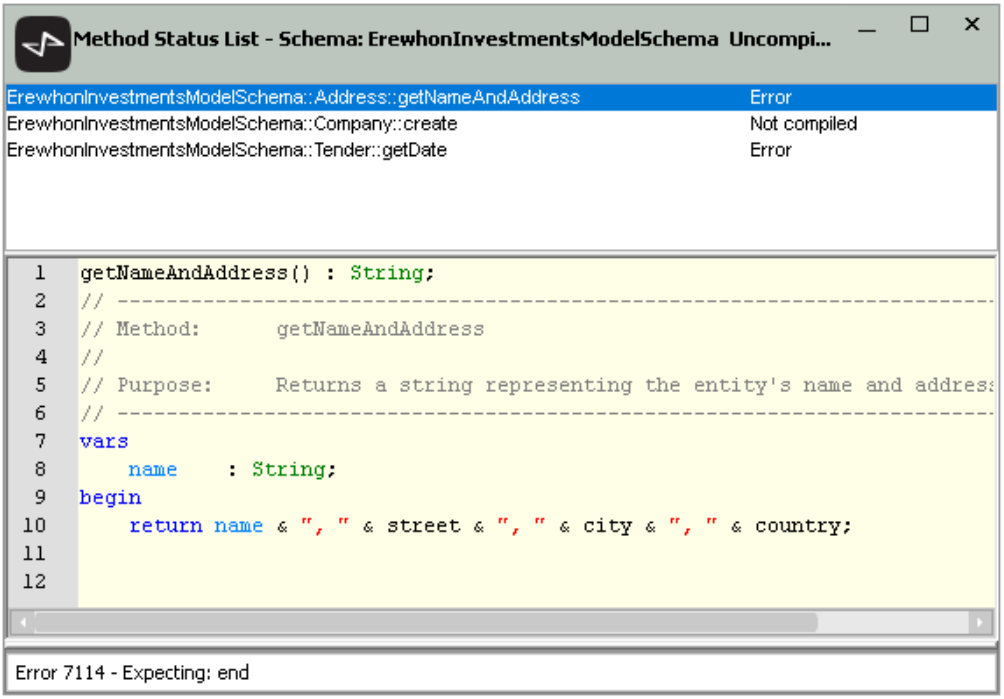
15. Select **ErewhonInvestmentsModelSchema** in the Schema Browser and then press Ctrl+Shift+C, to open the Method Status List dialog.



16. Check the **Last changed by current user only** check box and then click **Search**.

Tip In a single user system, **Last changed by current user only** has no effect, as there is only one user changing methods. However, when working collaboratively, it can be useful to see only those methods that you have changed.

The Method Status List Browser will then display the three methods you changed that have errors or are unsaved.



17. For each method, fix the error and then compile the method.

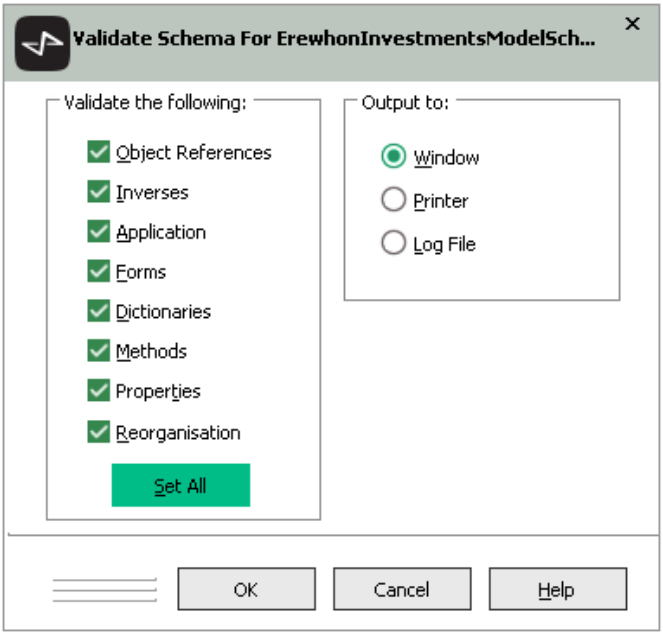
You can do this directly from the Method Status List Browser, and each method will automatically be removed from the list as it is completed.

Exercise 10 – Validating the Schemas

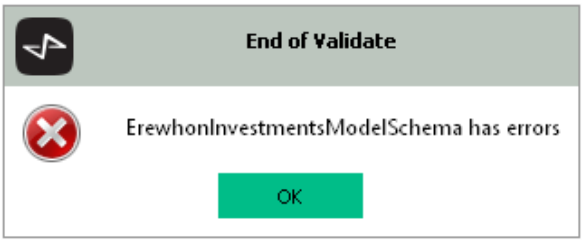
In this exercise, use the schema validator to find inconsistencies in the **ErewhonInvestmentsModelSchema** and then resolve these inconsistencies.

1. With **ErewhonInvestmentsModelSchema** selected in the Schema Browser, select the **Validate** command from the Schema menu.

The Validate Schema dialog is then displayed.



2. Click **Set All** to select all possible validations and then click **OK** to run the validator.
3. The following message box should be displayed. Click **OK**.



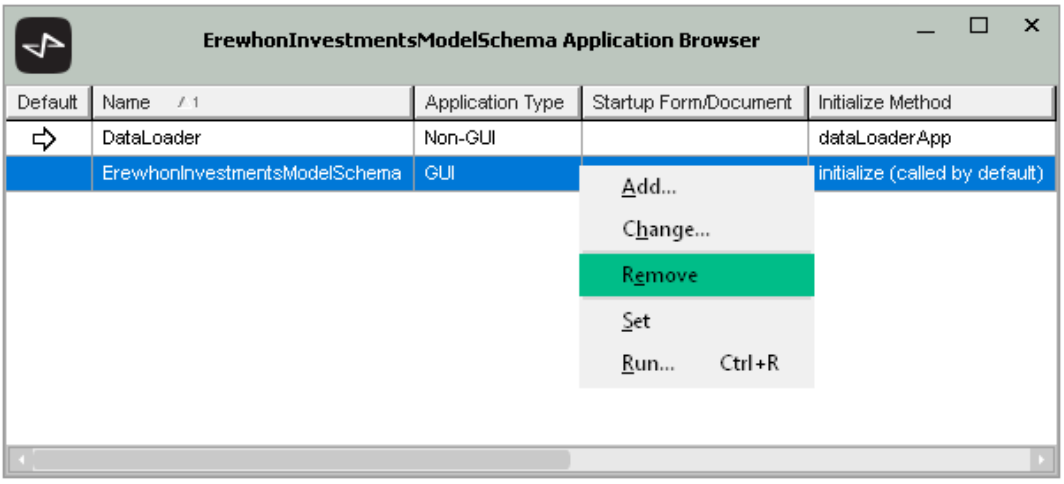
4. The results of the validation are written to the Jade Interpreter Output Viewer. The following errors should be identified.

```
=======================
Validating Applications
=======================
Application ErewhonInvestmentsModelSchema does not have a startup form
```

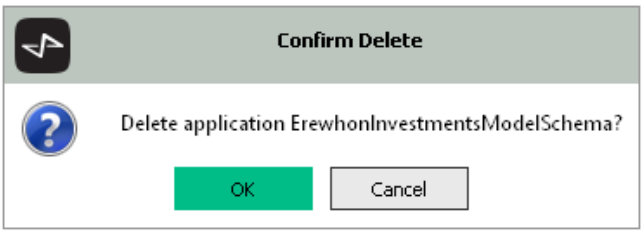
```
=======================
Validating Dictionaries
=======================
Key path does not have inverse SaleItem::codePrefix - OrderProxy::mySaleItem.SaleItem::codePrefix
Key path does not have inverse SaleItem::codeNumber - OrderProxy::mySaleItem.SaleItem::codeNumber
```

5. Select **ErewhonInvestmentsModelSchema** in the Schema Browser and then use the Ctrl+L shortcut keys to open the Application Browser.

6. Select the **ErewhonInvestmentsModelSchema** application in the Application Browser, right click on it, and then select the **Remove** command.



7. Click **OK** in the Confirm Delete message box.

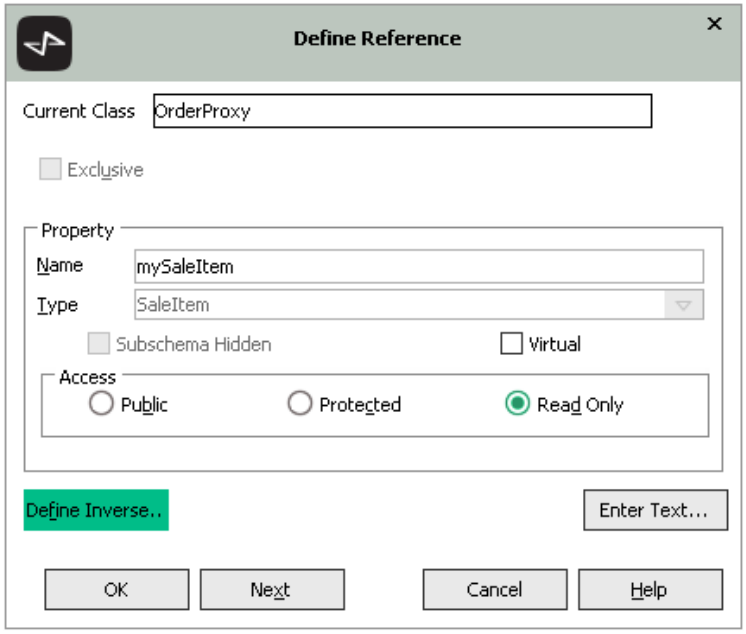


Note The **ErewhonInvestmentsModelSchema** application is the default application that was automatically generated when the **ErewhonInvestmentsModelSchema** schema was created. As it is not used for anything, it can be deleted.

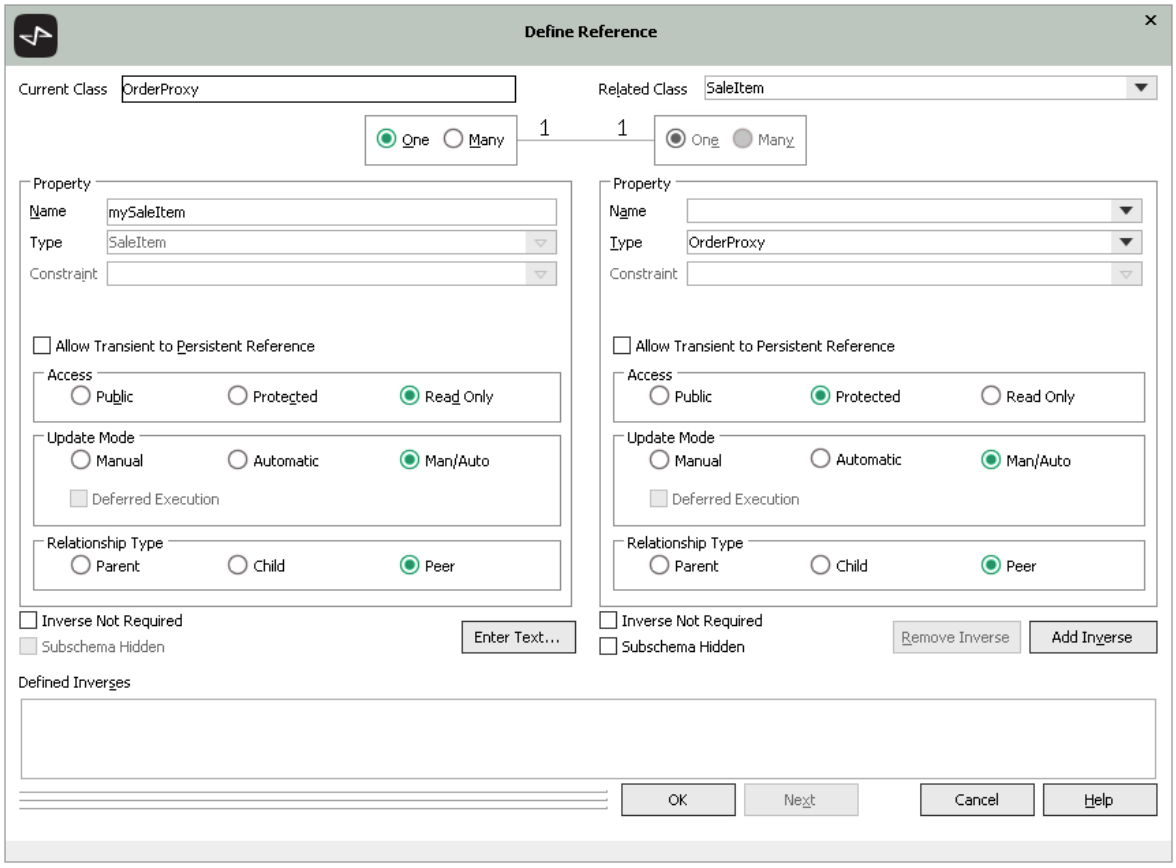
8. Open the **ErewhonInvestmentsModelSchema** schema in a Class Browser. (Press Ctrl+B from the Schema Browser.)

9. Navigate to the **OrderProxy** class (which is a subclass of **ModelTransient**).

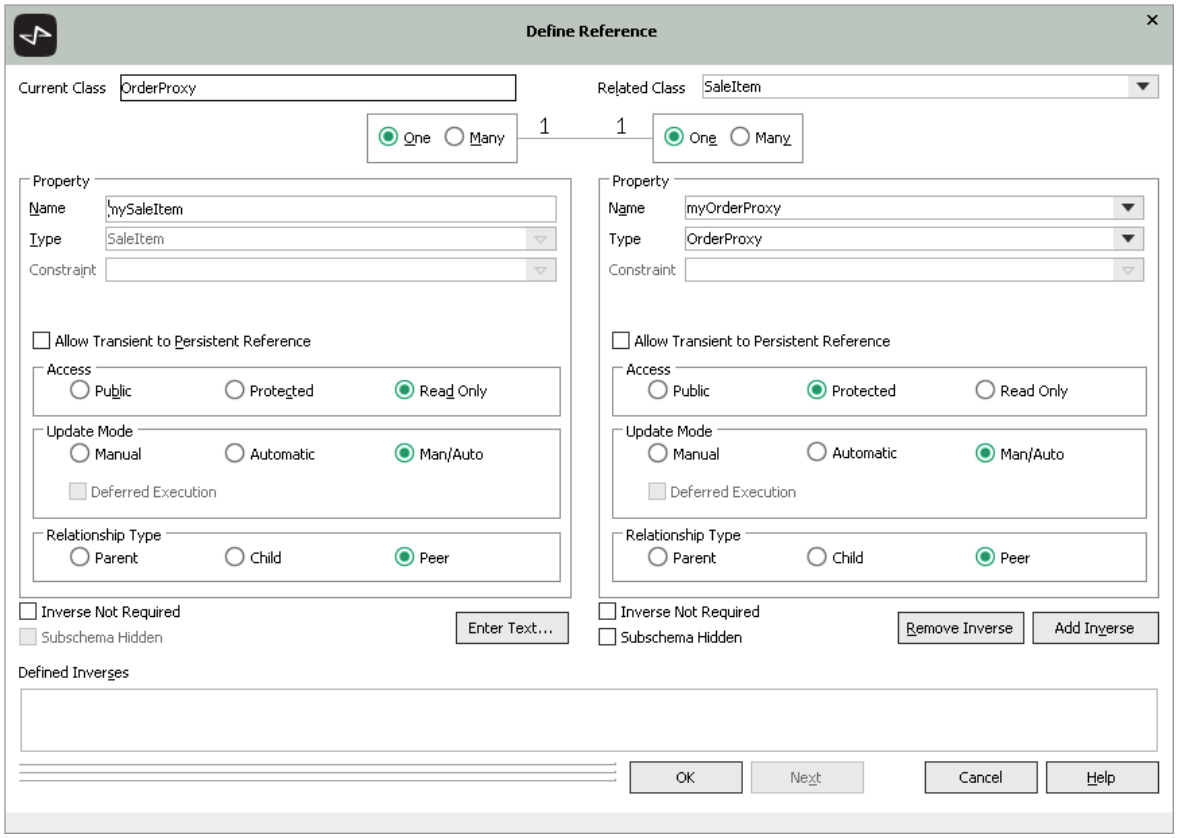
10. Right-click on the **mySaleItem** reference and then select the **Change** command.



11. Click **Define Inverse**.



12. To the **SaleItem** class (in the Property group box at the right of the dialog), set the **Name** value to **myOrderProxy**.



13. Click **OK**.

14. Re-run the **Validate Schema** command on **ErewhonInvestmentsModelSchema**.

You should see that there are no longer any errors.

.NET Exposures

The Jade Platform provides for integration with Microsoft .NET systems through use of the **JoobContext** object.

In this module, the **Banking** system that is built in the Jade Platform Developer's course is exposed to .NET, allowing for a WPF front-end application to display and add to the Jade back-end database.

As the **Banking** system was designed with a separation between the model and the view, the view can be substituted with a completely different view without the need for any change to the model code.

DotNetConnection Application

For Microsoft .NET to use Jade classes, it maintains a reference to a **JoobContext** object that is generated by Jade and it provides an interface to the Jade classes by delegating requests to the appropriate Jade classes.

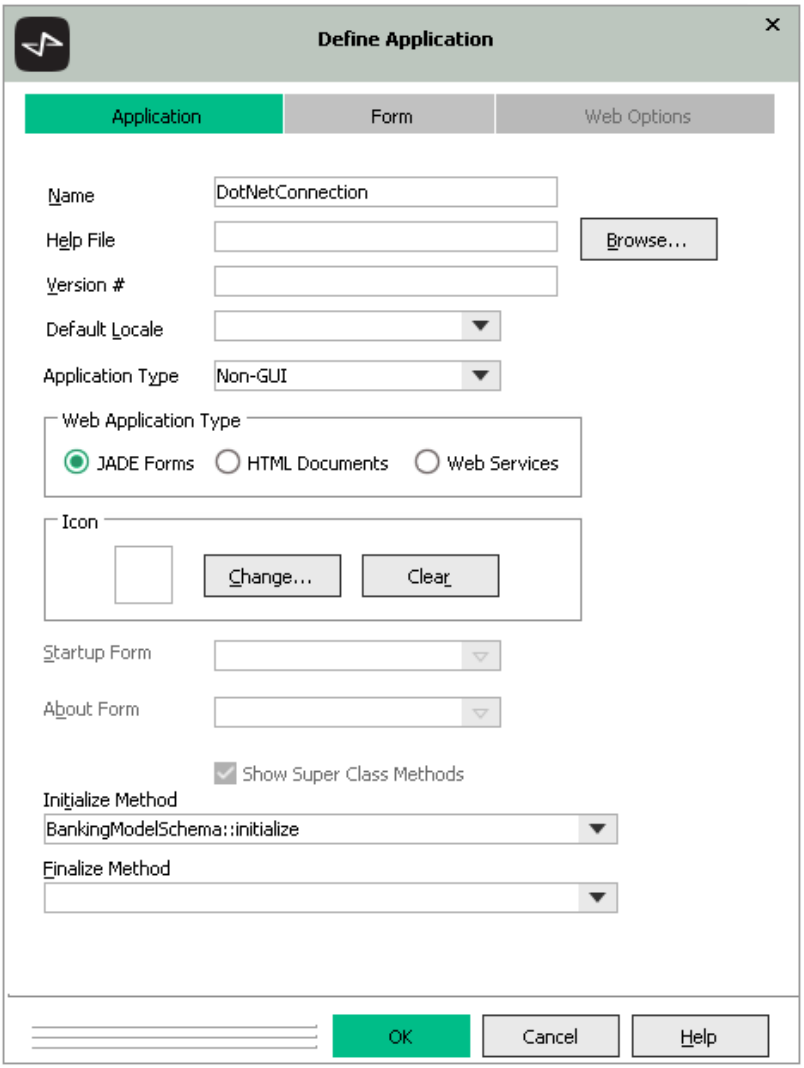
As the **JoobContext** object needs to have a dedicated process to be able to read properties or run methods of Jade classes, we use a non-GUI application called **DotNetConnection** to generate this process.

Exercise 1 – Creating a DotNetConnection Application

In this exercise, create the **DotNetConnection** application that the **JoobContext** object will use to delegate requests from .NET to Jade.

1. Select **BankingViewSchema** in the Schema Browser and then open the Application Browser by pressing Ctrl+L or clicking on the **Browse Applications** icon in the toolbar.

2. Select the **Add** command from the Application menu and then fill out the **Application** sheet of the Define Application dialog as follows.



   a. Enter **DotNetConnection** in the **Name** text box.

   b. Select **Non-GUI** in the **Application Type** drop-down list box.

   c. Check the **Show Super Class Methods** check box, so that you can set the initialize method to **BankingModelSchema::initialize** in the **Initialize Method** combo box.

   d. Click **OK**.

Jade Exposures

Rather than letting external applications have full access to Jade databases, Jade defines an exposure that contains only the properties and methods that the specific external application requires.

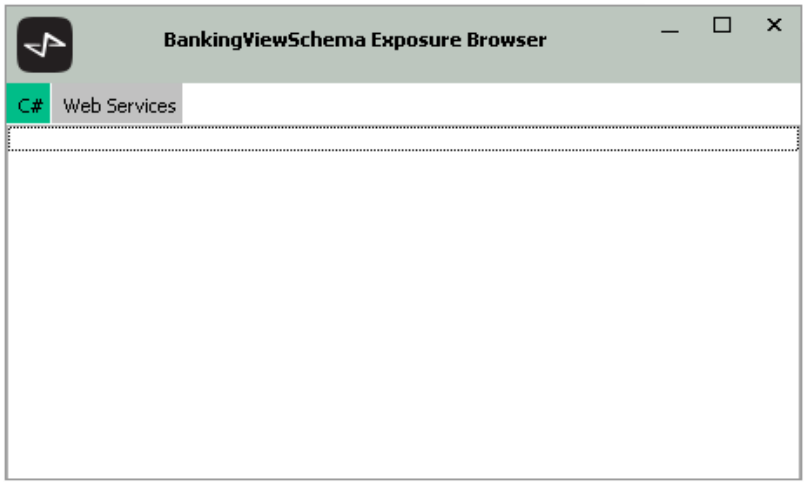
The advantage of this is twofold, as it:

- Provides a more-simple interface to the Jade database
- Allows for greater control over which Jade properties and methods external applications can access

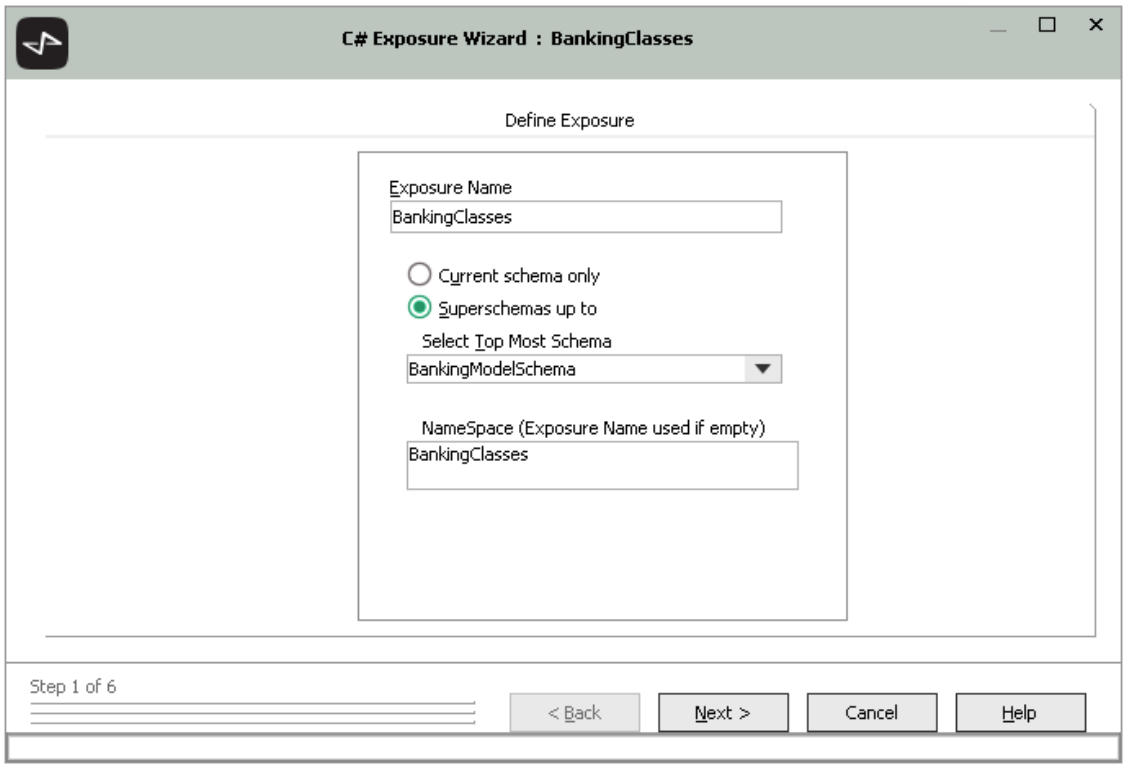
Exercise 2 – Using the Jade Exposure Wizard

In this exercise, define the exposure that the .NET application will access.

1. Select **BankingViewSchema** in the Schema Browser, and then select the **Exposures** command from the Browse menu. Right-click on the Exposure Browser and then select the **Add** command, to create a new C# exposure.



2. Name the exposure **BankingClasses** and then select **BankingModelSchema** option in the **Superschemas up to** drop-down list box.



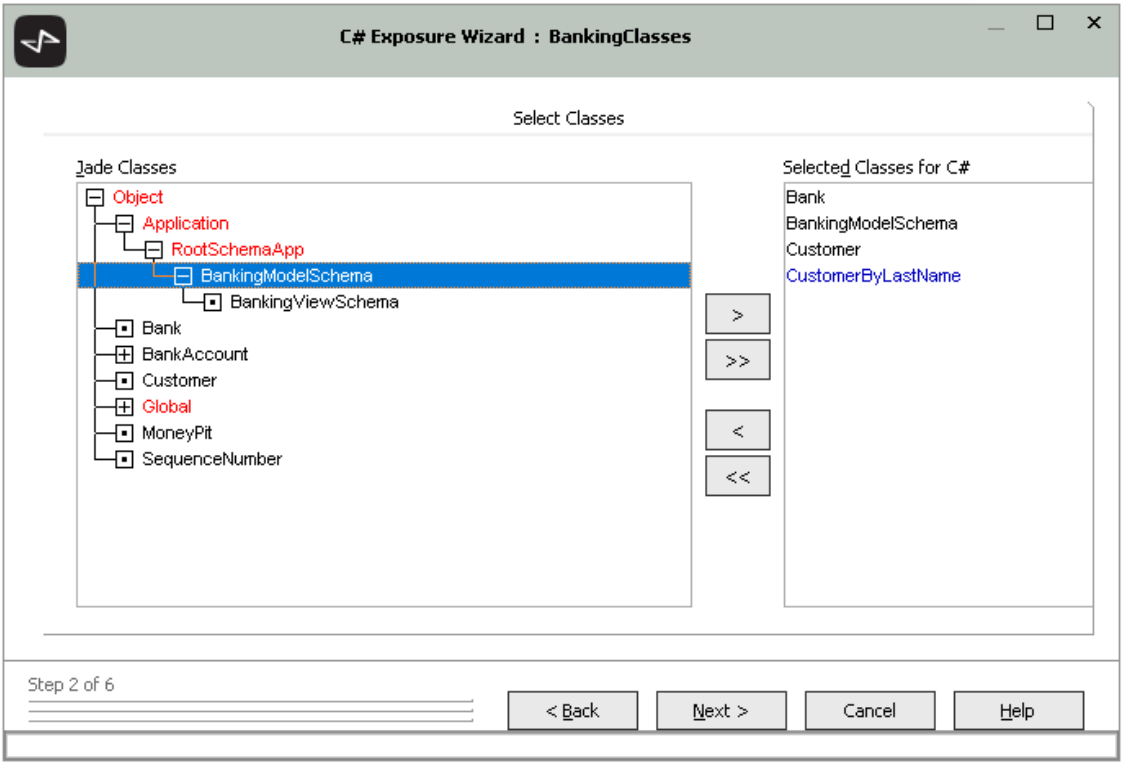
3. Select the following classes.

   a. **Bank**

   b. **BankingModelSchema** (a subclass of the **RootSchemaApp** application class)

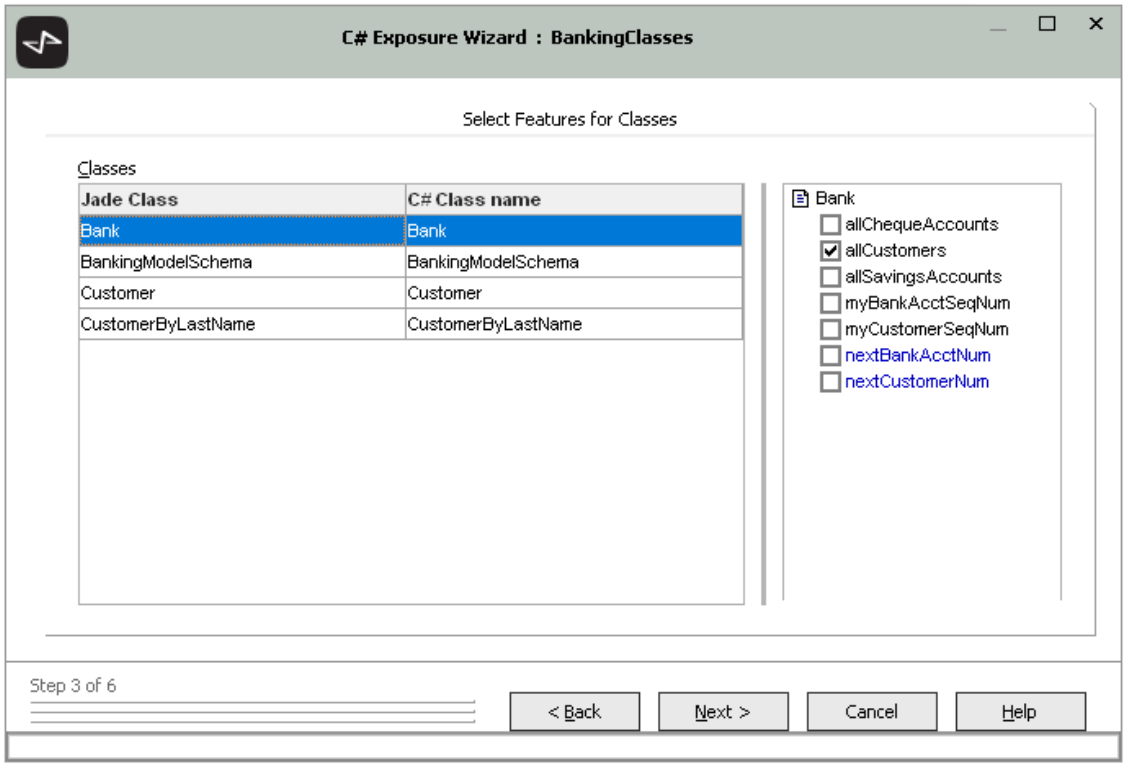
c. **Customer**

These will be the classes that are accessible from .NET.

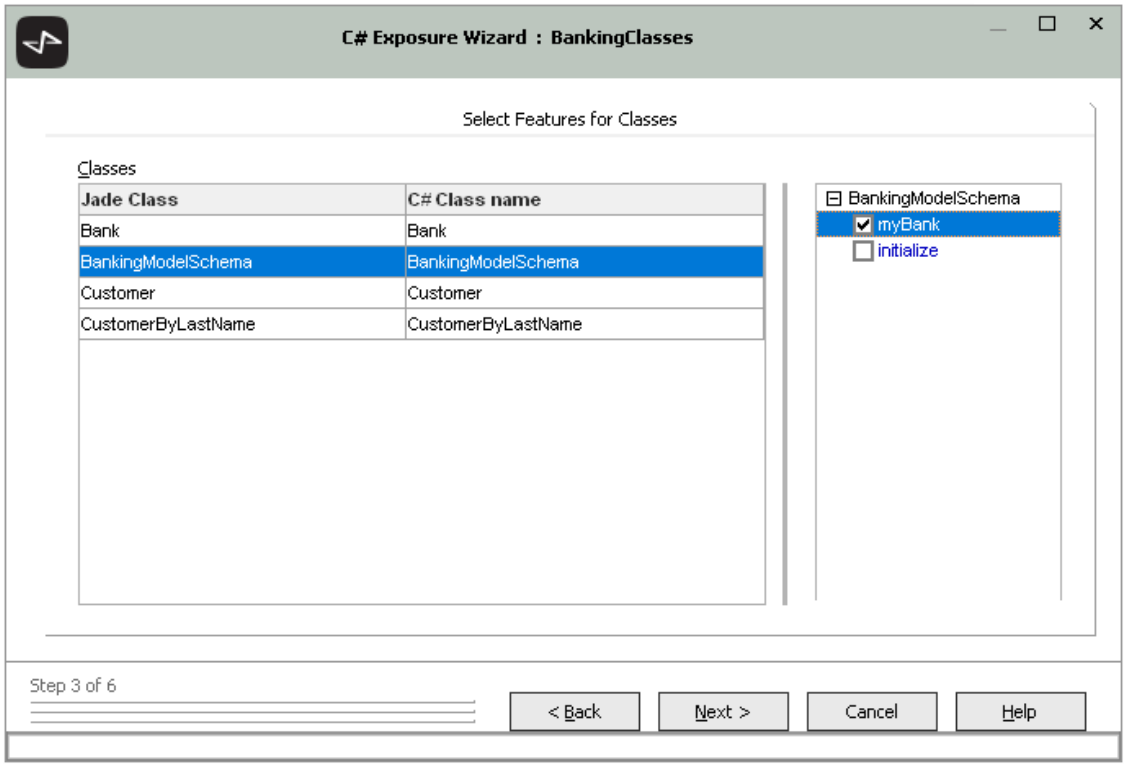


Note Selecting only the classes that you intend to use provides a simpler interface to the Jade Platform. You can always add additional classes to the exposure later, if required.

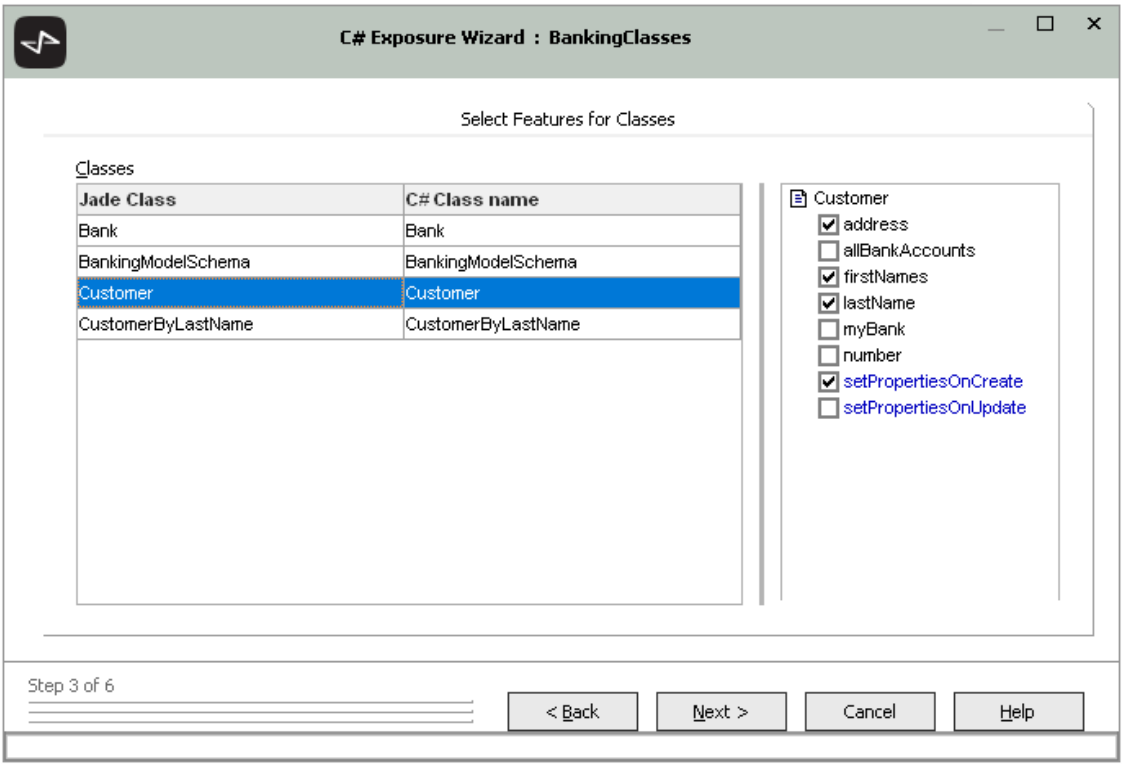
4. Select the **Bank** class and then select (check) the **allCustomers** property from the features on the right of the sheet. This exposes the **allCustomers** collection to .NET; the other methods and properties are *not* exposed.



5. In the **BankingModelSchema** class, check the **myBank** feature.



6. In the **Customer** class, check the **address**, **firstNames**, **lastName**, **number**, and **setPropertiesOnCreate** feature properties.



7. As the steps on sheets 4 and 5 of this wizard do not require any changes, click **Next** until you reach step 6 (that is, the **Generate** sheet).

   Fill out this sheet, as follows.

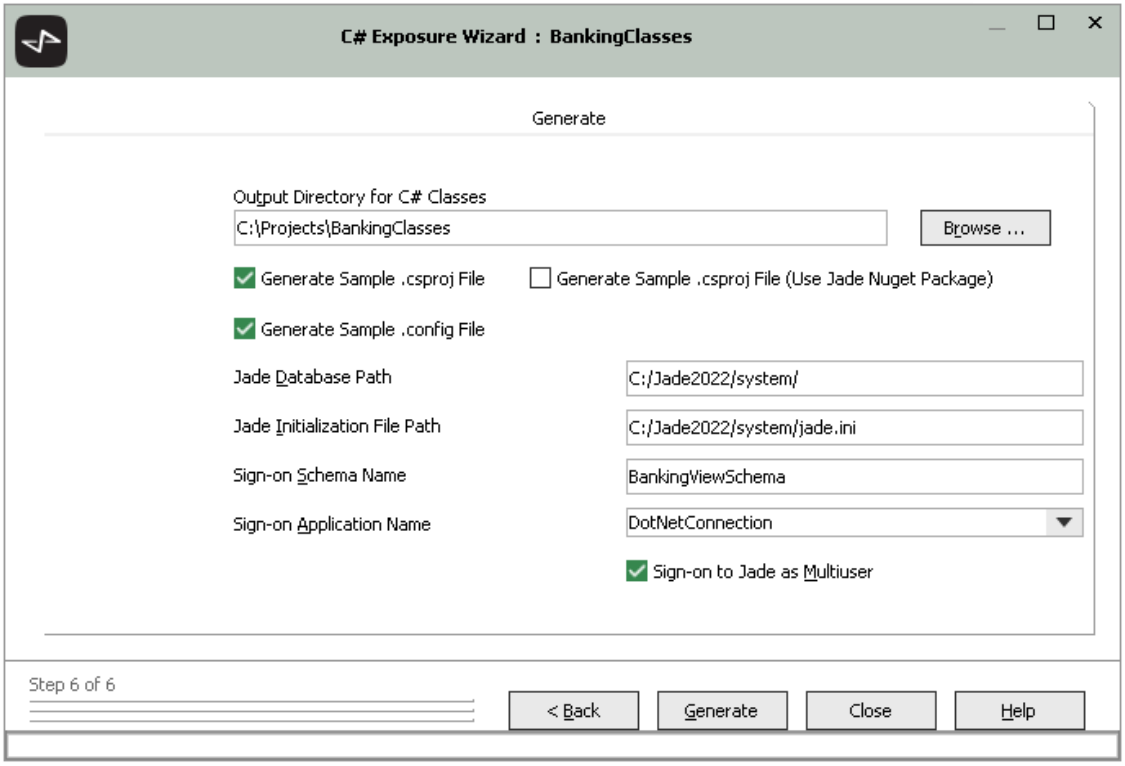
   a. Enter **C:\Projects\BankingClasses** in the **Output Directory** text box. This is the directory where the C# project file and class files will be created.

   b. Check the **Generate Sample .csproj File** and **Generate Sample .config File** check boxes.

   c. Enter the absolute path of the Jade database directory (including the **system** folder) in the **Jade Database Path** text box.

   d. Enter the location and file name of your Jade initialization file (which is probably called **jade.ini**) in the **Jade Initialization File Path** text box.

e. Check the **Sign-on to Jade as Multiuser** check box.



8. Click **Generate**.

The wizard does not close, but *Generation completed* should be displayed at the lower left of the wizard. You can now click **Close**.

Dynamic Link Libraries (*.dll Files)

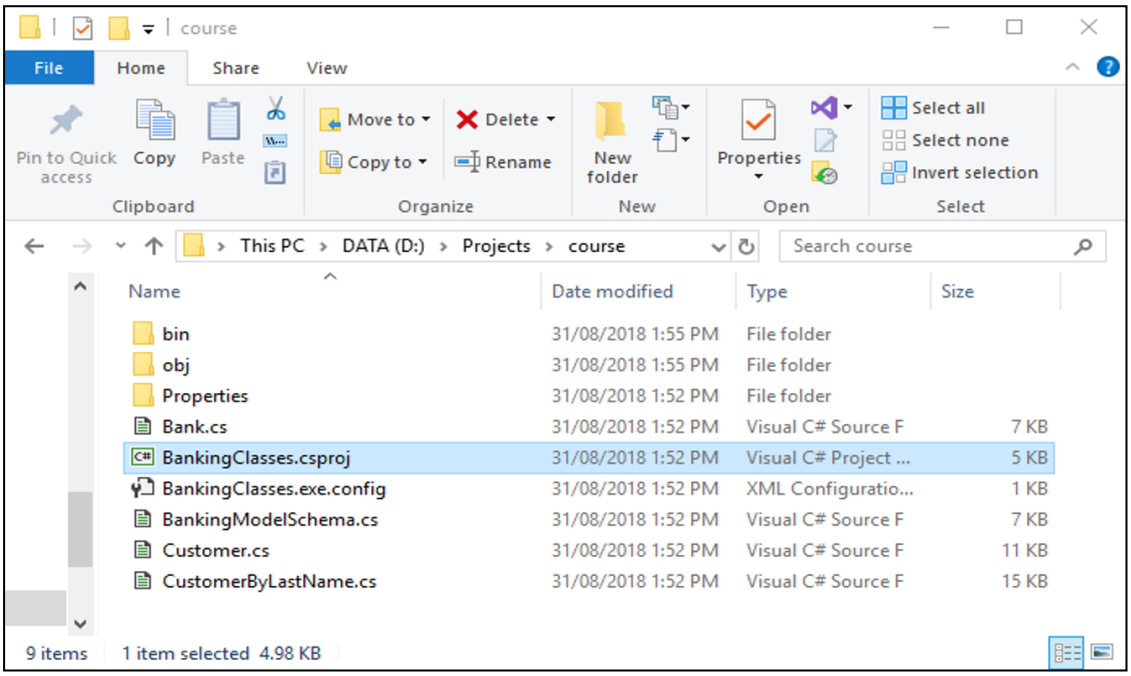
A Dynamic Link Library is an external method (or routine) written in any language that can create a Dynamic Link Library (DLL). DLL methods can then be executed as standard Jade methods from within the Jade Platform.

The purpose of creating DLLs is to allow different environments to integrate; for example, allowing an application written in C# to use Jade methods.

Exercise 3 – Creating a DLL File

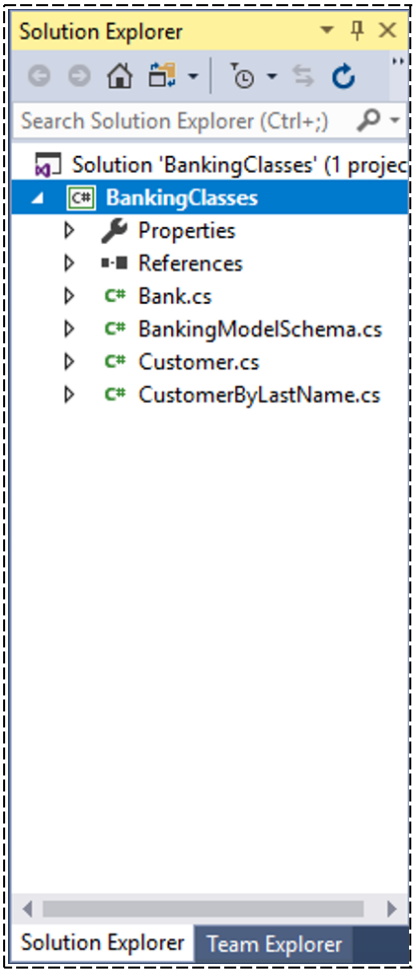
In this exercise, use the generated C# project to create the **BankingClasses.dll** file.

1. In the File Explorer, navigate to **C:\Projects\BankingClasses**.



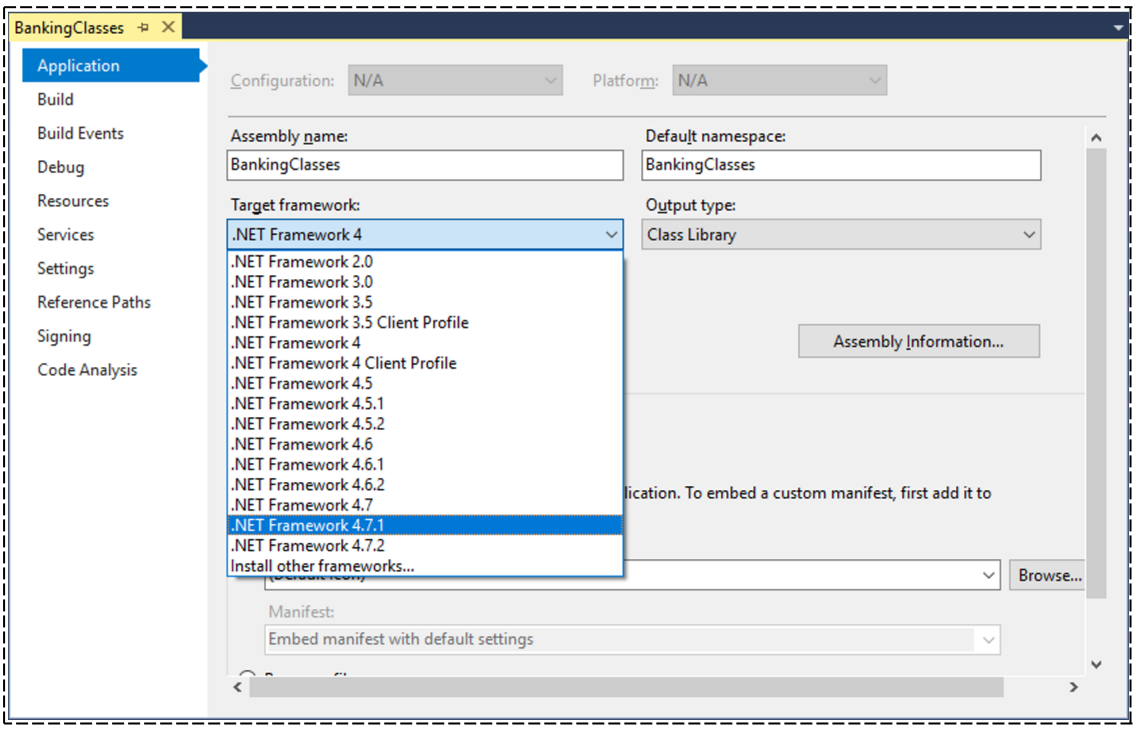
2. Open **BankingClasses.csproj** in Microsoft Visual Studio 2017.

Find the Solution Explorer and then select **BankingClasses** under Solution 'BankingClasses'.

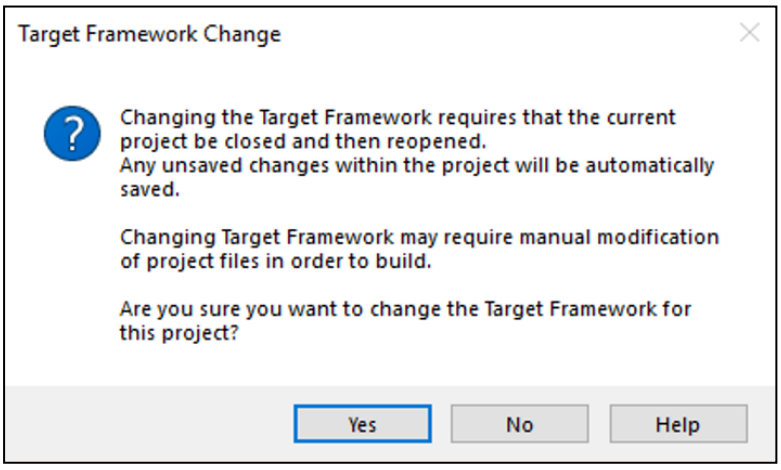


3. Press Alt+Enter to open the Properties menu.

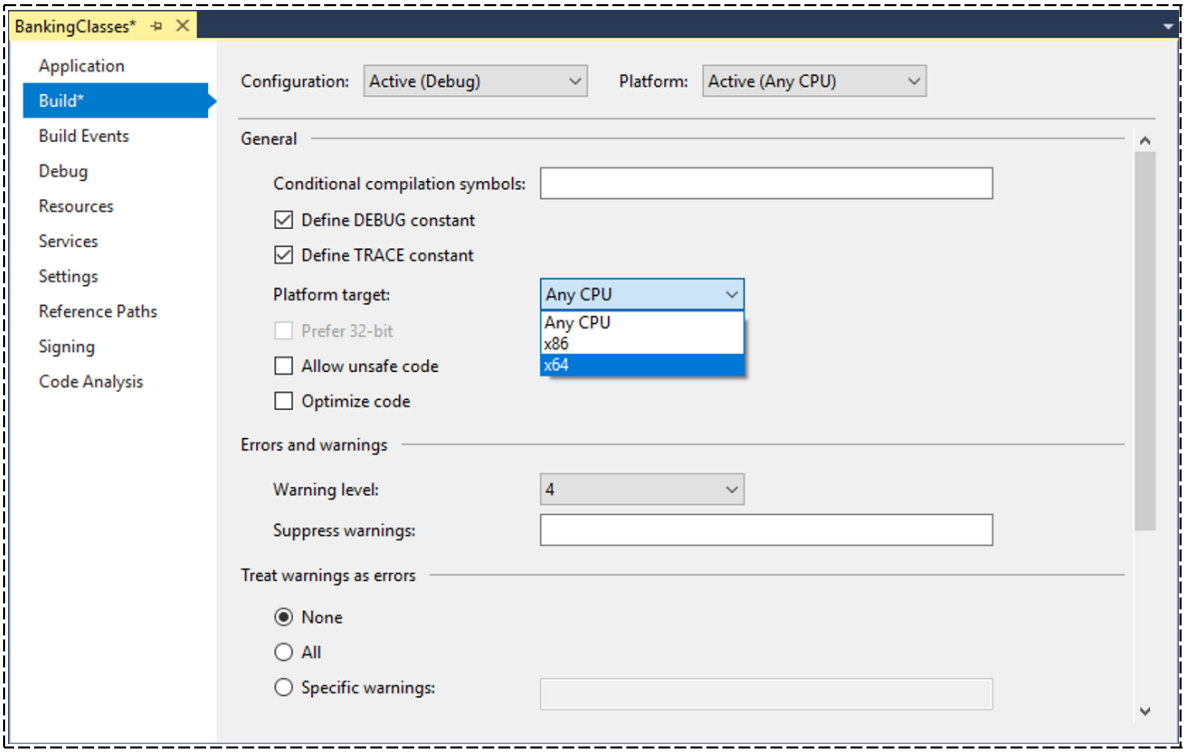
   Change the Target framework to **.NET Framework 4.7.1**.



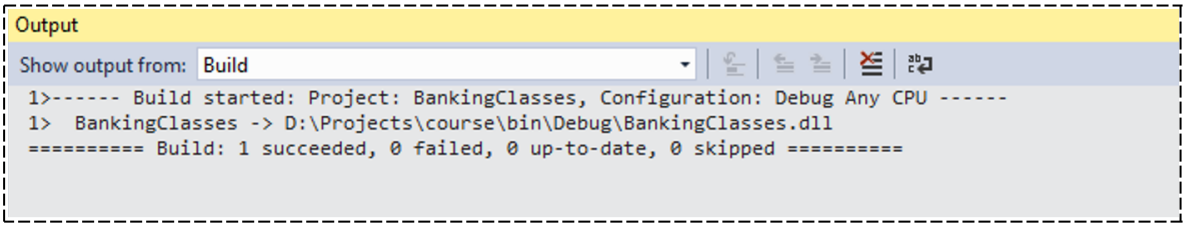
4. In the following message box that is then displayed, click **Yes**.



5. Click the **Build** tab, and then change the platform target from **Any CPU** to **x64**.



6. Build the project by pressing Ctrl+Shift+B. You should then see a message like the following displayed in the Output window.

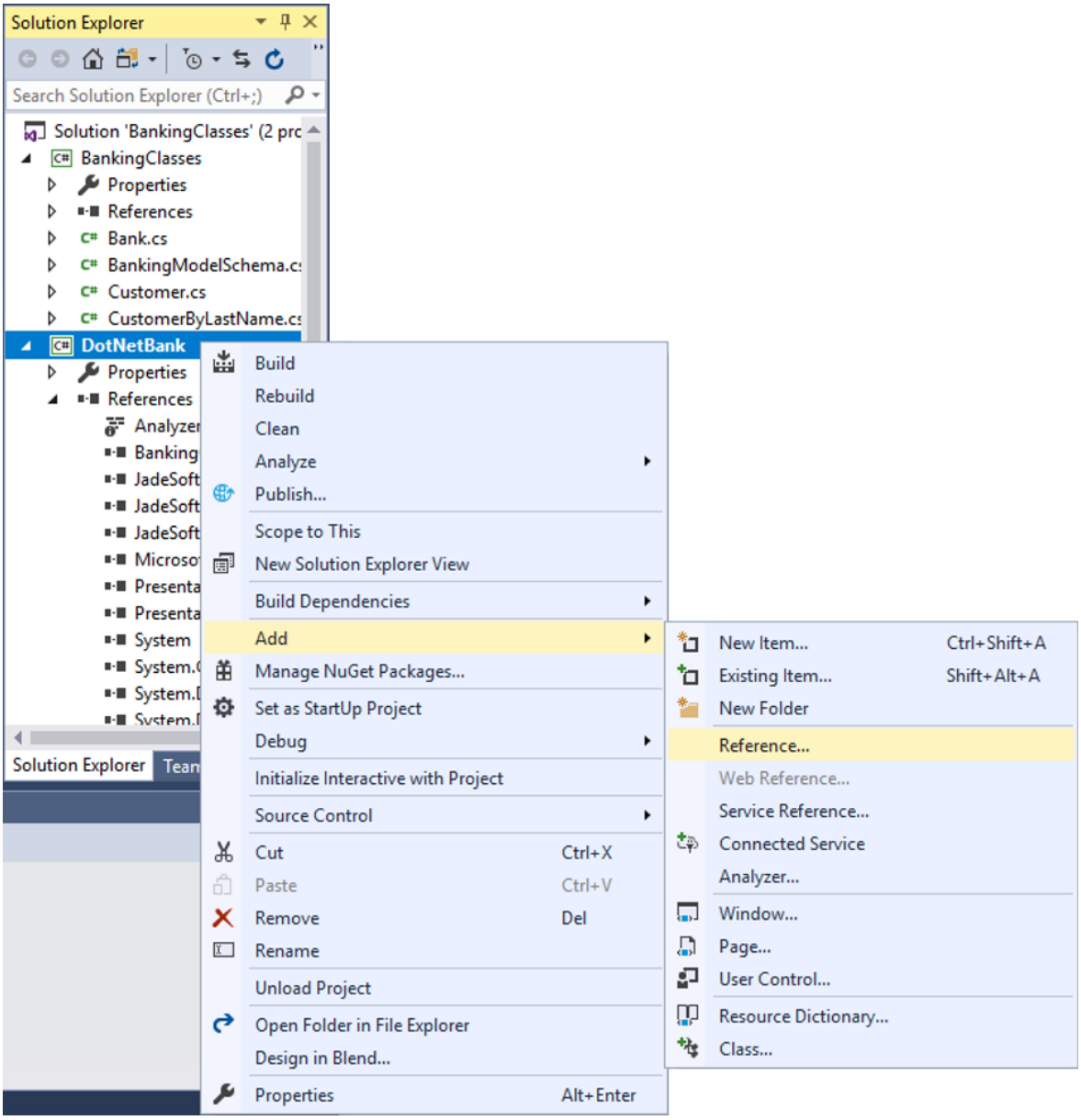


Building this project with the correct settings creates a **.dll** file for the Jade exposure defined in Exercise 2 of this module.

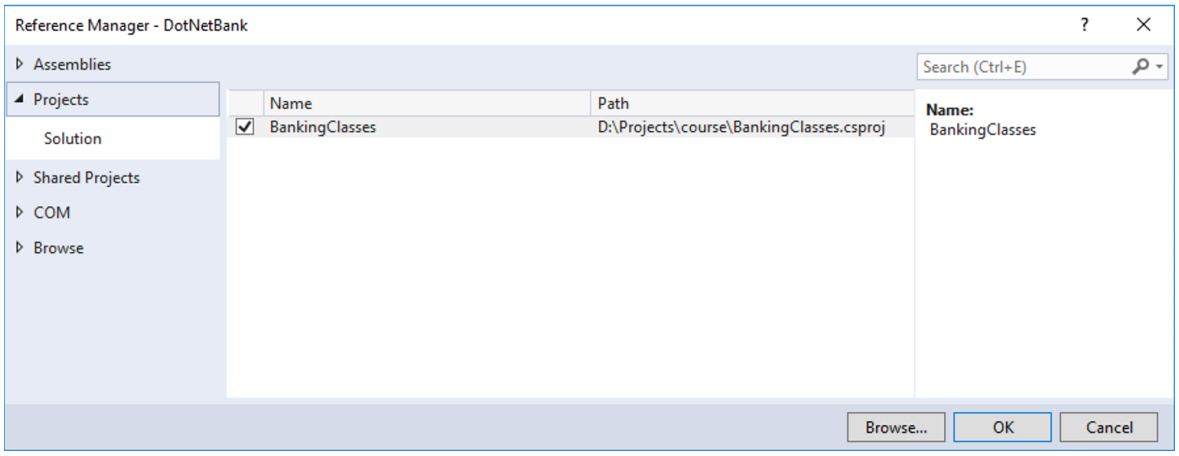
Exercise 4 – Importing Jade DLL Files into .NET

In this exercise, set the required DLL files as references in a .NET application.

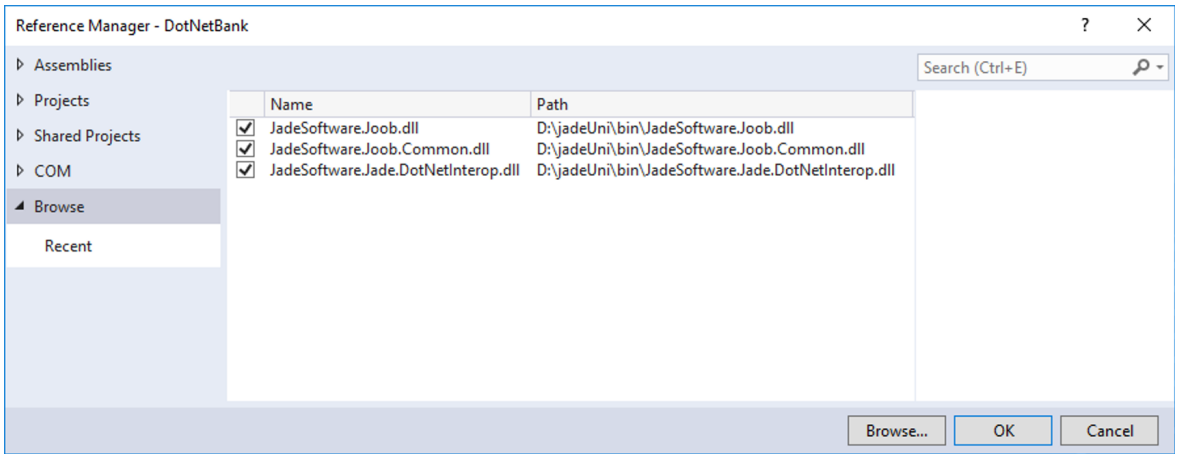
1. Add a new project called **DotNetBank** to the Solution 'BankingClasses' in the Solution Explorer.
2. Add a new reference to the **DotNetBank** project by right-clicking on the **DotNetBank** project and then selecting the **Reference** command in the Add menu.



3. Check **BankingClasses** in the **Projects** tab.



4. In the **Browse** tab, click the **Browse** button and then add the following three references.



5. Navigate to **C:\Projects\BankingClasses** in the File Explorer and then open **BankingClasses.exe.config** in Visual Studio.

6. Copy the contents of **BankingClasses.exe**.config and then paste it into **app.config**.

App.config* | BankingClasses.exe.config | MainWindow.xaml | MainWindow.xaml.cs*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="joob" type="JadeSoftware.Joob.Configuration.JoobConfigurationSection, JadeSoftware.Joob" />
  </configSections>

  <connectionStrings>
    <add name="myDefault" providerName="JadeSoftware.Joob.JoobConnection" connectionString="DataSource=D:/jadeUni
  </connectionStrings>

  <joob defaultConnection="myDefault">
    <installation directory="D:\jadeUni\bin" />
  </joob>

  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
</configuration>
```

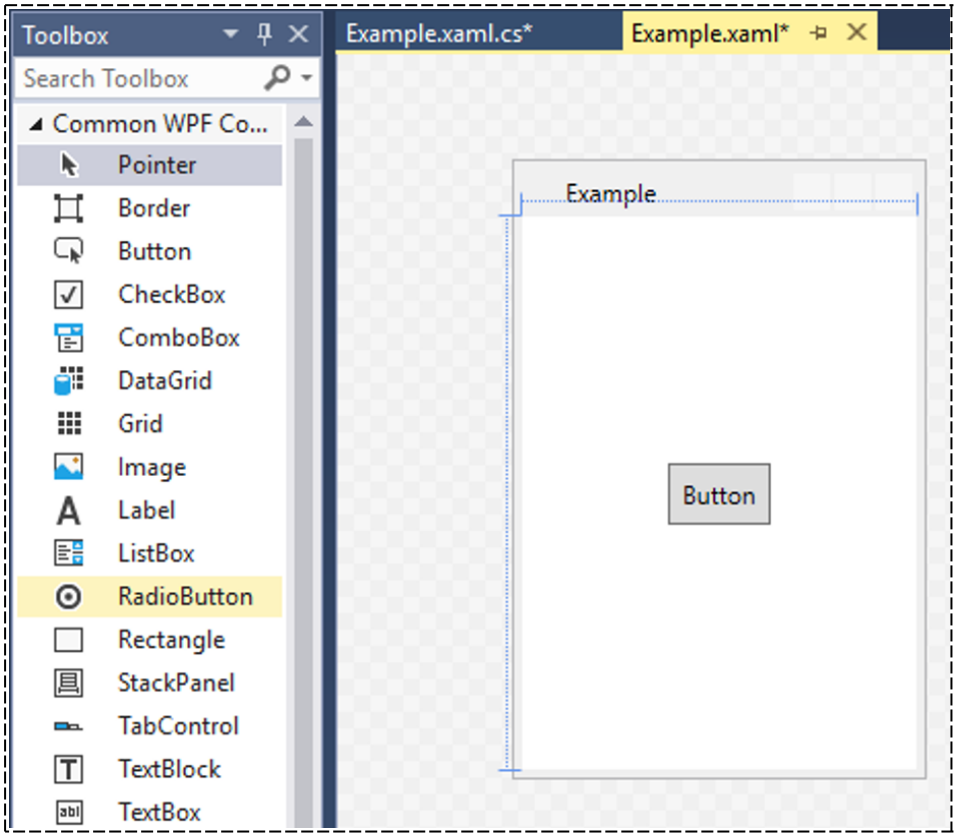
7. Ensure that the **Application** and **Build** settings are set to the same as those of the **BankingClasses** application; that is:

   - Target Framework is set to **.NET Framework 4.7.1** in the **Application** tab of the project properties.
   - Platform Target is set to **x64** in the **Build** tab of the project properties.

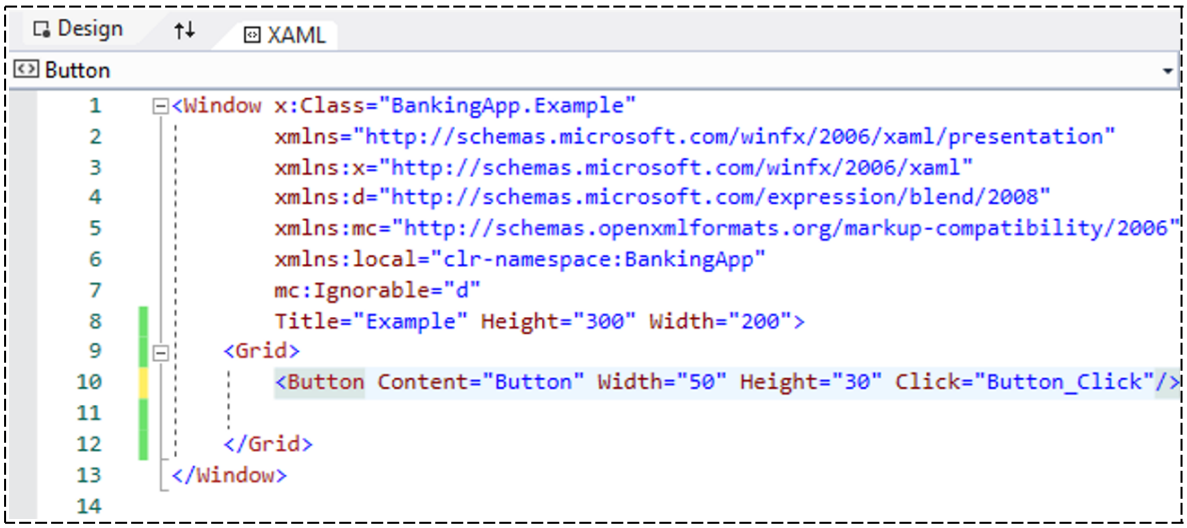
C#, XAML, and the Windows Presentation Framework

When developing .NET applications, one typically uses the Windows Presentation Framework (WPF), which uses eXtensible Application Markup Language (XAML, pronounced "Zammel") to define the presentation of the application and the C# programming language to define the logic of the application.

XAML can be generated in Visual Studio by using the Designer - a What-You-See-Is-What-You-Get (WYSIWYG) editor.



XAML elements can also be created by manually typing in the required markup.



C# code is used to define the logic of the form; for example, the behavior when a button is clicked.

```
Example.xaml.cs  Example.xaml*
BankingApp                                  BankingApp.Example
 1  using System.Windows;
 2
 3  namespace BankingApp
 4  {
 5      /// <summary> Interaction logic for Example.xaml
 8      public partial class Example : Window
 9      {
10          public Example()
11          {
12              InitializeComponent();
13          }
14
15          private void Button_Click(object sender, RoutedEventArgs e)
16          {
17              this.Close();
18          }
19      }
20  }
21
```

Exercise 5 – Creating a WPF Application

In this exercise, create a WPF application that will provide a .NET interface to the Jade **Banking** system exposure. At first, you will simply display all customers of the bank in a list box.

1. Create a **ListBox** control in the center of the window using the Designer and then add **Name="lbCustomers"** to the **ListBox** element in the XAML.

```
Design  ↑↓  XAML
ListBox
 1  <Window x:Class="DotNetBank.MainWindow"
 2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
 5          xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
 6          xmlns:local="clr-namespace:DotNetBank"
 7          mc:Ignorable="d"
 8          Title="MainWindow" Height="450" Width="800">
 9      <Grid>
10          <ListBox Name="lbCustomers" Height="200" Width="300"/>
11      </Grid>
12  </Window>
```

2. Switch to Code View by pressing F7 and then add the following **using** statements.

```
using System.Windows;
using BankingClasses;
using JadeSoftware.Joob;
using JadeSoftware.Joob.Client;
using JadeSoftware.Joob.Exceptions;
using JadeSoftware.Jade.DotNetInterop;
```

Note There may be superfluous using statements when the code is first opened. To remove them, select any **using** statement, press Alt+Enter, and then click **Remove Unnecessary Usings**.

3. Add the following code. This will create a reference called **bank**, which through the **JoobContext** object, refers to the **Bank** class defined in Jade.

```
using System.Windows;
using BankingClasses;
using JadeSoftware.Joob;
using JadeSoftware.Joob.Client;
using JadeSoftware.Joob.Exceptions;
using JadeSoftware.Jade.DotNetInterop;

namespace DotNetBank
{
    public partial class MainWindow : Window
    {
        JoobContext context;    // Declare a variable of type JoobContext (which is an inbuilt JADE class)
        Bank bank;              // Declare a variable of type Bank (Which is one of our classes from JADE)

        public MainWindow()     // Constructor, is like a JADE create method.
        {
            InitializeComponent();  // Creates the window described in XAML
            context = new JoobContext();    // Instantiates the JoobContext, running the DotNetConnection application
                                            // that we made in JADE
            bank = context.FirstInstance<Bank>();   // Similar to Class.FirstInstance in Jade - gets the first Bank/
                                                    // Remember, Bank is a singleton so there is always only one.
        }
    }
}
```

Note The comments are for illustrative purposes and can be omitted.

4. To populate the **lbCustomers** list box with the customers of the bank, add the following to the **MainWindow** constructor method.

```
            lbCustomers.ItemsSource = bank.AllCustomers;    // Listboxes display a collection, so we can just set
                                                            // the ItemsSource property to the allCustomers collection

            lbCustomers.DisplayMemberPath = "LastName";     // Uses the LastName property of the Customer as the label
                                                            // in the ListBox.
        }
    }
}
```

5. Press F5 to run the application. You should now see the last names of the bank's customers displayed in the list box.

Internationalization

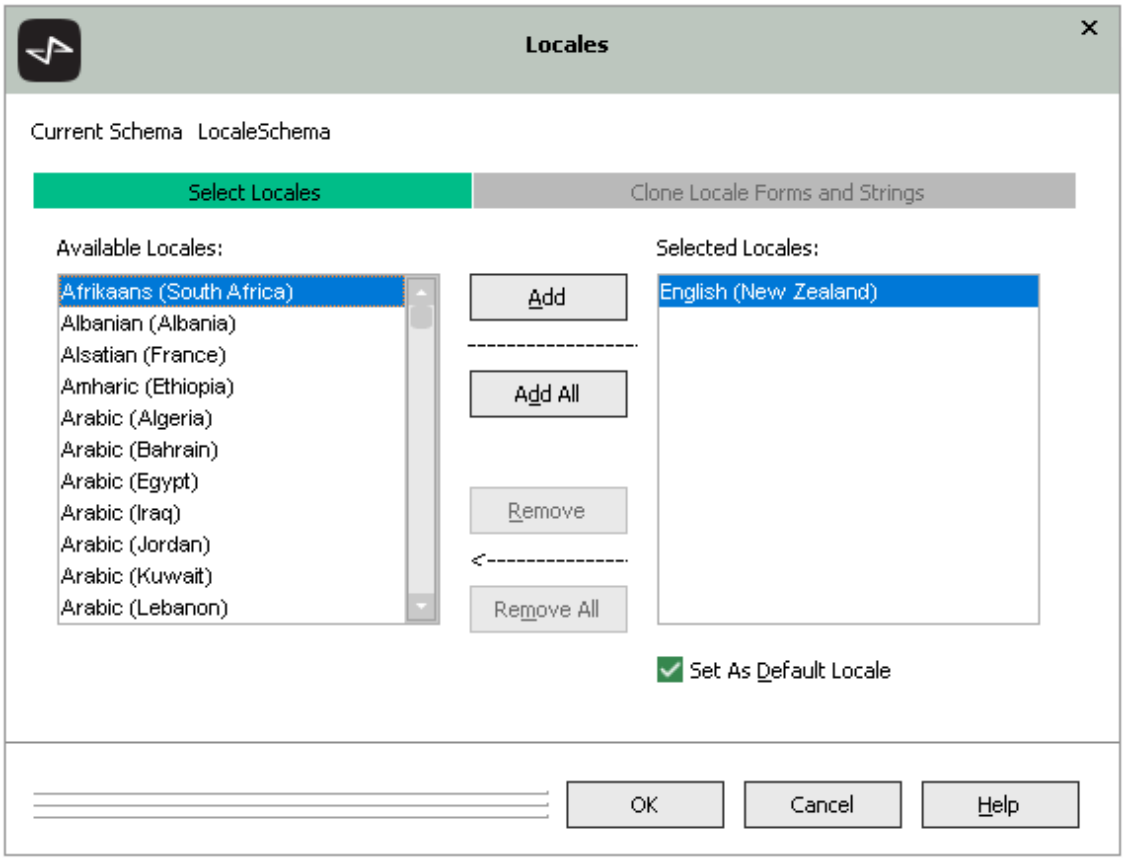
The Jade Platform provides for the ability to internationalize Jade systems to be compatible with a wide variety of languages and formats. By using translatable strings, you can specify translations for any text in any number of different languages.

The Formats Browser allows for the specification of date, currency, and number formats (which may differ, between locales).

Locales

In Jade, a locale refers to a language and the formatting considerations associated with the use of that language. For example, while the language of **English (New Zealand)** and **English (United States)** is similar (other than a few minor spelling differences), a New Zealand user will likely be used to a date format of **dd/mm/yyyy** while an American user might prefer **mm/dd/yyyy**. Having separate locales for New Zealand and the United States therefore could allow for all dates to be presented in the appropriate format for each country.

Use the **Locales** command in the Schema menu to specify which locales are to be supported in a specific schema.



When running a Jade application, forms are displayed in the default locale specified by the operating system. You can use the **Application** class **setJadeLocale** method to explicitly set the locale; for example, on a form that allows the user to select their language from the form itself. To use this method, you must provide it with the Language Code Identifier (LCID) number of the target locale.

To find the appropriate LCID, each locale has a unique LCID number and a corresponding global constant. For example, the New Zealand English locale has the LCID 5129 and the **JadeLocaleIdNumbers** category global constant **LCID_English_NewZealand**.

Locales in Multiuser Systems

At run time, Jade applications automatically select the appropriate locale based on the client's operating system. For single user and fat client systems, the presentation locale always matches the application locale. However, more consideration must be taken when Jade is running in thin client mode.

The **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the Jade initialization file determines whether the locale settings are synchronized between the standard client (application server) and the thin client.

When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false**:

- Application server regional overrides are suppressed on the thin client; that is, the **Application** class **setJadeLocale** method no longer has any effect on the locale for the thin client.
- The two-digit mask year of **yy** is calculated using the current century. For example, **99** becomes **2099** rather than **1999**.

When the **EnhancedLocaleSupport** parameter is set to **true**:

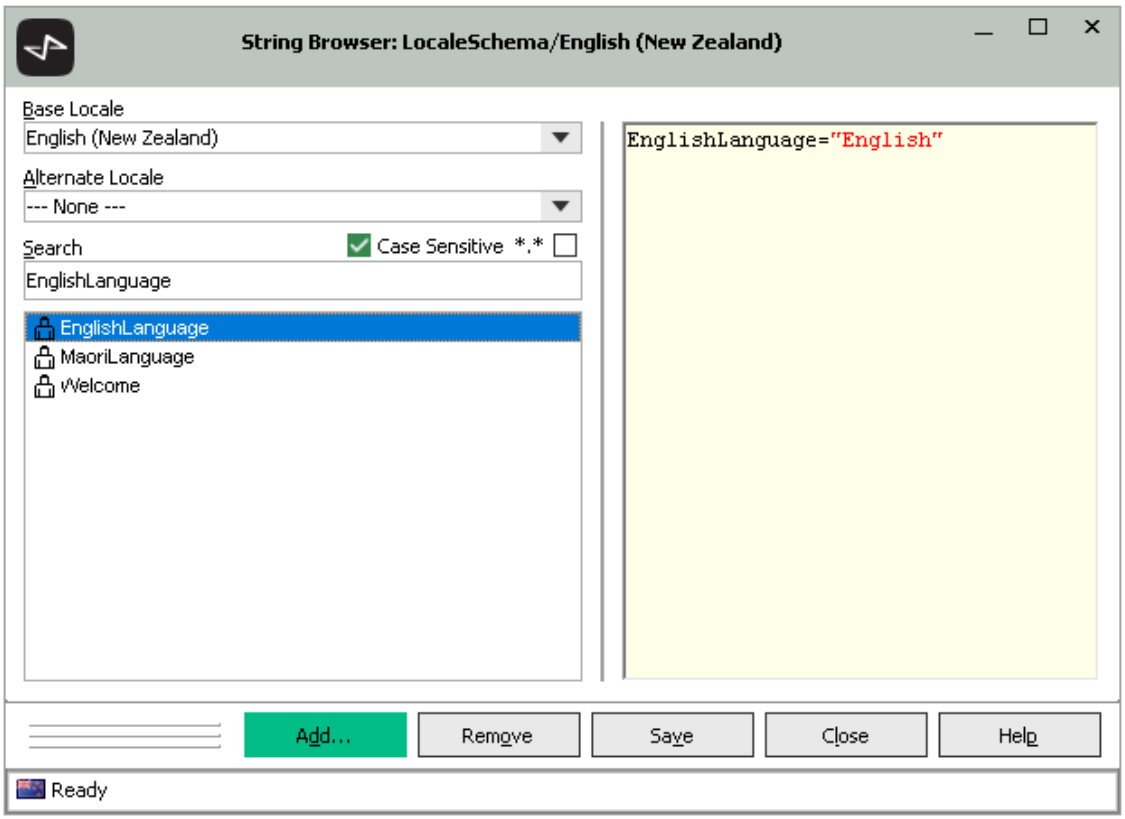
- The required language or languages must be installed on both the application server and the thin client.
- Regional overrides are applied; that is, the **Application** class **setJadeLocale** method overrides the operating system locale on the thin client.
- The Windows Control Panel setting is used to convert a two-digit year into a four-digit year for a two-digit edit mask year of **yy**. For example, **99** by default becomes **1999** rather than **2099**.

Translatable Strings

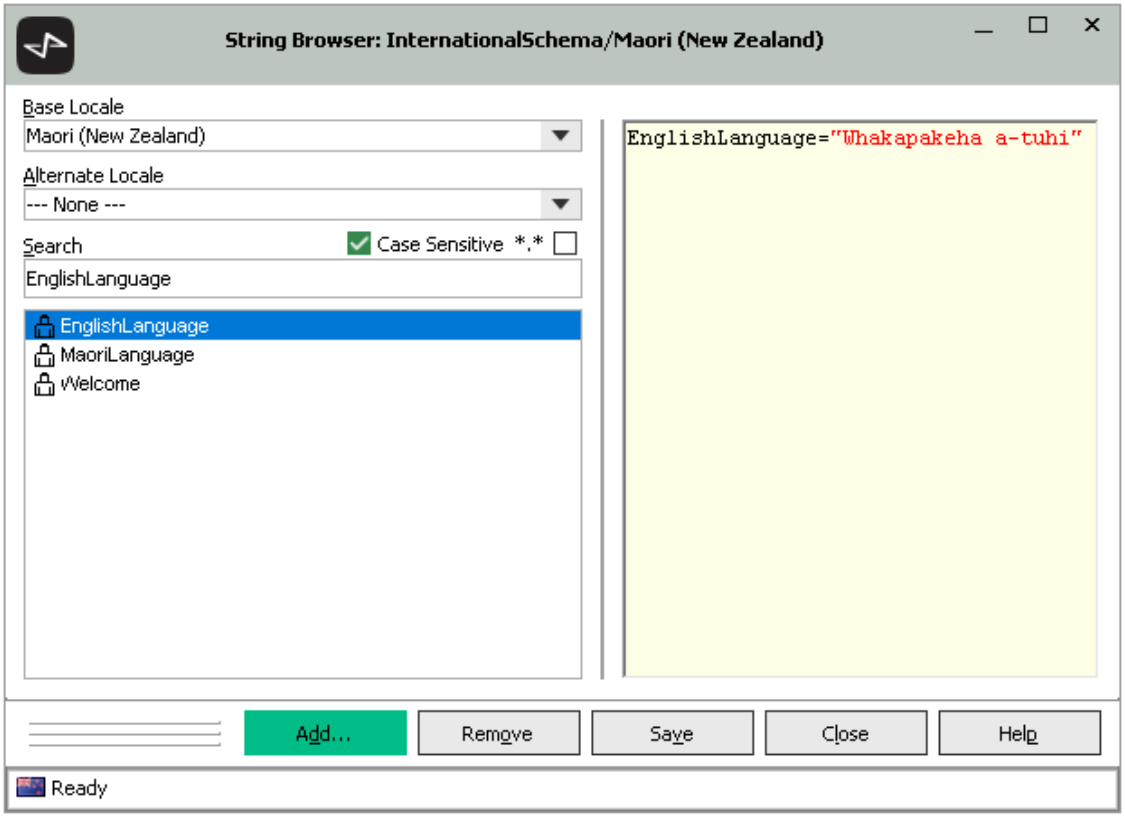
Translatable strings are like global constant strings, but with the important distinction that multiple values can be assigned to the same translatable string: one for each of the locales supported by the schema.

By diligently using translatable strings rather than string literals, it becomes comparatively easy to translate Jade applications between languages and provides the possibility of having a Jade application automatically translate to the user's language based on his or her locale settings.

Translatable strings can be added to a schema from the String Browser by clicking the **Add** button, and defined for each locale by selecting the appropriate locale from the **Base Locale** combo box and then modifying the definition for that locale.



The following example shows the translation from the **EnglishLanguage** translatable string.



When a translatable string has been defined in a schema, it can be added to any Jade method within that schema in place of a string literal, by prefixing the name of the translatable string with a dollar sign (**$**).

When the locale is set to **English (New Zealand)**, the first of the following examples is the same as the second example.

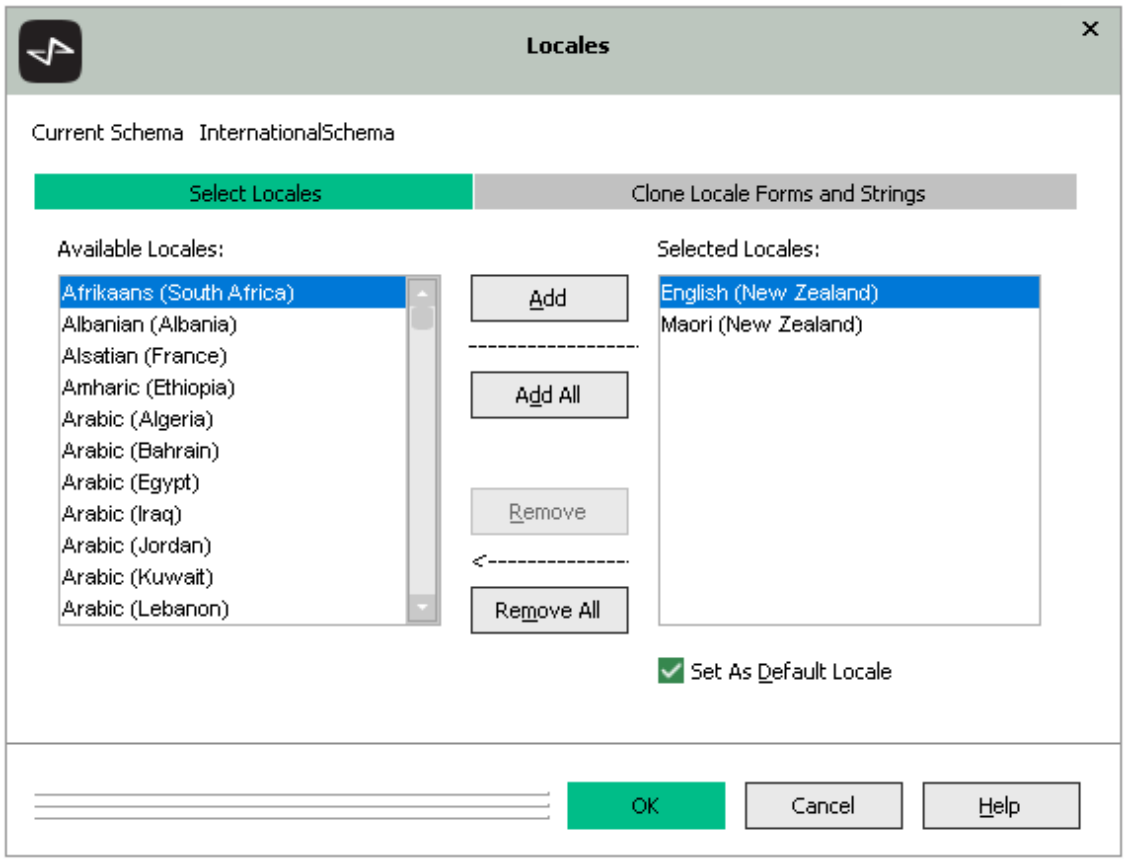
```
translatableStringExample();

begin
    write "A language of New Zealand is " & $EnglishLanguage;
end;
```

```
translatableStringExample();

begin
    write "A language of New Zealand is " & "English";
end;
```

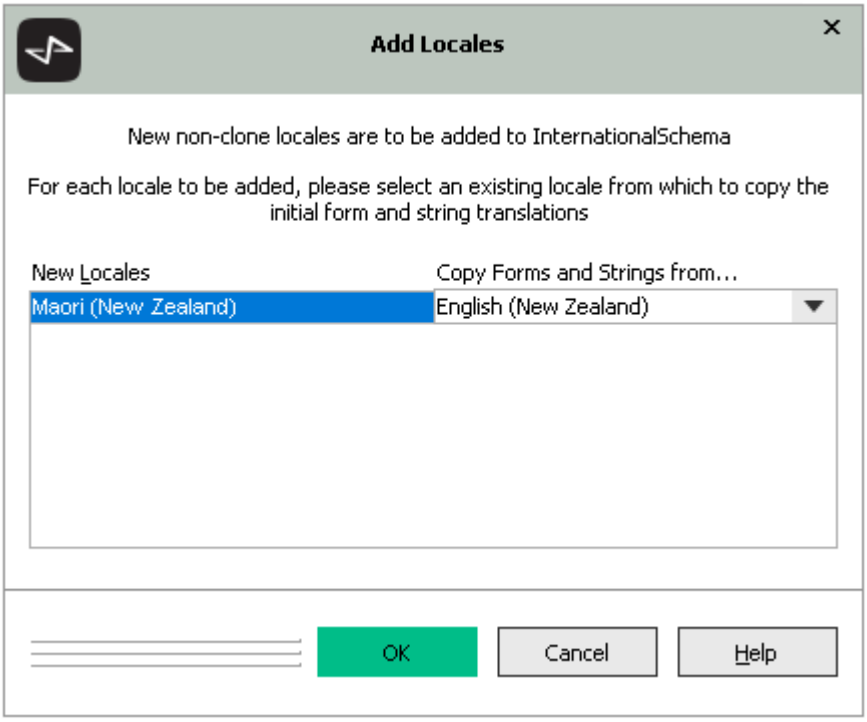
Exercise 1 – Adding a Locale

In this exercise, create a new schema and add a locale for te reo Māori (that is, the Maori language) alongside New Zealand English. You will also design a simple form that is translated to and from te reo Maori in future exercises.

1. Create a schema called **InternationalSchema**.
2. When **InternationalSchema** is selected in the Schema Browser, select the **Locales** command from the Schema menu.
3. From the **Available Locales** list box, select **Maori (New Zealand)**, add it to the **Selected Locales** list box, and then click **OK**.

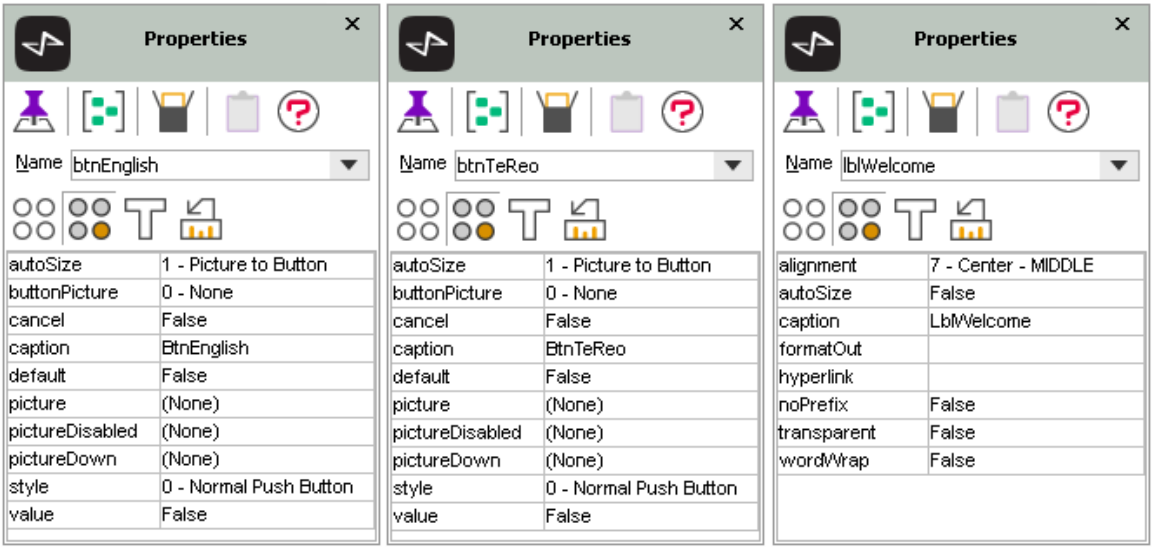


4. On the Add Locales dialog that is then displayed, click **OK**.



5. Open the Jade Painter and create a form called **WelcomeForm**.

6. Design the WelcomeForm as follows.



Note The control captions don't matter, as we will be setting these programmatically (that is, from the Jade code).

7. Save the form.

Exercise 2 – Adding and Using Translatable Strings

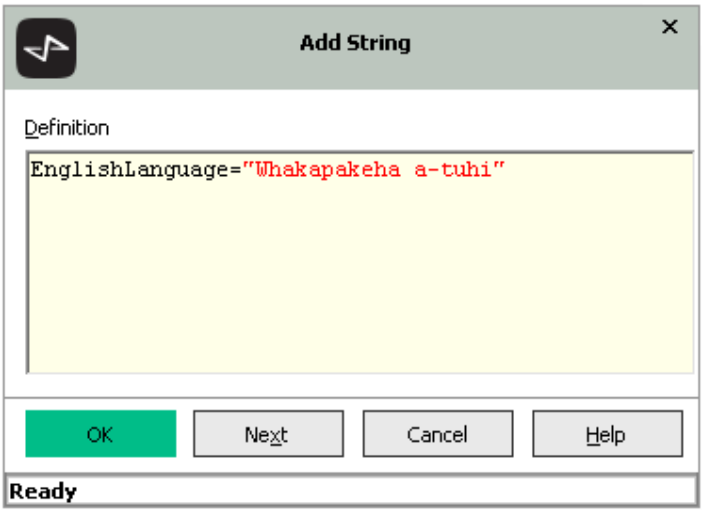
In this exercise, create translatable strings with two definitions for each string: one for English and one for te reo Maori.

1. Select **InternationalSchema** in the Schema Browser, select the **Strings** command from the Schema menu.

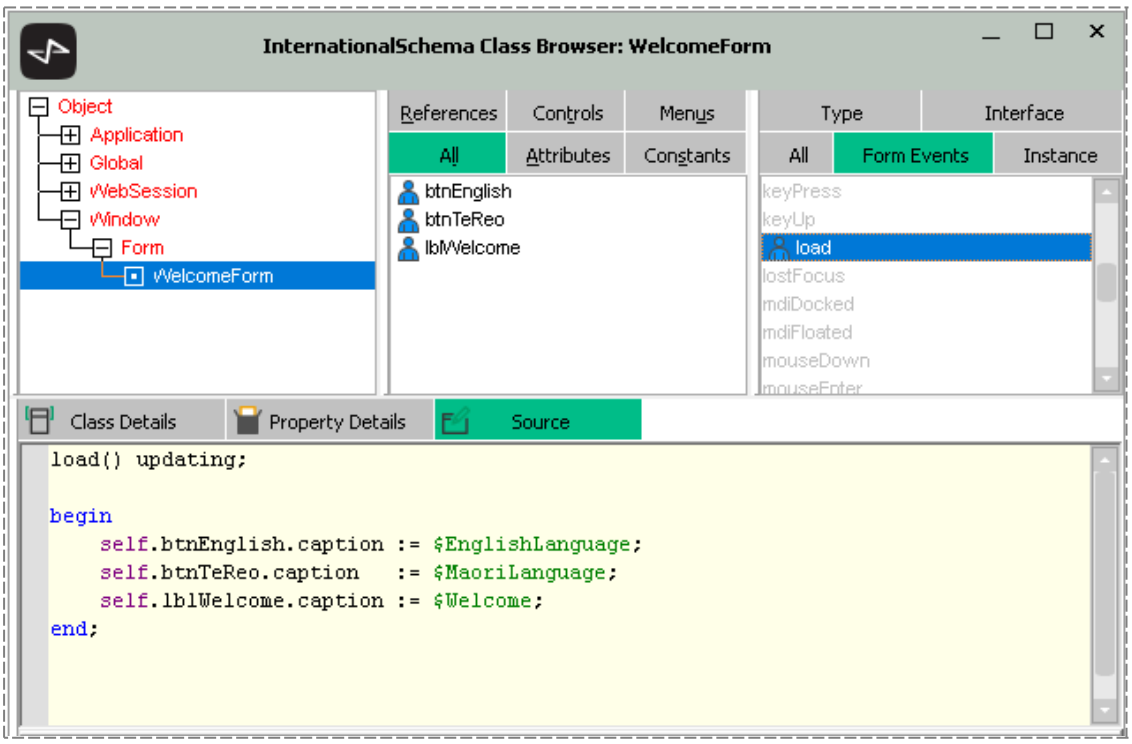
   In the String Browser that is displayed add the following translatable strings.

String Name	English	Maori
EnglishLanguage	English	Whakapakeha a-tuhi
MaoriLanguage	Maori	Te Reo
Welcome	Welcome and thank you for visiting.	Nau mai, ka mihi ki a koe mo te haerenga.

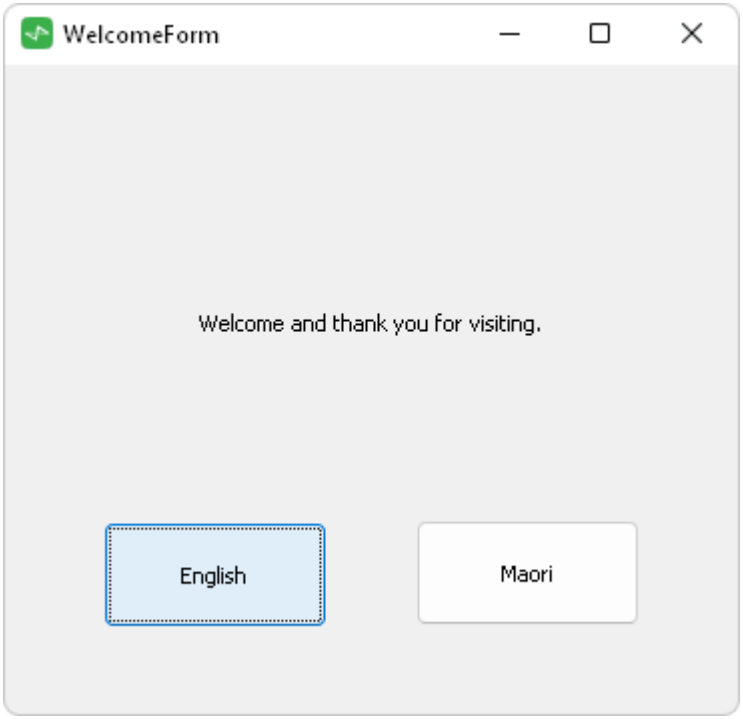
To add a new string, click **Add** in the String Browser and then specify it in the format *string-name*="*description*"; for example:



2. In the **Form Events** folder for the **WelcomeForm**, code the **load** method as follows.



3. Run the **InternationalSchema** application by right-clicking on the **Run Application** toolbar button. You should see the following form, in English.



Exercise 3 – Translating Translatable Strings

In this exercise, allow for the runtime translation of the Welcome Form between English and Maori by using the **Application** class **setJadeLocale** method.

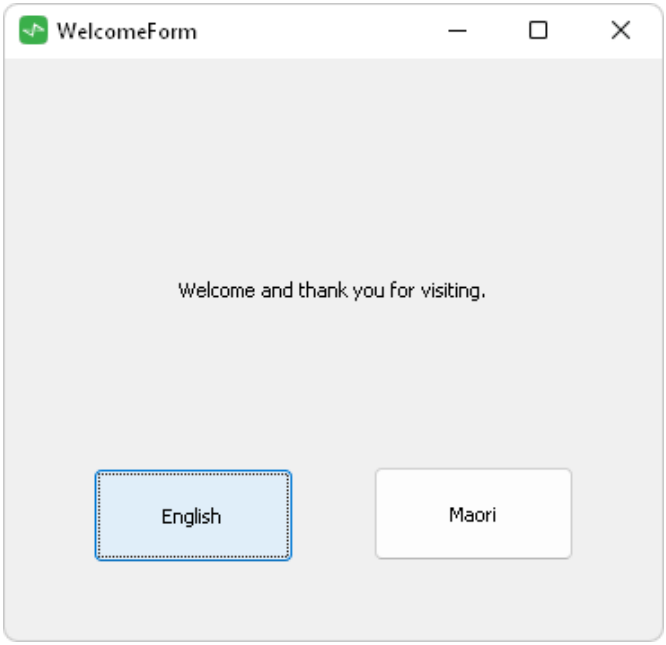
1. Code the **click** method of the **btnEnglish** button as follows.

```
btnEnglish_click(btn: Button input) updating;

begin
    app.setJadeLocale(LCID_English_NewZealand);
    self.load();
end;
```

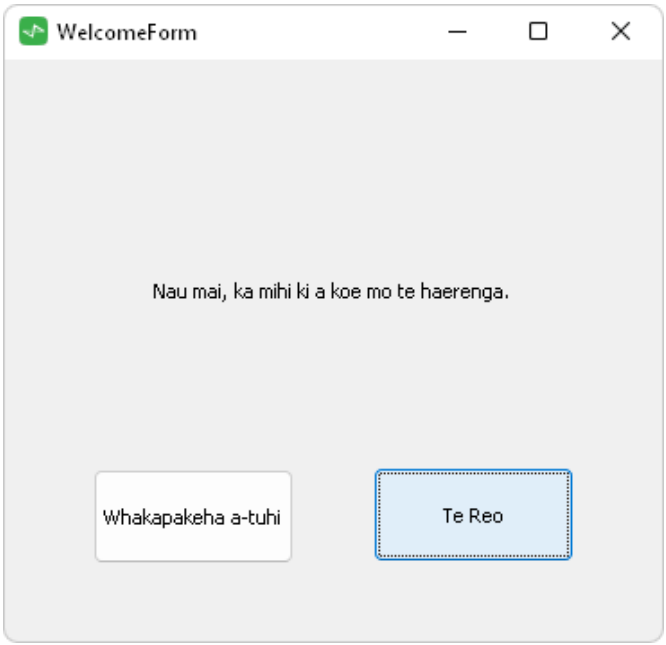
2. Code the **click** method of the **btnTeReo** button as follows.

```
btnTeReo_click(btn: Button input) updating;

begin
    app.setJadeLocale(LCID_Maori_NewZealand);
    self.load();
end;
```

3. Run the **InternationalSchema** application by right-clicking on the **Run Application** toolbar button. You should see the following form, in English.



4. Click the **Maori** button, to translate the form into te reo Maori.



5. Click the **Whakapakeha a-tuhi** button, to translate the form back into English.

Note Typically, usability best practices suggest that all controls that translate a page or form should be written in the target language rather than the currently selected language. However, this example shows the translation of strings rather than being an example of good usability.

Programmatically Maintaining Translatable Strings

In addition to using the String Browser to maintain translatable strings, the **Schema** class **addCompileTranslatableString** and the **TranslatableString** class **updateCompile** methods enable you to add and edit translatable strings, respectively.

The **addCompileTranslatableString** method, which is used to add new translatable strings to all locales of a schema at run time, takes the following parameters.

Parameter	Type	Purpose
Source	String	The translatable string definition to be added to the schema and must be in the format *string-name*="*description*". However, as nesting " " characters is not possible if using a string literal, single quote ' ' characters can be substituted, as required.
errorCode	Integer output	This output parameter is set to the error code after running the method. It is zero (**0**) if there is no error or it can be turned into a descriptive error message by using the **Process** class **getErrorText** method.
errorOffset	Integer output	This output parameter is set to the position in the **source** parameter where the error was encountered. Note that unlike most Jade strings and arrays, this offset begins at zero (**0**) for the first character of the source string rather than at **1**.
errorLength	Integer output	This output parameter is set to the number of characters in the **source** parameter that were in error.

The **updateCompile** method is to modify existing translatable strings and it takes the same parameters as the **addCompileTranslatableString** method. However, as this method is called from a translatable string rather than a schema, the target translatable string must first be located, and changes are specific to a target locale within that schema.

For example, to find and modify the **Welcome** string for the **English (New Zealand)** locale, you must first find the **English (New Zealand)** locale using the **Schema** class **getLocale** method, then retrieve the translatable string using the **getTranslatableStringLocal** method of the located **Locale**.

Exercise 4 – Adding a Translatable String Programmatically

In this exercise, use the **Schema** class **addCompileTranslatableString** method to add a translatable string to all base locales of the **InternationalSchema** schema.

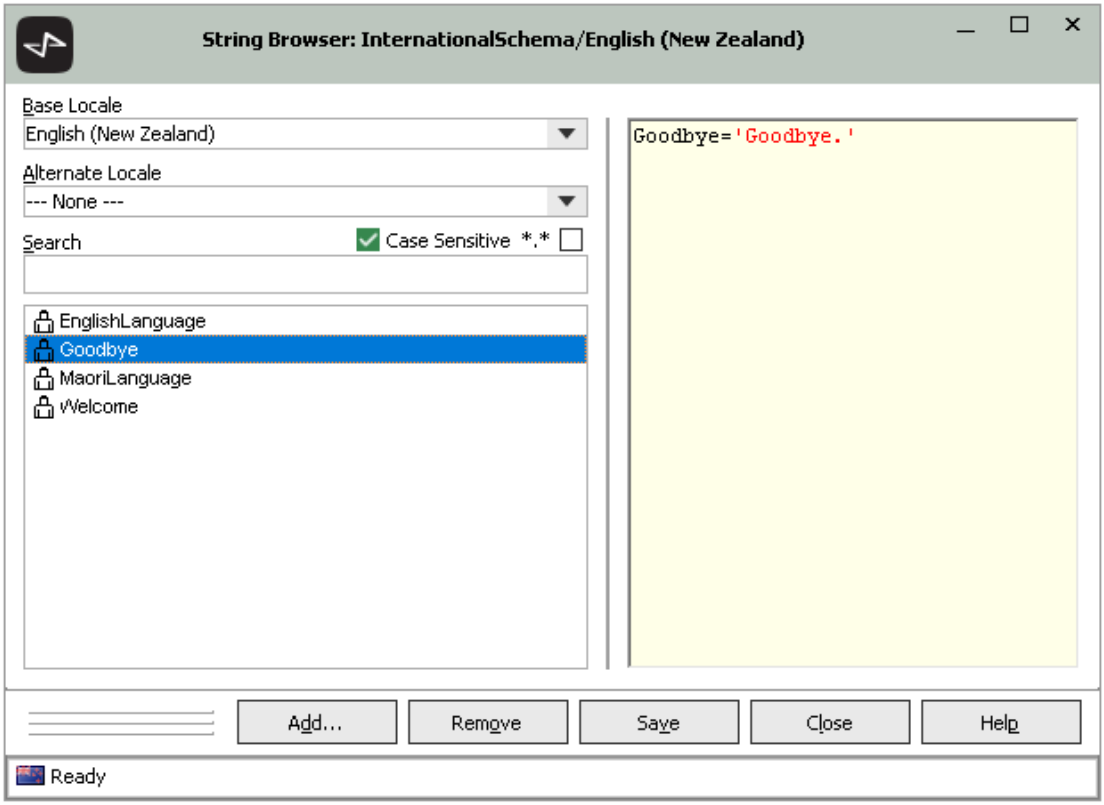
1. Add a **JadeScript** class method called **addTranslatableString** and code it as follows.

```
addTranslatableString();

vars
    errorCode, errorOffset, errorLength : Integer;
begin
    beginTransaction;
    currentSchema.addCompileTranslatableString(
                                                "Goodbye='Goodbye.'",
                                                errorCode,
                                                errorOffset,
                                                errorLength
                                                );
    if errorCode = 0 then
        commitTransaction;
    else
        write "ERROR: " & process.getErrorText(errorCode);
        write "At position " & errorOffset.String;
        write errorLength.String & " characters were invalid";
    endif;
end;
```

Note If there is an error, the **addCompileTranslatableString** method automatically aborts the transaction, so we don't need to include an **abortTransaction** instruction.

2. Run the method and then open the String Browser.

You should see that the **Goodbye** string has been added to the list of translatable strings.



However, viewing the **Maori (New Zealand)** locale shows that the same value has been given for the English and Maori languages.

Exercise 5 – Updating a Translatable String Programmatically

In this exercise, use the **TranslatableString** class **updateCompile** method to set the **Maori (Te Reo)** definition for the **Goodbye** string.

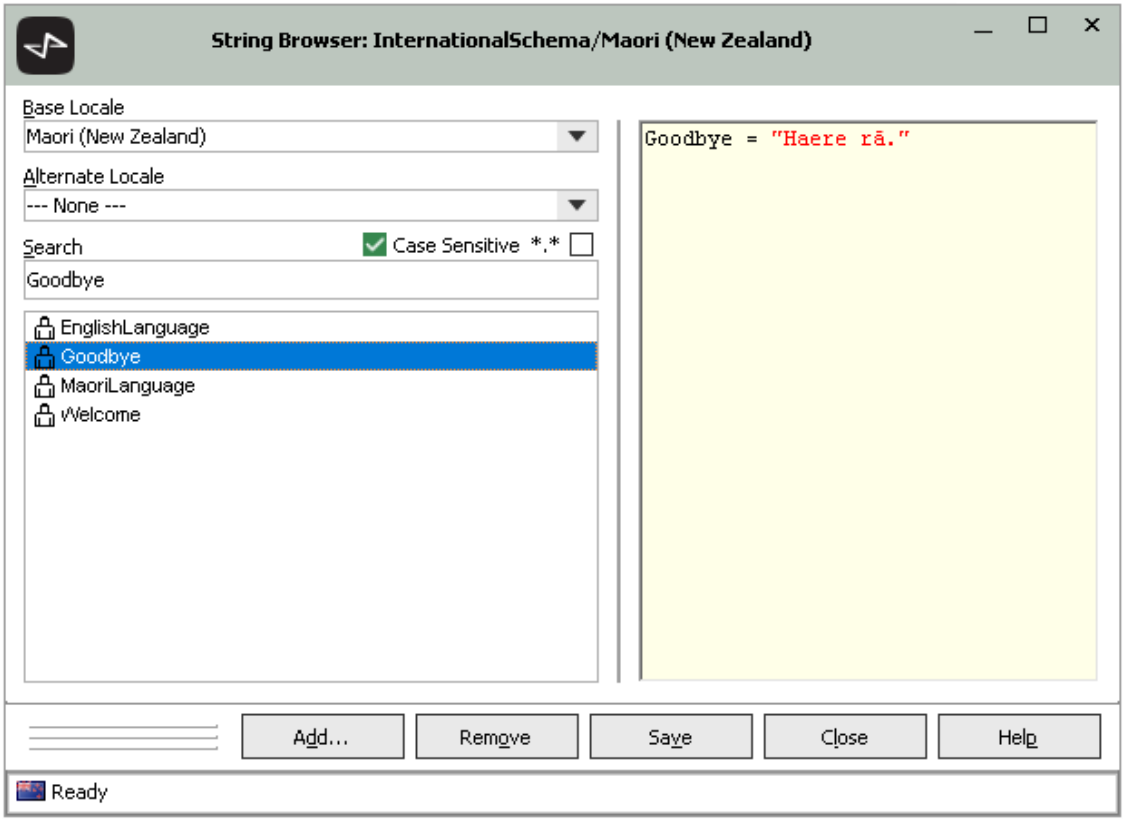
1. Add a **JadeScript** method called **updateTranslatableString** and code it as follows.

```
updateTranslatableString();

vars
    locale : Locale;
    errorCode, errorOffset, errorLength : Integer;
begin
    locale := currentSchema.getLocaleLocal(LCID_Maori_NewZealand.String);

    beginTransaction;
    locale.getTranslatableStringLocal("Goodbye").updateCompile(
                                                        'Goodbye = "Haere rā."',
                                                        errorCode,
                                                        errorOffset,
                                                        errorLength);
    if errorCode = 0 then
        commitTransaction;
    else
        write "ERROR: " & process.getErrorText(errorCode);
        write "At position " & errorOffset.String;
        write errorLength.String & " characters were invalid";
    endif;
end;
```

Note In ANSI Jade systems, the **ā** character is replaced by an **a**.

2. Run the method and then open the String Browser.

You should see that the **Goodbye** string's definition has been changed to **Haere rā** for the **Maori** locale.



Date Formats

By default, when using the **shortFormat** or **longFormat** method of the **Date** primitive type, the Jade Platform uses the date format of the current locale.

However, you can explicitly specify a date format for a specific date string by:

- Using the **format** method of the **Date** type.

  This method takes a string representation of a date format and returns the date in the specified format. For example, passing **"dd.MM.yyyy"** returns **17.08.2023** if the date is the 17th of August 2023.

- Creating user date formats with the Format Browser and then using the **userFormat** method of the **Date** primitive type.

The string representation of the date format that is passed to the **format** method of the **Date** primitive type is called the *picture* of the format.

The picture can contain the following string picture elements, or tokens. (Separate each element with a space or separator character.)

Picture	Description	Output Example
d	Day of the month as digits, with no leading zero.	9
dd	Day of the month as digits, with a leading zero, if applicable.	09
ddd	Day of the month as abbreviated name, as specified in the locale definition.	Wed
dddd	Day of the week as the full name.	Wednesday

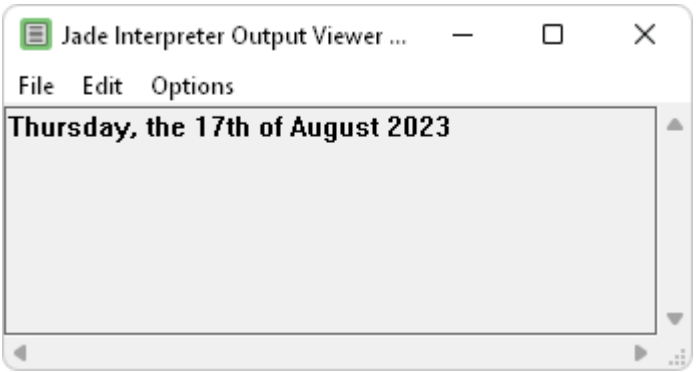
Picture	Description	Output Example
M	Month as digits, with no leading zero.	1
MM	Month as digits, with a leading zero, if applicable.	01
MMM	Month as abbreviated name, as specified in the locale definition.	Jan
MMMM	Month as full name.	January
y	Year, represented by only the last two digits, or last one digit if the last two digits are less than 10.	8
yy	Year, represented by only the last two digits.	18
yyy	Year, represented by all significant digits.	2018

Note The locale definitions for dates are specified by the operating system, rather than in Jade.

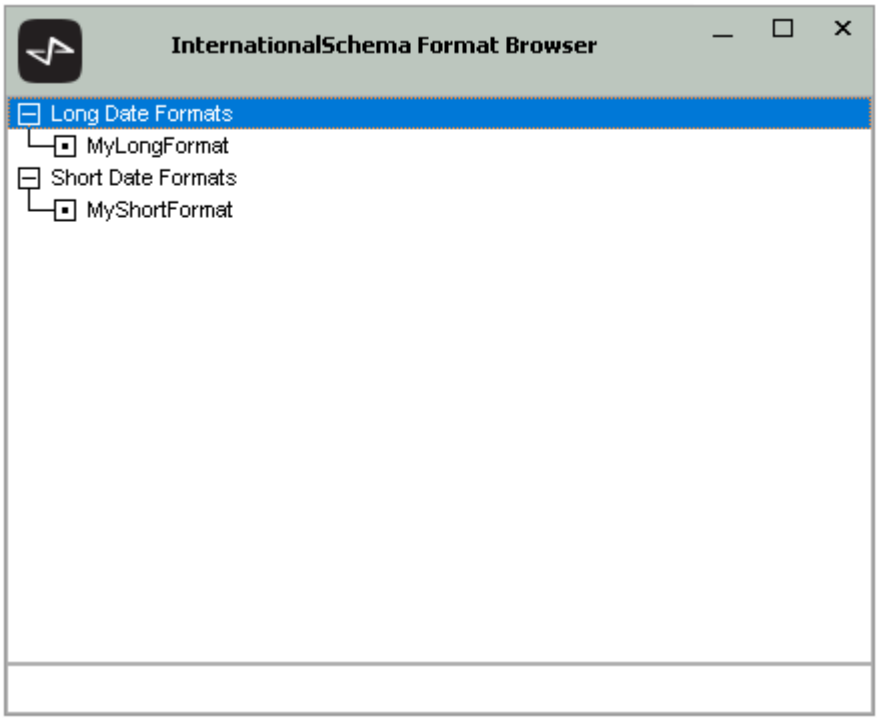
When constructing a picture string for a date format, you will typically use only one of each of the day, month, and year combined with preferred delimiters or other characters, and in your preferred order. However, you can combine delimiters and elements in any manner, including repeating elements; for example, consider the following **JadeScript** class **dateExample** method.

```
dateExample();

vars
    today : Date;
begin
    write today.format("dddd, the dth of MMMM yyyy");
end;
```

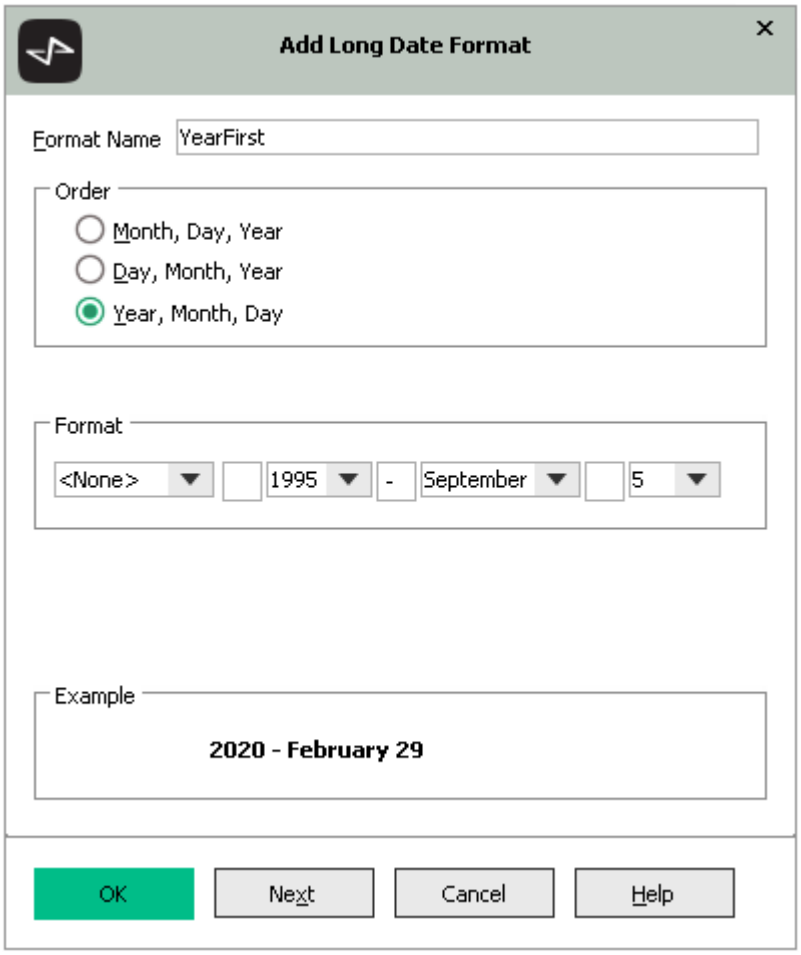
In this method, the day of the month is presented in both full name and single digit, and words are used to form a sentence containing the date. Running the method produces the following output.



As the usage of user-defined date formats increases, it is useful to create and save common formats by using the Format Browser, accessed by selecting the **Formats** command from the Schema menu.



The Format Browser provides the ability to create date formats via a form, label them with a descriptive name, and then use them multiple times in a schema by using the **userFormat** method of the **Date** primitive type.

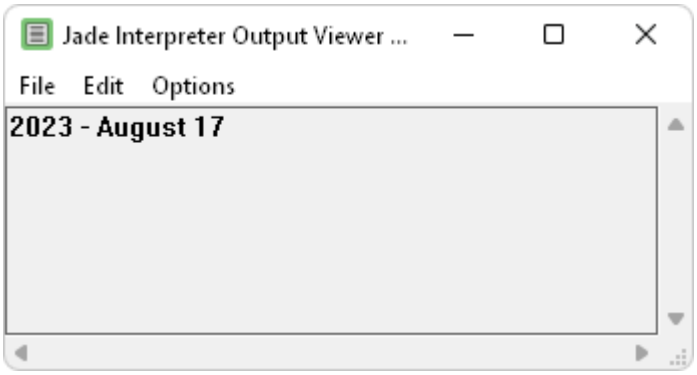


When a date format is created, it can be added to a Jade method by passing it to the **userFormat** method of the **Date** primitive type.

To access a created date format, prefix the format name with a dollar sign (**$**), as shown in the following **JadeScript** method.

```
dateExample();

vars
    today : Date;
begin
    write today.userFormat($YearFirst);
end;
```

Running the method produces the following output.



Exercise 6 – Adding a Date Format

In this exercise, create several date formats and present the same date in different ways, using the created date formats.

1. With **InternationalSchema** selected in the Schema Browser, open the Format Browser by selecting the **Formats** command from the Schema menu.

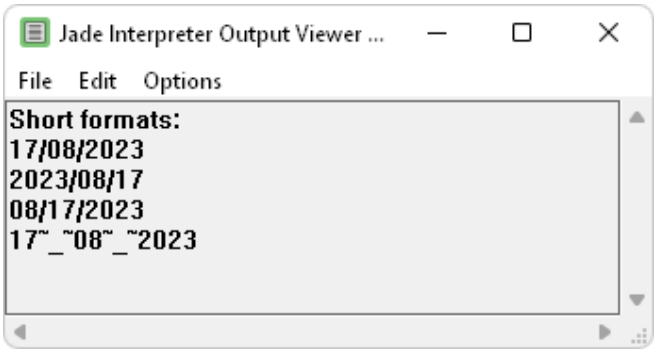
2. Add the following short date formats. (You can right-click and then select **Add Short Date Format** from the popup (or context) menu or select the **Add Short Date Format** command from the Formats menu.)

Name	Delimiter	Format
ShortForwards	/	Day, Month, Year
ShortBackwards	/	Year, Month, Day
ShortUS	/	Month, Day, Year
ShortSilly Delimiter	~_~	Day, Month, Year

3. Add a **JadeScript** class method called **testDateFormats** and code it as follows.

```
testDateFormats();

vars
    today : Date;
begin
    write "Short formats:";
    write today.userFormat($ShortForwards);
    write today.userFormat($ShortBackwards);
    write today.userFormat($ShortUS);
    write today.userFormat($ShortSillyDelimiter);
end;
```

Running the method produces the following output.



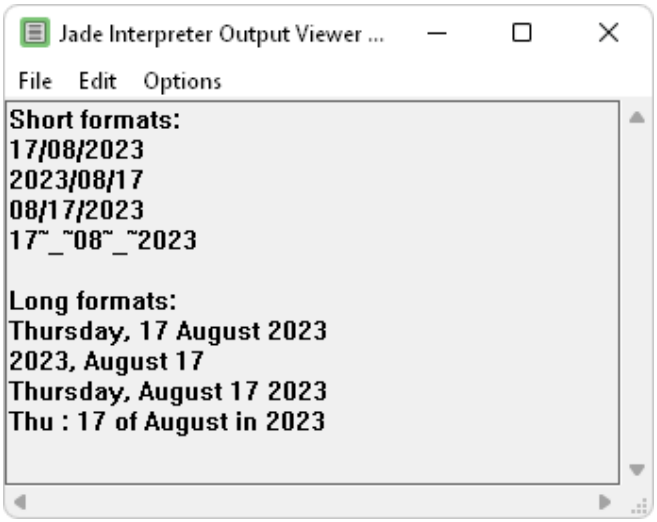
4. Add the following long date formats from the Format Browser.

Name	Format
LongForwards	Format: Sunday , 5 September 1995
LongBackwards	Format: <None> 1995 , September 5
LongUS	Format: Sunday , September 5 1995
LongSillyDelimiter	Format: Sun : 5 of September in 1995

5. Modify the **testDateFormats** method as follows.

```
testDateFormats();

vars
    today : Date;
begin
    write "Short formats:";
    write today.userFormat($ShortForwards);
    write today.userFormat($ShortBackwards);
    write today.userFormat($ShortUS);
    write today.userFormat($ShortSillyDelimiter);

    write CrLf & "Long formats:";
    write today.userFormat($LongForwards);
    write today.userFormat($LongBackwards);
    write today.userFormat($LongUS);
    write today.userFormat($LongSillyDelimiter);
end;
```

Running the method produces the following output.



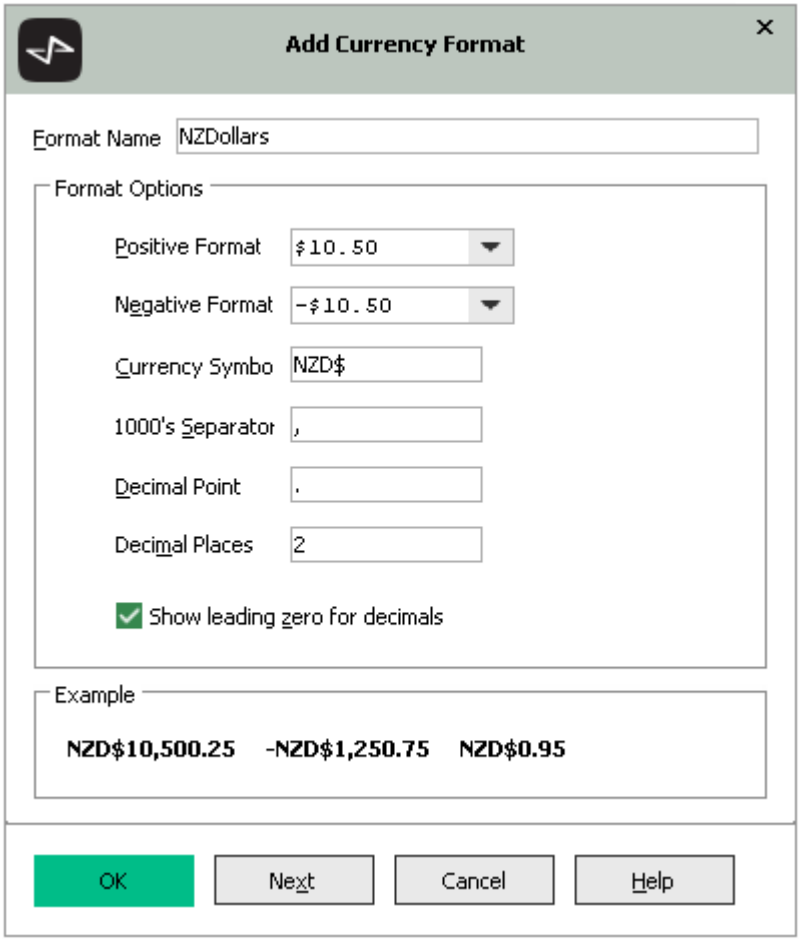
6. Modify the **testDateFormats** method as follows.

```
testDateFormats();

vars
    today : Date;
begin
    write "Short formats:";
    write today.userFormat($ShortForwards);
    write today.userFormat($ShortBackwards);
    write today.userFormat($ShortUS);
    write today.userFormat($ShortSillyDelimiter);

    write CrLf & "Long formats:";
    write today.userFormat($LongForwards);
    write today.userFormat($LongBackwards);
    write today.userFormat($LongUS);
    write today.userFormat($LongSillyDelimiter);

    write CrLf & "Custom formats:";
    write today.format("d MMM yyy");
    write today.format("M d yyy");

    // One of these will work, the other will do something strange...
    write today.format("The month is MMMM");
    write today.format("Today is dddd");
    // Why is this?
    // Bonus exercise: Fix the non-working one!
end;
```

7. Before you run the **testDateFormats** method, try and work out which of the format methods will have issues.

Currency Formats

Like date formats, Jade uses the currency format of the operating system by default if no currency format is specified. To specify a custom currency format, you can add a new currency format using the Format Browser and add it to a Jade method using the **userCurrencyFormat** method of the **Decimal** primitive type.

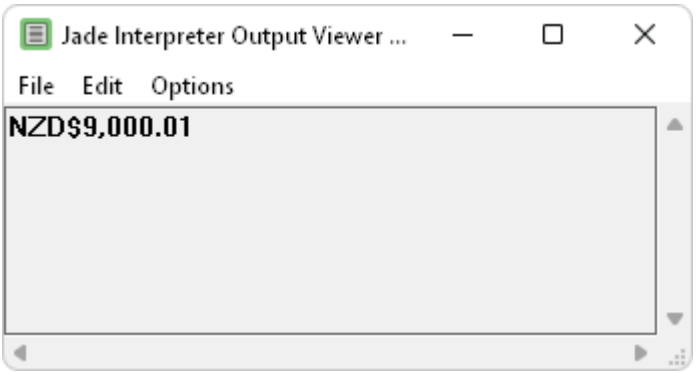


When a currency format is created, it can be added to a Jade method by passing it to the **userCurrencyFormat** method of the **Decimal** primitive type.

To access a created currency format, prefix the format name with a dollar sign (**$**), as shown in the following **JadeScript** class method.

```
currencyExample();

vars
    money : Decimal [12,2];
begin
    money := 9000.01;
    write money.userCurrencyFormat($NZDollars);
end;
```

Running the method produces the following output.



Exercise 7 – Adding a Currency Format

In this exercise, add several currency formats and present the same decimal amount in different ways, using the created currency formats.

1. With **InternationalSchema** selected in the Schema Browser, open the Format Browser by selecting the **Formats** command from the Schema menu.

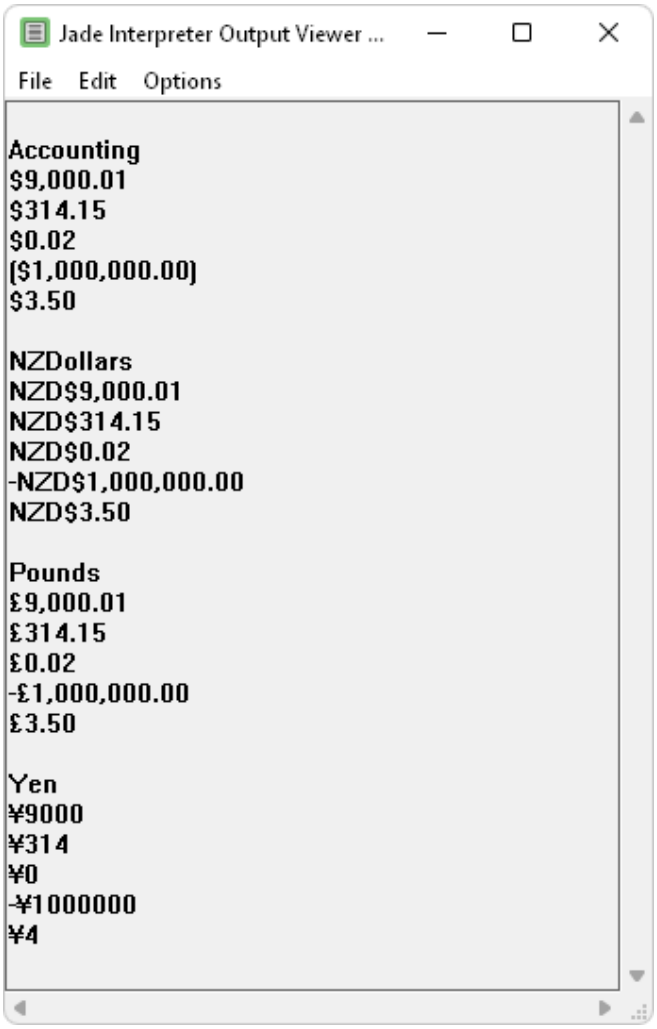
2. Add the following currency formats. (You can right-click and then select **Add Currency Format** from the popup (or context) menu or select the **Add Currency Format** command from the Formats menu.)

Name	Fields to Change	Values
Accounting	Negative Format	($10.5)
NZDollars	Currency Symbol	NZ$
Pounds	Currency Symbol	£
Yen	Currency Symbol Decimal Places 1000s Separator	¥ 0 None

3. Create a **JadeScript** class method called **testCurrencyFormats** and code it as follows.

```
testCurrencyFormats();

vars
    format  : CurrencyFormat;
    formats : CurrencyFormatArray;
    number  : Decimal[12,2];
    numbers : DecimalArray;
begin
    create formats transient;
    formats.add($Accounting);
    formats.add($NZDollars);
    formats.add($Pounds);
    formats.add($Yen);
    create numbers transient;
    numbers.add(9000.01);
    numbers.add(314.15);
    numbers.add(0.02);
    numbers.add(-1000000);
    numbers.add(3.50);

    foreach format in formats do
        write CrLf & format.getName();
        foreach number in numbers do
            write number.userCurrencyFormat(format);
        endforeach;
    endforeach;

epilog
    delete formats;
    delete numbers;
end;
```

4. Running the method produces the following output.



Jade Interfaces

An interface is a mechanism that provides a set of methods guaranteed to be available on any implementing class. When a class implements an interface, it agrees to implement all methods defined by the interface. With this contract in place, the implementing classes can participate in useful type safe callback mechanisms.

Exposing a class via its interface effectively hides the implementation details of the class, as the user is forced to communicate only through the public interface methods.

Classes that implement an interface can be grouped by that interface type; for example, a collection can have a membership of a specified interface.

Using interfaces, non-related classes can be grouped to capture their similarities, without the need to artificially force a class relationship. Think of this as allowing a class to perform multiple roles outside of the role dictated by its class hierarchy.

Tip Interfaces are a useful way to group classes by behavior.

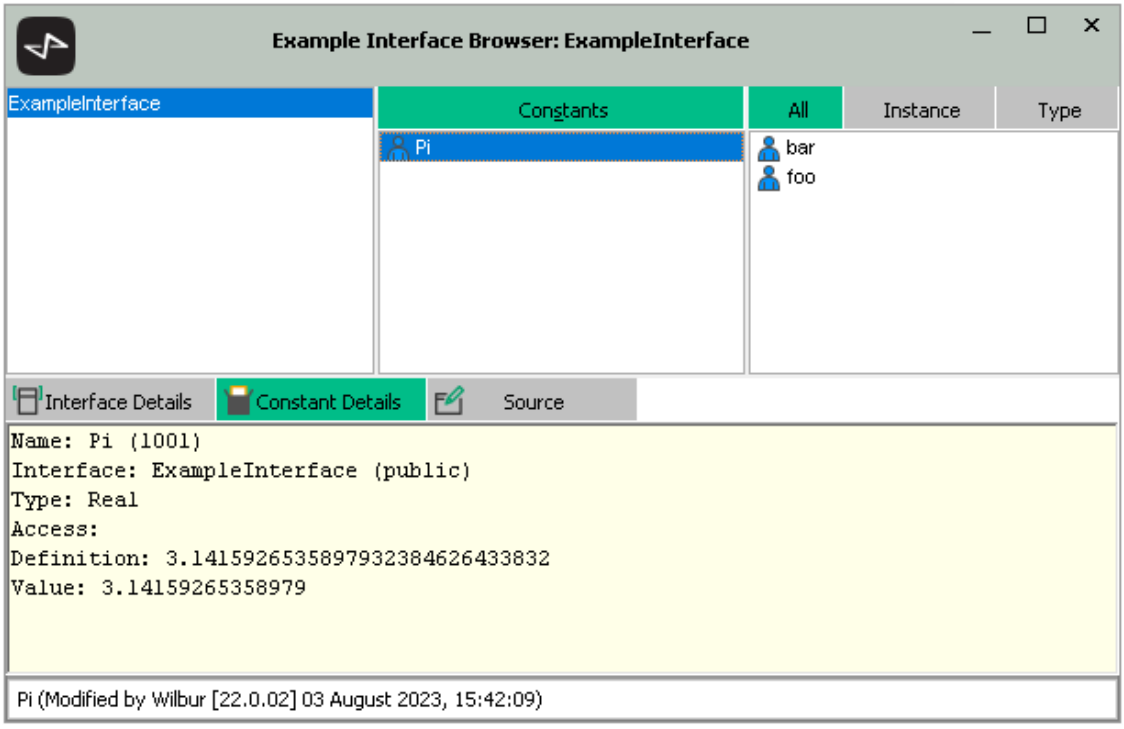
The Interface Browser

Jade allows for the defining of interfaces through the Interface Browser, in which you can specify the required methods and constants.

The following actions open the Interface Browser.

- Clicking on the **Browse Interfaces** icon in the browser toolbar.
- Selecting the **Interfaces** command from the Browse menu.
- Pressing Ctrl+N.

An Interface Browser is then opened for the selected schema.



The pane at the upper left contains a list of all interfaces defined for the selected schema, the middle pane displays any constants defined for the selected interface, and the pane at the upper right displays the required methods of the selected interface.

Although you may notice that the Interface Browser is similar to the Class Browser, note the following differences.

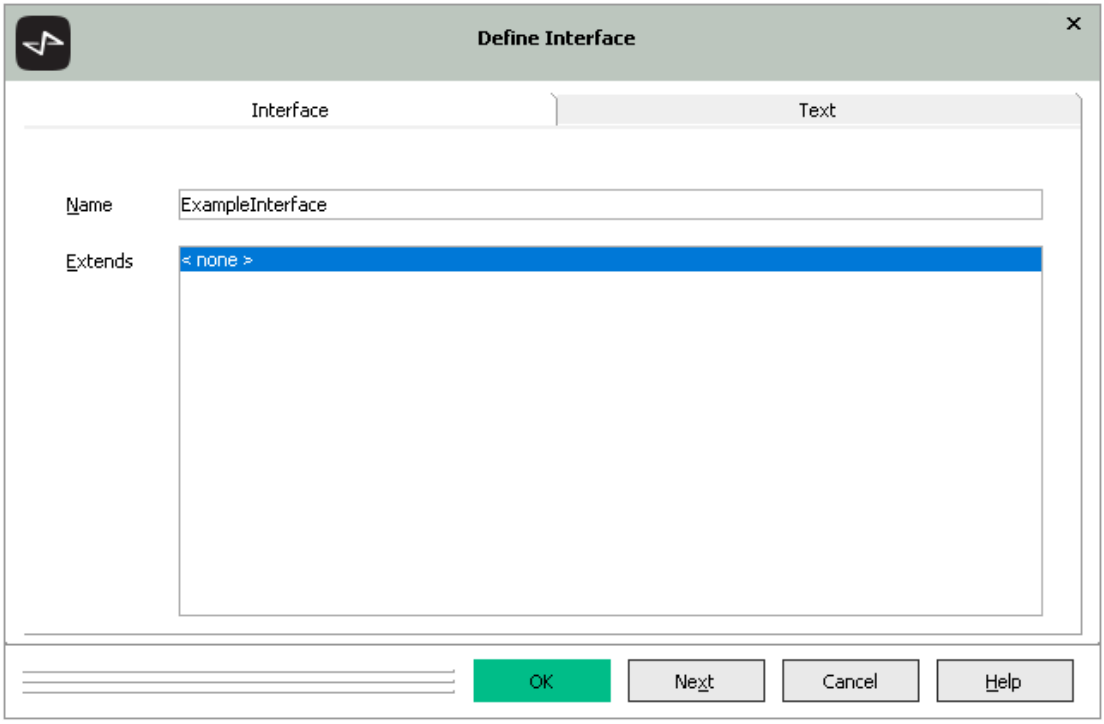
- Interfaces can define constants and methods only.
- Interface methods contain only the method signature; they do not have any implementation.
- Interfaces are always displayed in alphabetical order, with a flat (rather than nested) structure.
- While classes can extend a single superclass only, interfaces can extend any number of other interfaces.

Defining an Interface

The following actions create a new interface from the Interface Browser.

- Select the **Add** command from the Interfaces menu.
- Right-click in the pane at the upper left and then select the **Add** command from the popup (context) menu.

The Define Interface dialog is then displayed.

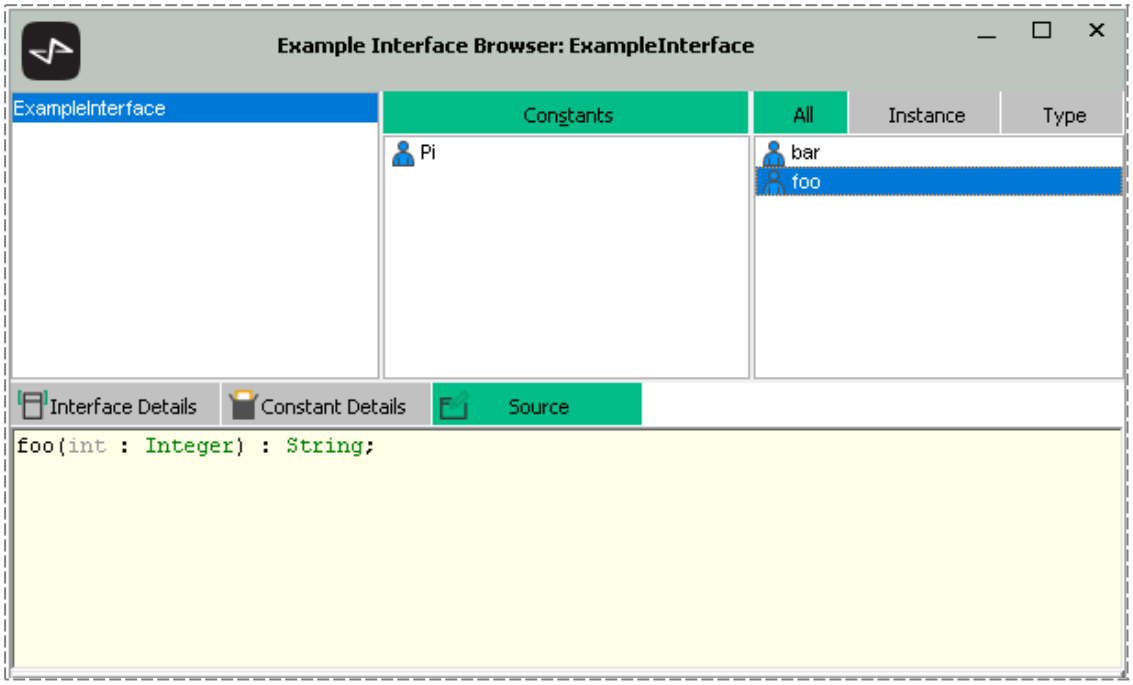


Enter a unique name for the interface, and whether to extend from one of the existing interfaces.

To create the interface, click **Next** to create the interface while keeping the Define Interface dialog open (so that you can create multiple interfaces) or click **OK** to create the interface and close the Define Interface dialog.

Note Most of the normal options for creating a method are disabled for interface methods, because an interface contains the method signatures only. It is up to implementing classes to provide the method sources.

To add parameters or return types to an interface method, select the method in the Interface Browser and then modify the method signature.



Note Attempting to provide a method body in an interface method will result in a compiler error.

Implementing an Interface

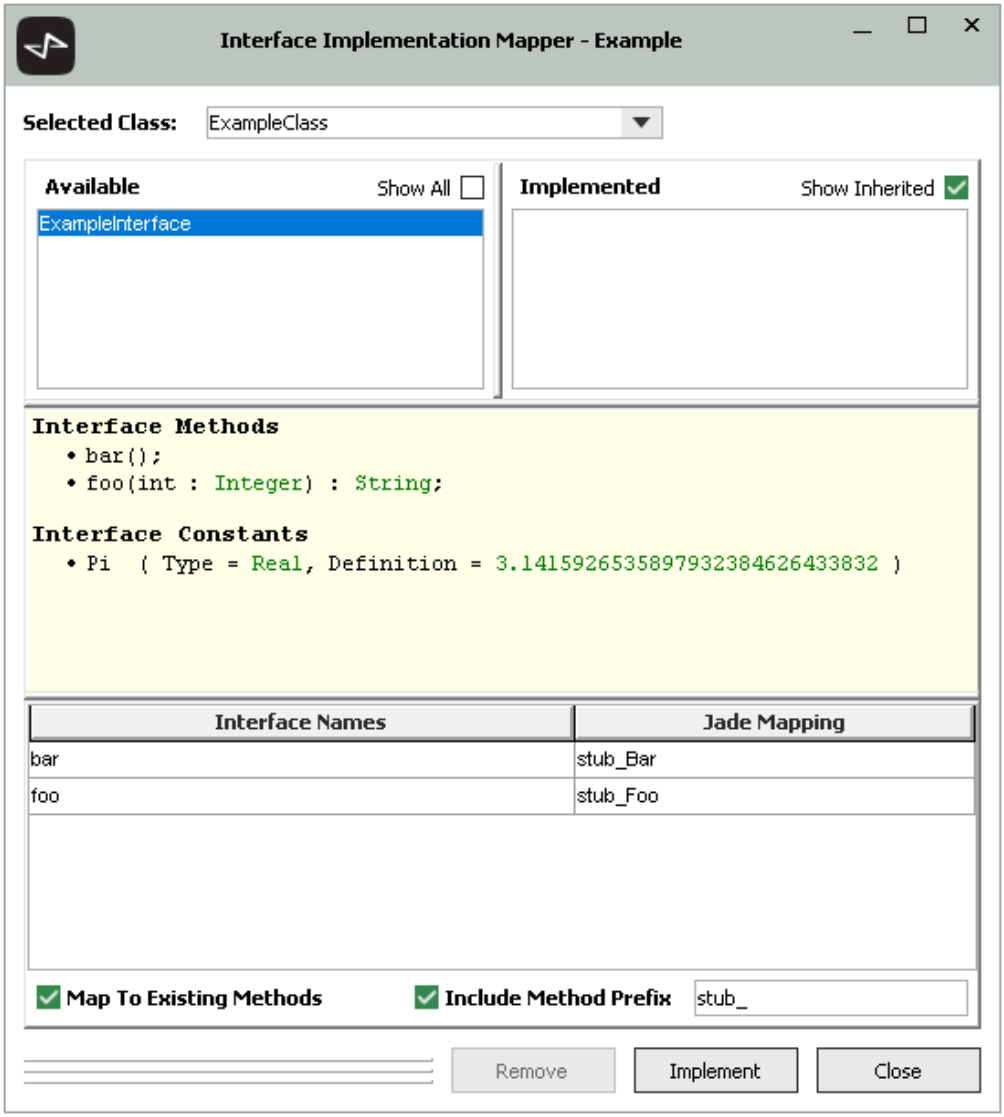
For a class to be able to implement an interface, it must:

- Explicitly indicate that it will provide the specific interface.
- Fulfil all methods of the interface.

To indicate the interfaces a class will implement, select the class in the Class Browser and then open the Interface Implementation Mapper dialog by doing one of the following.

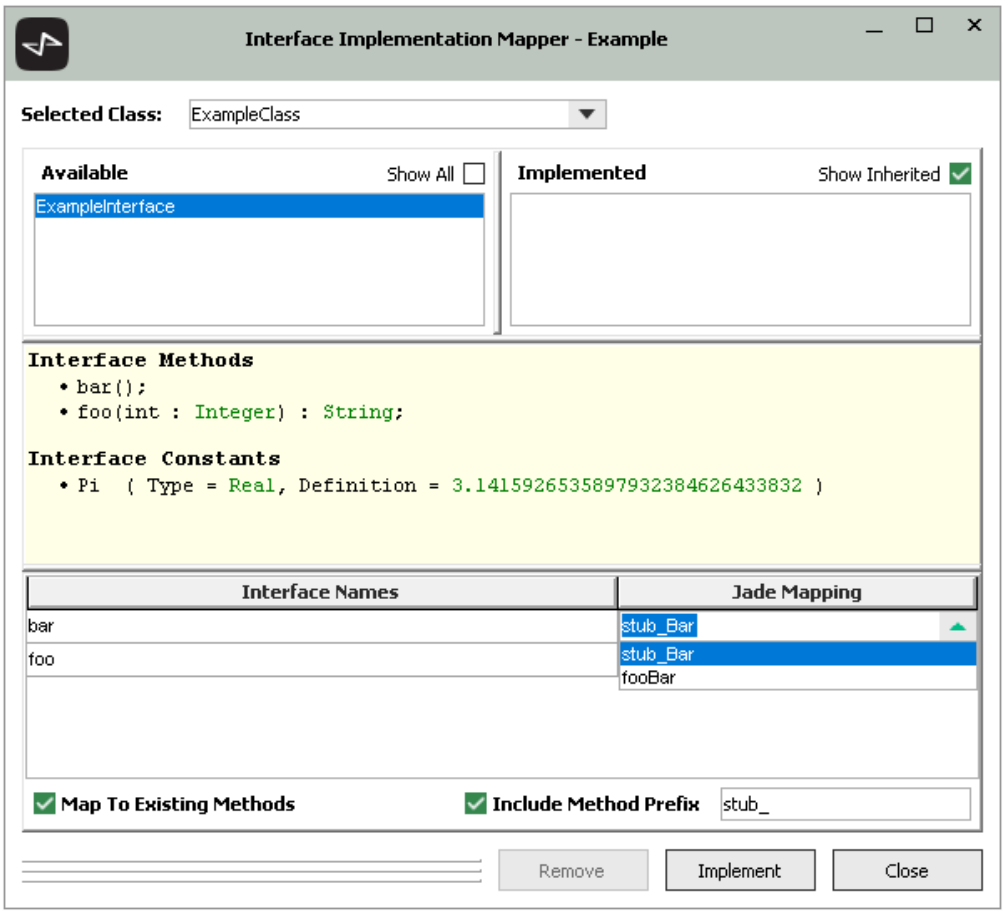
- Selecting the **Interface Mapping** command from the Classes menu.
- Right-clicking the class in the Class Browser and then selecting the **Interface Mapping** command from the popup menu.

To choose an interface to add to the class, select it in the **Available** list box.



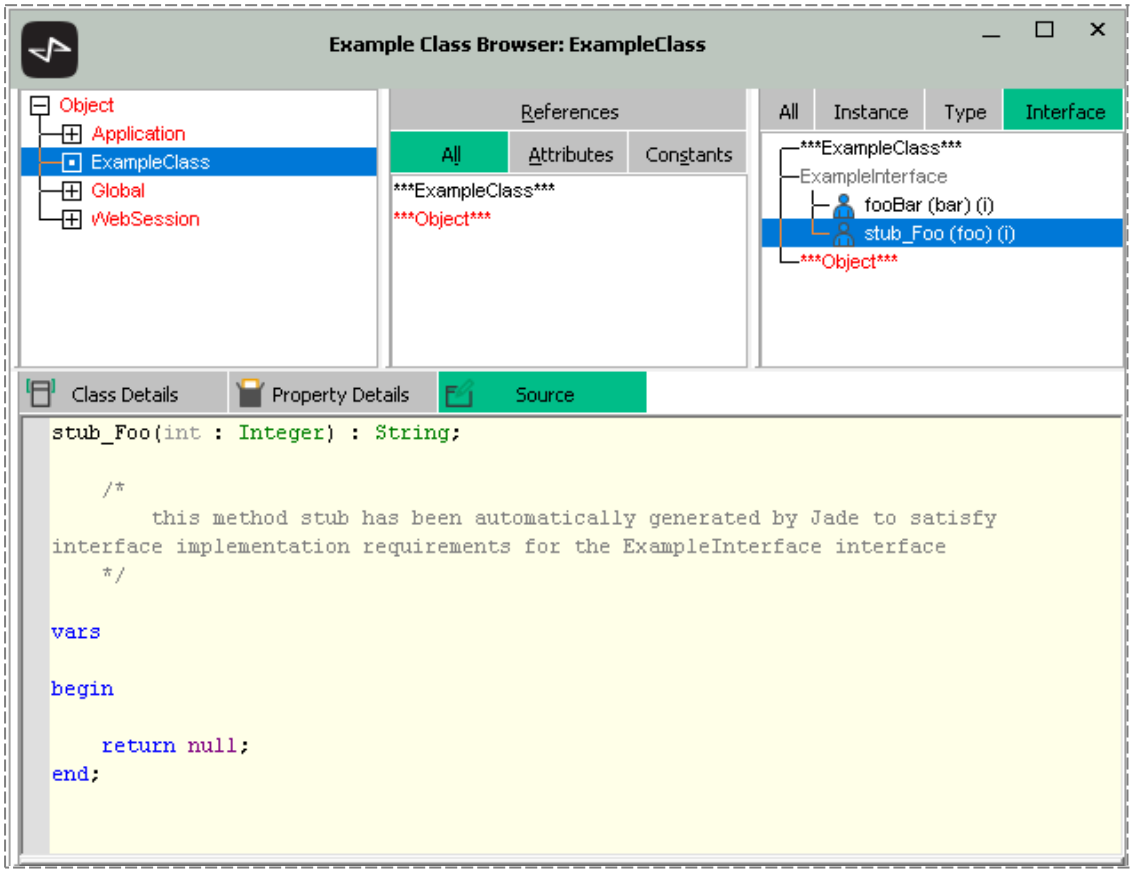
The details of the interface will be displayed, including the signature of the interface methods and the types and values of the interface constants.

For each of the interface methods, the class must provide an implementation. By default, these will be automatically generated method stubs called **stub_*method-name***. However, you can click on any of these to select an existing method from the class.



Note The class method used to implement an interface method does not need to have the same name as the interface method; only the same parameters and return type. A single class method can fulfil the implementation of multiple interface methods if they have compatible method signatures.

To confirm the implementation, click **Implement**. The interface will then be added to the class, and by clicking on the **Interface** tab in the Methods List (the upper right pane) in the Class Browser, you can see the interface methods.



Note Any interface method with which no class method was mapped has a stub implementation.

Exercise 1 – Adding the IWithdrawable Interface

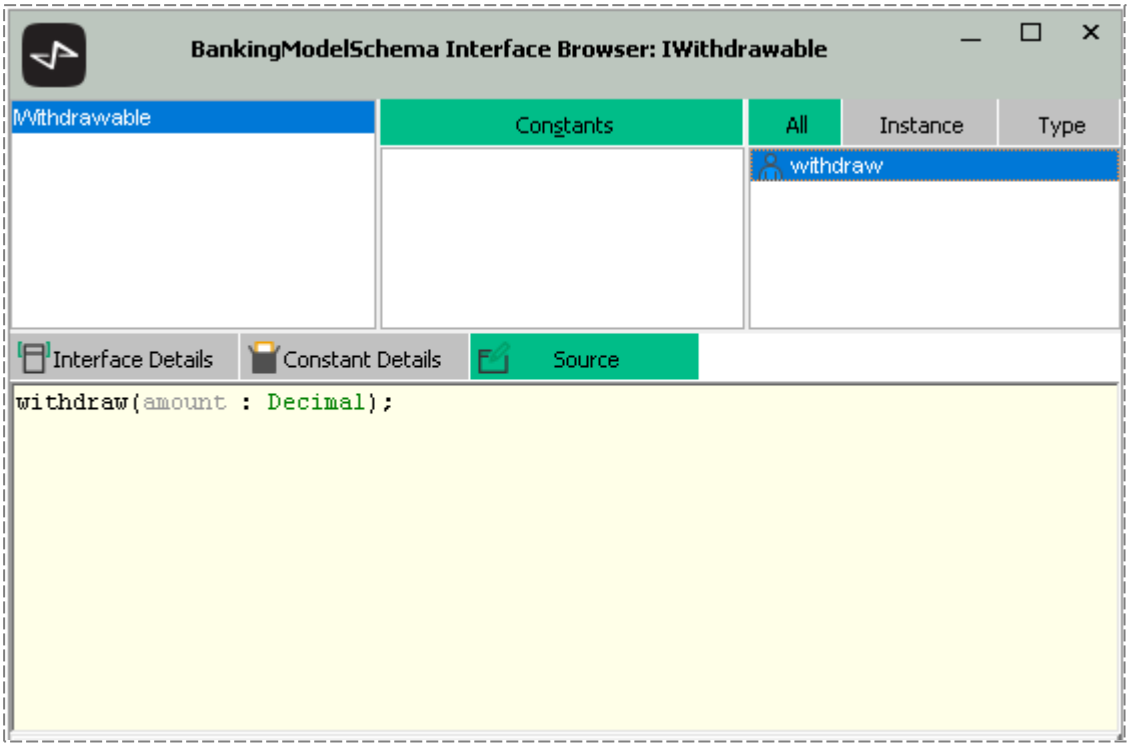
In this exercise, add an interface to **BankingModelSchema** that provides an alternative to the polymorphism approach used for **Accounts** in Module 9 of the Jade Platform Developer's course.

1. Select **BankingModelSchema** in the Schema Browser and then open the Interface Browser.
2. Add a new interface called **IWithdrawable** to the schema.

   Note Interfaces typically have a prefix of **I** or a suffix of **IF**.

3. Add a method called **withdraw** to the **IWithdrawable** interface.

4. Code the **withdraw** method signature, as follows.



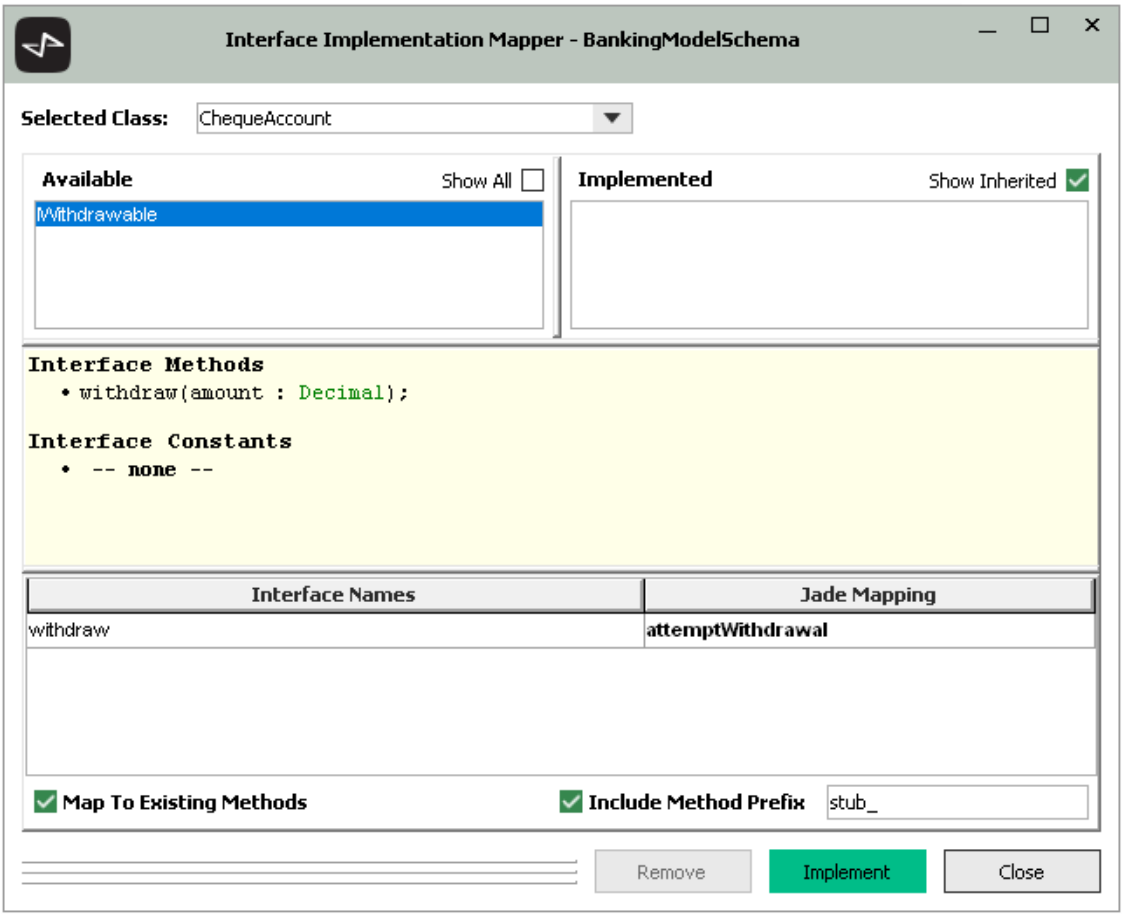
Exercise 2 – Implementing the IWithdrawable Interface

In this exercise, implement a variety of methods that all fulfil the withdraw interface method in different ways.

1. Add a method called **attemptWithdrawal** to the **ChequeAccount** class and code it as follows.

```
attemptWithdrawal(amount : Decimal) updating;

begin
    write "Cheque Balance:" & self.balance.String;
    write "Overdraft limit:"  & self.overdraftLimit.String;
    write "Requested Funds:" & amount.String;
    if amount <= self.balance + self.overdraftLimit then
        beginTransaction;
        self.balance := self.balance - amount;
        commitTransaction;
        write amount.String & " withdrawn successfully.";
        write self.balance.String & "remaining";
    else
        write "Insufficient funds: " & (self.balance + self.overdraftLimit).String & "Available";
    endif;
end;
```

2. Select the **ChequeAccount** class in the Class Browser and then open the Interface Implementation Mapper dialog by selecting the **Interface Mapping** command in the Classes menu.



3. Select **IWithdrawable** in the **Available** list box and then enter **attemptWithdrawal** for the withdraw method in the **Jade Mapping** column. Click **Implement**, to confirm your action and close the window.

Note While interface methods themselves may not have the updating method option, **updating** methods can fulfil **interface** methods.

4. Add a method called **attemptWithdrawal**, this time to the **SavingsAccount** class, and code it as follows.

```
attemptWithdrawal(amount : Decimal) updating;

begin
    write "Savings Balance: " & self.balance.String;
    write "Requested Funds: " & amount.String;
    if amount <= self.balance then
        beginTransaction;
        self.balance := self.balance - amount;
        commitTransaction;
        write amount.String & " withdrawn successfully.";
        write self.balance.String & " remaining";
    else
        write "Insufficient funds: " & (self.balance).String & " remaining";
    endif;
end;
```

5. Select the **SavingsAccount** class in the Class Browser and then open the Interface Implementation Mapper dialog by selecting the **Interface Mapping** command in the Classes menu.

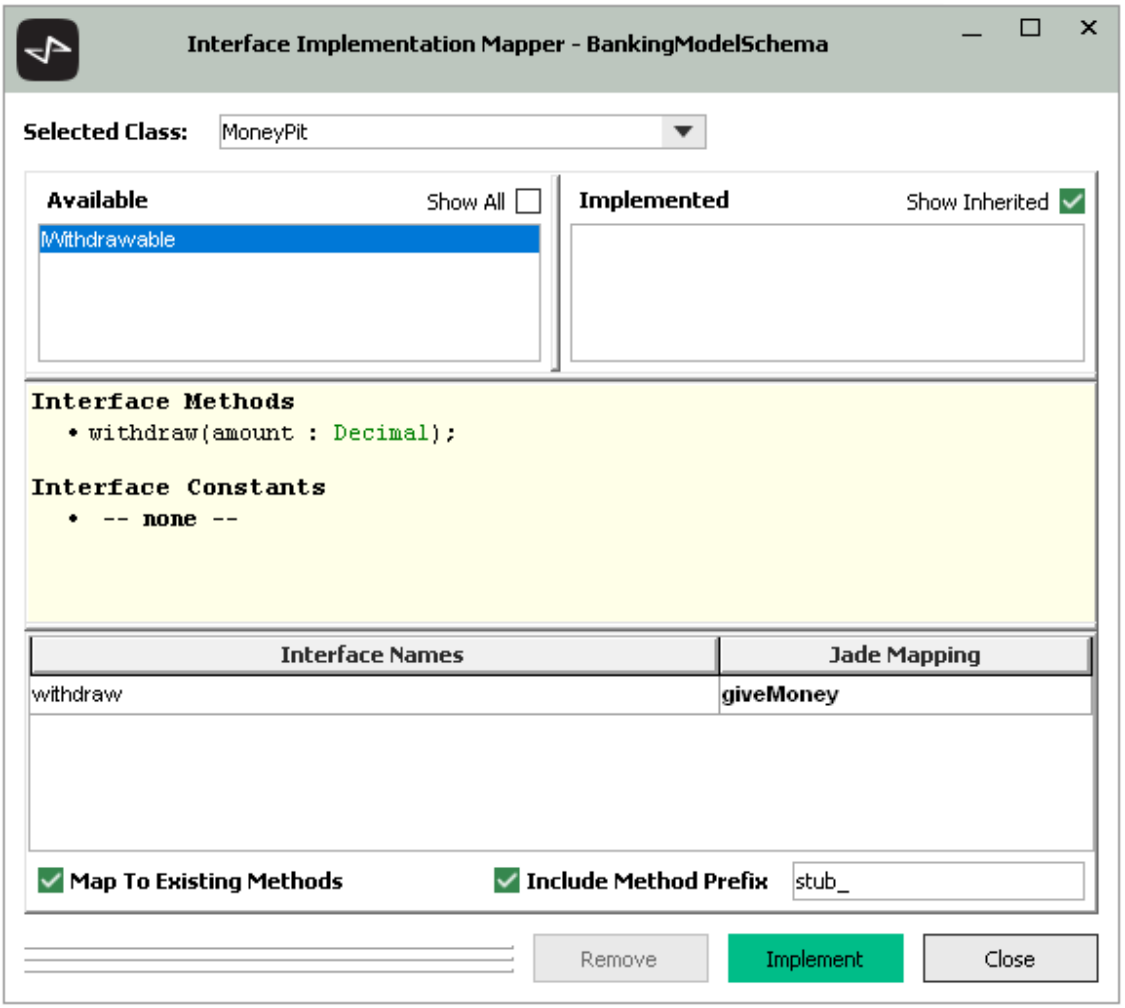
6. Repeat Step 3 of this instruction.

7. Create a new class called **MoneyPit** in **BankingModelSchema**, add a method called **giveMoney**, and code it as follows.

```
giveMoney(amount : Decimal);


begin
    write "Okay, have $" & amount.String;
end;
```

Note The **giveMoney** method is not an updating method. An interface method can have an updating or non-updating method fulfil it.

8. Select the **MoneyPit** class in the Class Browser and open the Interface Implementation Mapper dialog by selecting the **Interface Mapping** command in the Classes menu.



9. Select **IWithdrawable** in the **Available** list box and then enter **giveMoney** for the withdraw method in the **Jade Mapping** column. Click **Implement**, to confirm your action and close the window.

Exercise 3 – Testing the IWithdrawable Interface

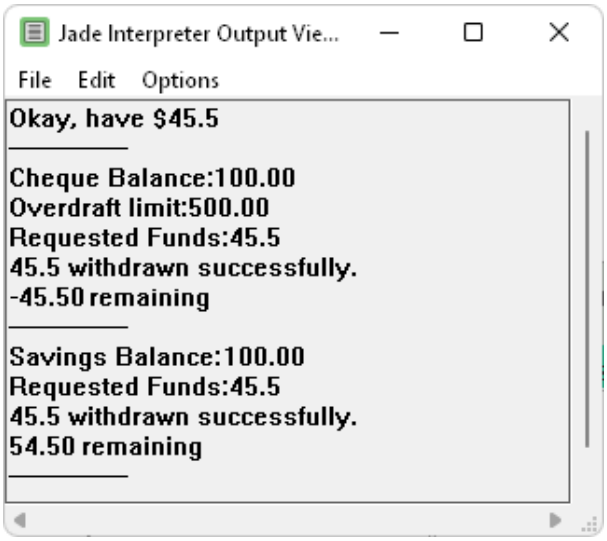
In this exercise, write a **JadeScript** class method that puts the three classes that implement **IWithdrawable** interface into a collection and then call the **withdraw** method of each one.

1. Add a new **ObjectArray** subclass called **Withdrawables**, with member type **IWithdrawable**.

2. Add a new **JadeScript** method called **testInterface**, and code it as follows.

```
testInterface();

vars
    dict : Withdrawables;
    acc  : IWithdrawable;
    pit  : MoneyPit;
begin
    create pit transient;
    create dict transient;

    dict.add(pit);
    dict.add(ChequeAccount.firstInstance().IWithdrawable);
    dict.add(SavingsAccount.firstInstance().IWithdrawable);
    foreach acc in dict do
        acc.withdraw(45.50);
        write "----------------";
    endforeach;
epilog
        delete pit;
        delete dict;
end;
```

3. Run the method. Output similar to the following should then be displayed.



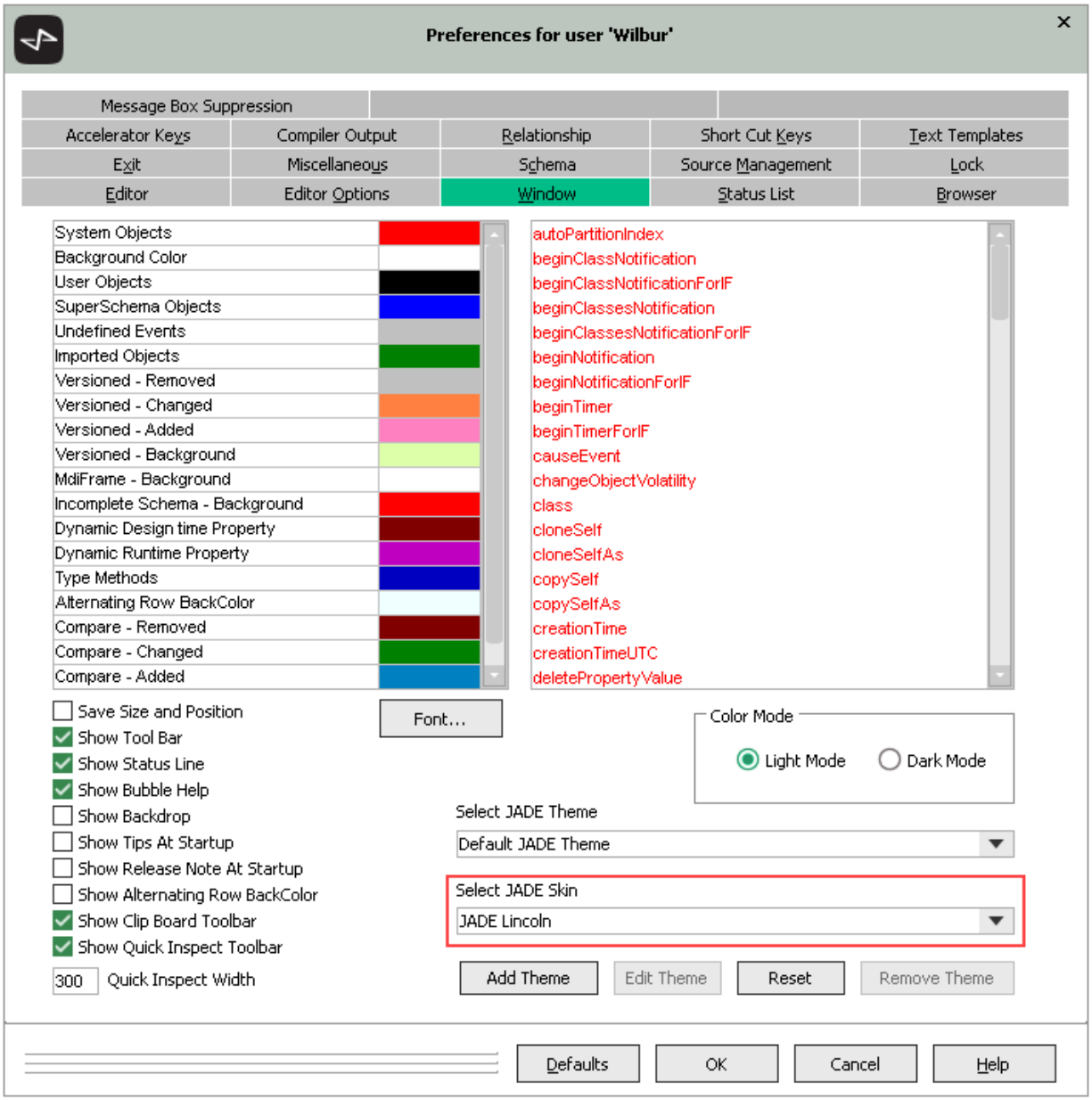
Jade Skins

Jade skins enable you to customize Jade form Graphical User Interface (GUI) elements. By configuring and saving colors, fonts, shapes, and sizes of the various controls on Jade forms, you can enforce a consistent look-and-feel across multiple applications.

Selecting a Skin for the Development Environment

As the Jade Platform development environment is implemented in Jade, you can use Jade skins to customize the look and feel of the development environment in the same way you can for any other Jade application.

To select a different skin for your Jade Platform development environment, select one of the provided skins in the **Select JADE Skin** drop-down list box on the **Window** sheet of the Preferences dialog (accessed by selecting the **Preferences** command from the Options menu).



Note **JADE Cashmere** is the default skin for versioned schemas.

If you use this skin for your standard Jade Platform development environment, you should change the default skin for versioned schemas (on the **Miscellaneous** sheet of the Preference dialog).

JadeSkinMaintenance Form

The **JadeSkinMaintenance** form (the Jade Skin Maintenance dialog) is used to maintain existing skins and define new skins.

This form enables you to re-skin any number of controls, forms, and menus. These re-skins are then combined into an application skin, which can be applied to any number of applications.

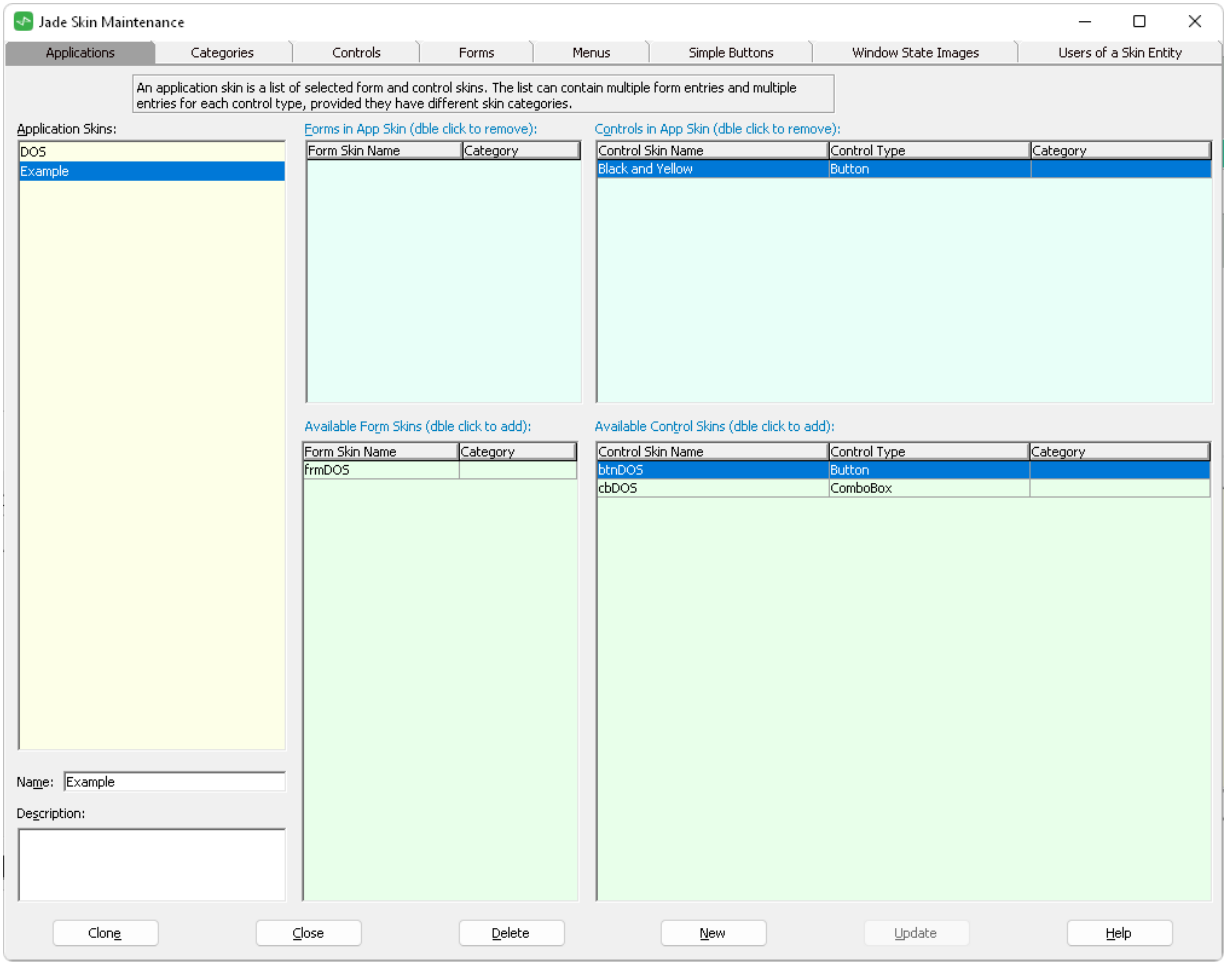
To open the Jade Skin Maintenance dialog, define and run the following **JadeScript** class **createJadeSkinMaintenance** method.

```
createJadeSkinMaintenance();

vars
    form : JadeSkinMaintenance;
begin
    create form transient;
    form.showModal();
epilog
    delete form;
end;
```

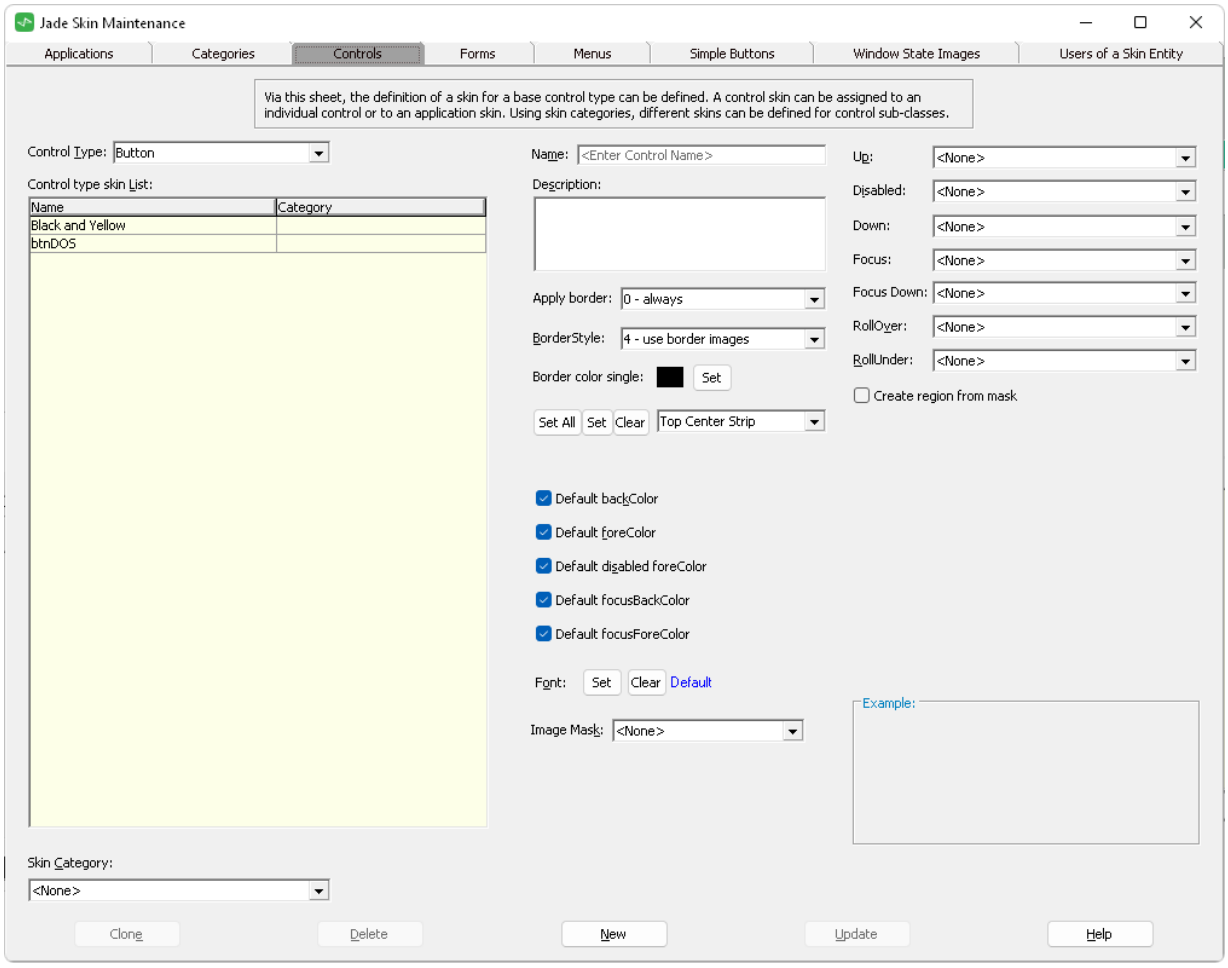
The first sheet of the Jade Skin Maintenance dialog is used to combine individual control re-skins into an application skin.

An application skin requires at least one **Form** or **Control** skin to be applied to it before it can be created.

As you create **Form** and **Control** skins, they are added to the lists of **Available Form Skins** and **Available Control Skins**.



To create a **Control** skin, select the **Controls** sheet of the Jade Skin Maintenance dialog.



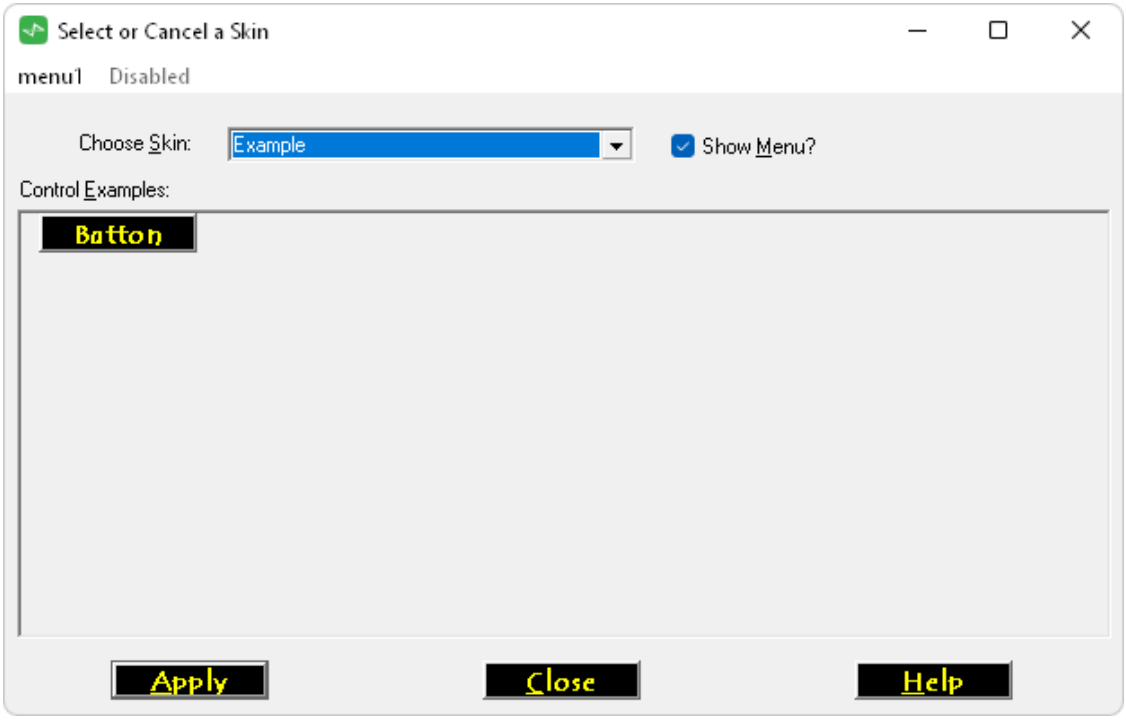
From this form, you can select a variety of options to be applied to all instances of a specific control. For example, you can select the color and font of the text of the button's label, the background color of the button, and the style of the border for the **Button** control type button. Alternatively, you can set an image for each of the parts of the **Button** control.

JadeSkinSelection Form

The **JadeSkinSelection** form (the Skin Selection dialog) is used to select the skin to be applied to the current application. The form can be created directly from the end-user application; for example, from a menu or the **click** event of a button. It can also be called from a **JadeScript** class method, to preview what a skin will look like. The form can be created with the following code.

```
createJadeSkinSelection();

vars
    form : JadeSkinSelection;
begin
    create form transient;
    form.showModal();
epilog
    delete form;
end;
```

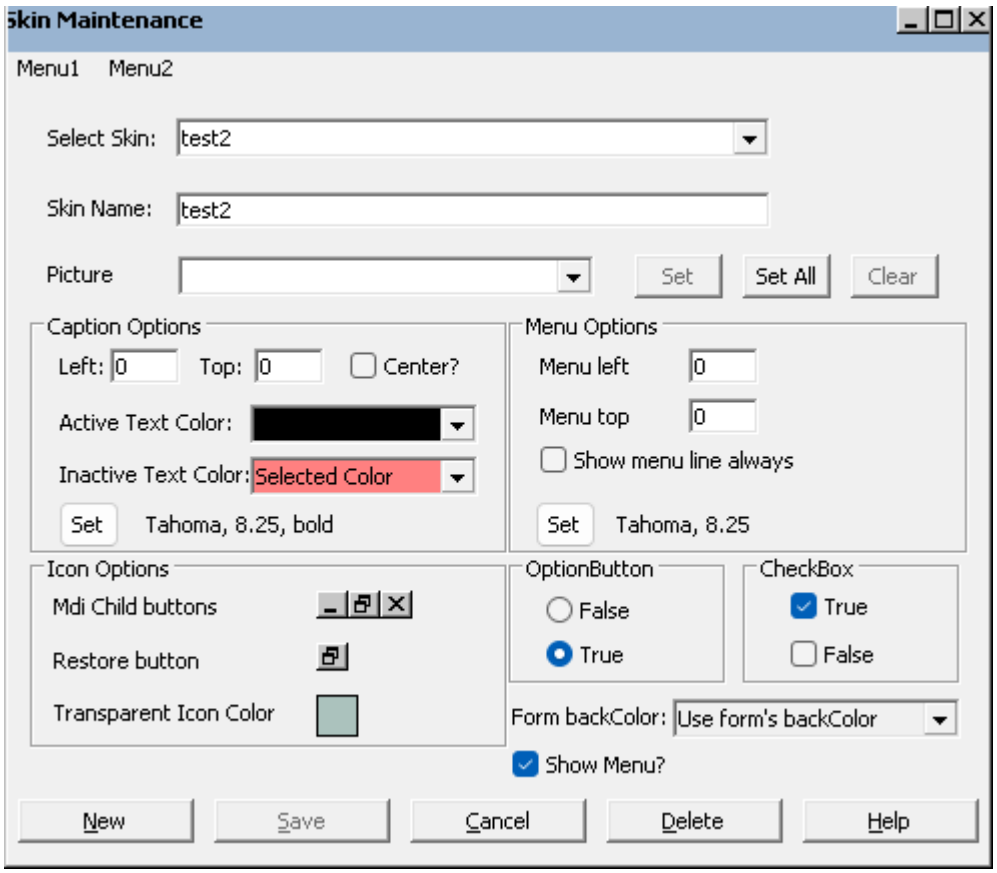
When viewing the Skin Selection dialog, the dialog itself is displayed using the selected skin. This allows the user to preview what the skin will look like before selecting a skin for use at run time.



In this simple example, the **Example** application skin has customized buttons with yellow text on a black background. The Skin Selection dialog displays its buttons in this style while **Example** is selected as the skin. Click **Apply**, to apply the **Example** skin to all application forms for the remainder of the application's lifetime.

JadeSkinMaint Form

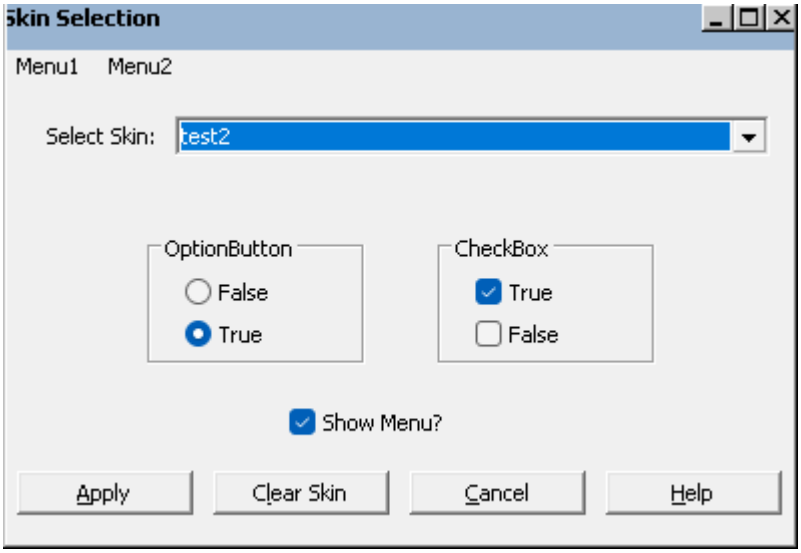
The **JadeSkinMaint** form (the Skin Maintenance dialog) was the precursor to the **JadeSkinMaintenance** form and it is retained in Jade for backwards compatibility.



Note We recommend that you use the Jade Skin Maintenance dialog rather than the Skin Maintenance dialog.

JadeSkinSelect Form

The **JadeSkinSelect** form was the precursor to the **JadeSkinSelection** form and it is retained in Jade for backwards compatibility.



Note We recommend that you use the **JadeSkinSelection** form rather that the **JadeSkinSelect** form.

Exercise 1 – Creating a Button Control Skin

In this exercise, create a simple application skin consisting of a few control skins.

1. Add the following **JadeScript** class **createJadeSkinMaintenance** method to **BankingViewSchema**.

```
createJadeSkinMaintenance();

vars
    form : JadeSkinMaintenance;
begin
    create form transient;
    form.showModal();
epilog
    delete form;
end;
```

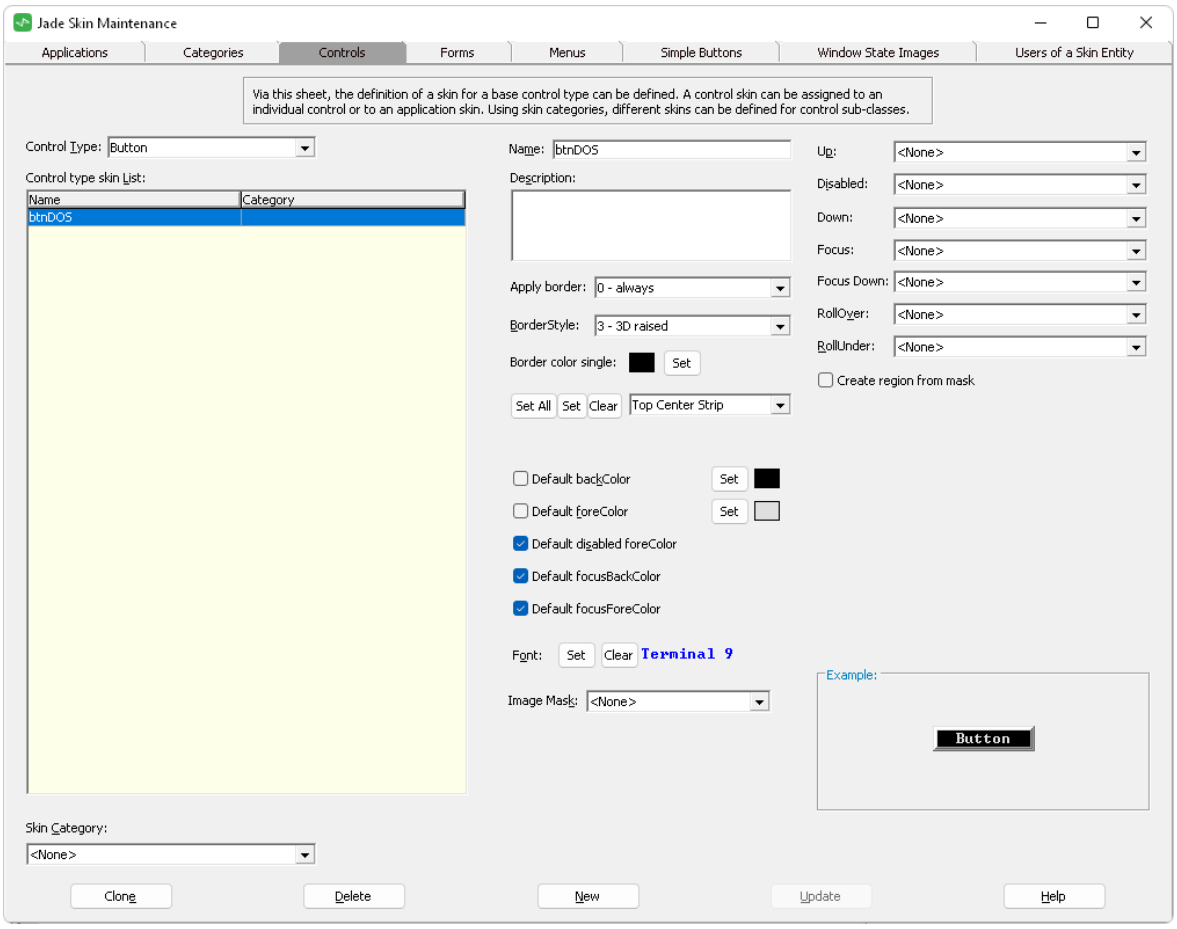
2. Run the method, to open the Jade Skin Maintenance dialog.
3. Select the **Controls** sheet and then select **Button** in the **Control Type** combo box.

Tip As the options in the following steps are examples only, feel free to customize the elements to your requirements.

4. Set **BorderStyle** to **3 - 3D** raised.
5. Uncheck the **Default backColor** check box and select **Black** as the color.
6. Uncheck the **Default foreColor** check box and select **Light Gray** as the color.
7. Set the font to **Terminal** and **Font Size** to **9**.

8. Call the skin **btnDOS**.

   The form should then look like the following.



9. Click **Update**, to save the **Button** skin.
10. Select the **Applications** sheet and double-click the **btnDOS** skin to add it to the current skin.
11. Enter **DOS** as the Application skin name and then click **Update**, to save it.

Exercise 2 – Previewing a Skin

In this exercise, use the **JadeSkinSelection** form to preview the changes to buttons made to your skin.

1. Add the following **JadeScript** class **createJadeSkinSelection** method to **BankingViewSchema**.

```
createJadeSkinSelection();

vars
    form : JadeSkinSelection;
begin
    create form transient;
    form.showModal();
epilog
    delete form;
end;
```

2. Run the method defined in the previous step and select **DOS** from the **Choose Skin** combo box.

You should see the buttons displayed using the skin you have designed in the first exercise in this module.

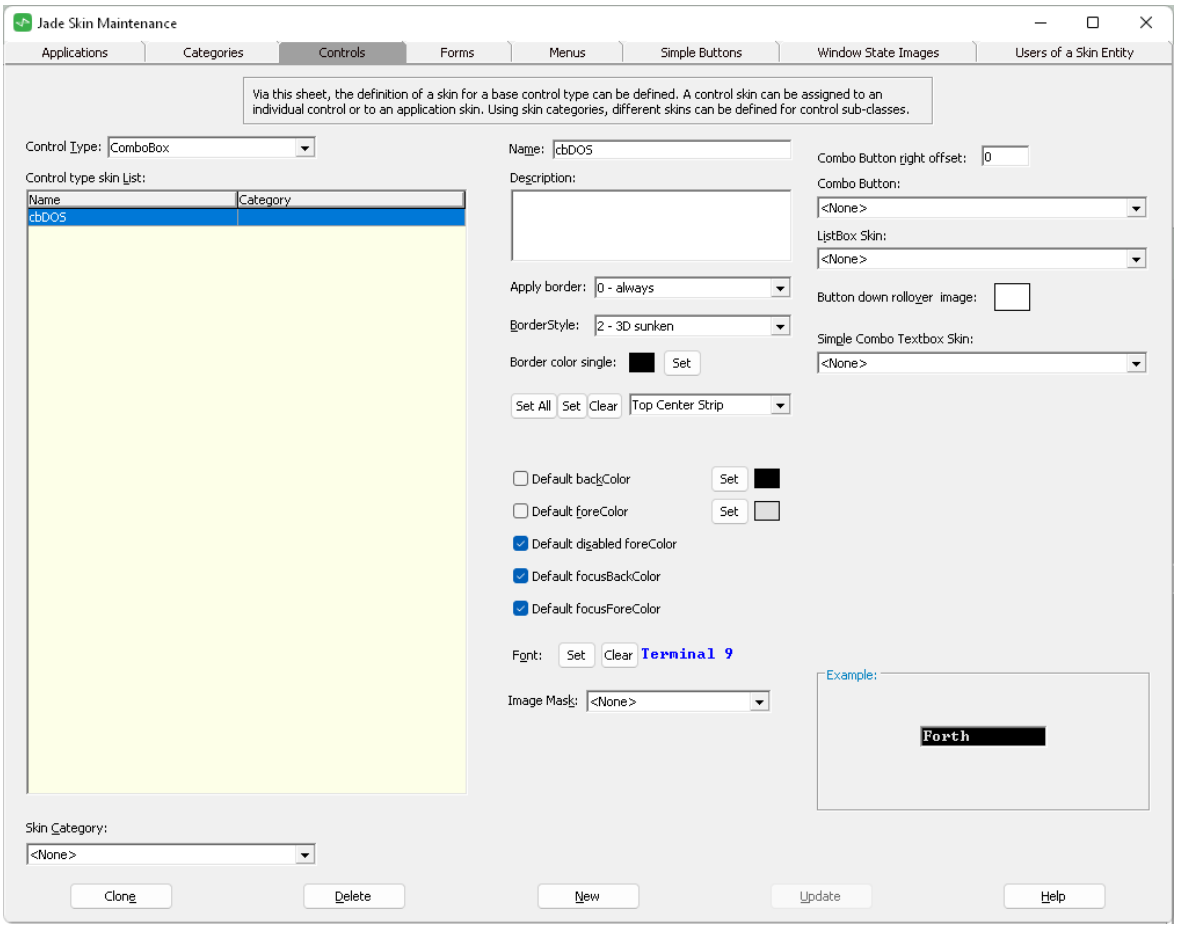
Exercise 3 – Creating a ComboBox Control Skin

In this exercise, add a control skin to combo boxes in the **DOS** application skin.

1. Open the Jade Skin Maintenance dialog by running the **JadeScript** class **createJadeSkinMaintenance** method.
2. Select the **Controls** sheet and then set the control type to **ComboBox**.
3. Set the **BorderStyle** to **2 - 3D** sunken.
4. Uncheck the **Default backColor** check box and select **Black** as the color.
5. Uncheck the **Default foreColor** check box and select **Light Gray** as the color.
6. Set the font to **Terminal** and the **Font Size** to **9**.

7. Call the skin **cbDOS**.

   The form should then look like the following.

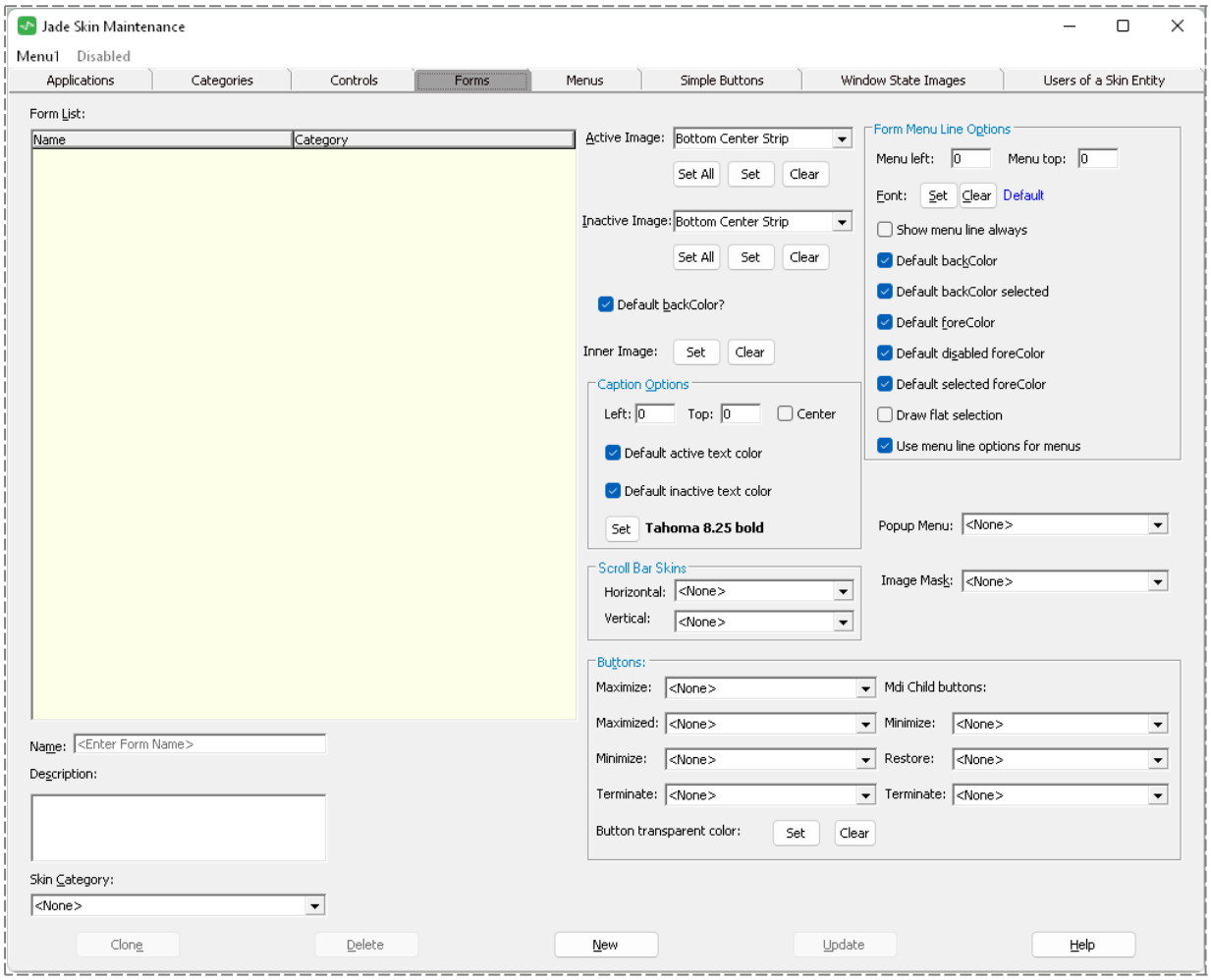


8. Click **Update**, to save the **ComboBox** skin changes.
9. Select the **Applications** sheet, add **cbDOS** to the **DOS** application skin, and then click **Update**.
10. Preview your changes in the **JadeSkinSelection** form as you did in the previous exercise.

The form should now look like the following.

Customizing Forms with Jade Skins

The **Forms** sheet of the Jade Skin Maintenance dialog is used to specify a forms skin, which applies to the forms of an application themselves, rather than the controls on that form.



This **Forms** sheet has the group boxes listed in the following table.

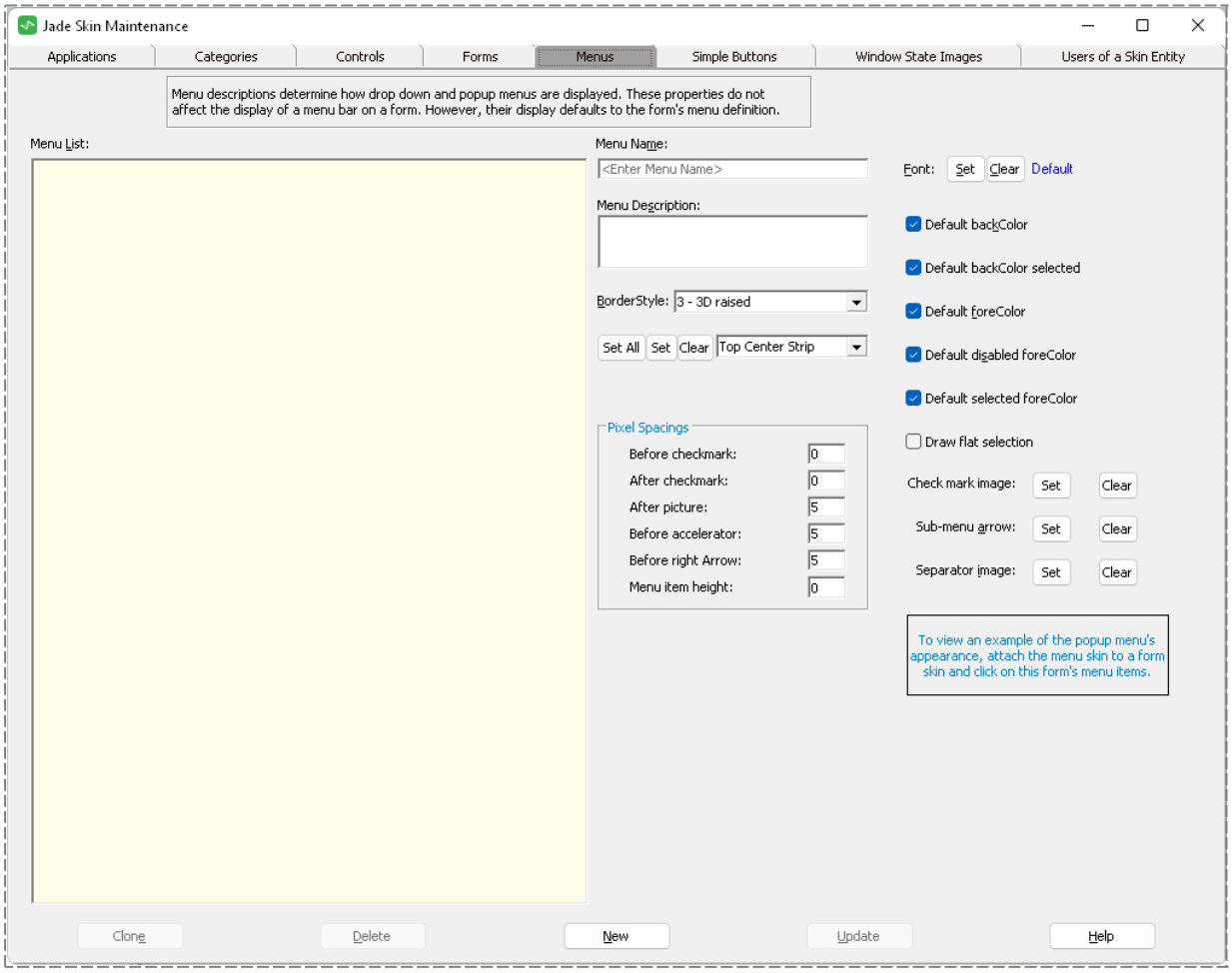
Group Box	Element to be Modified
Form Menu Line Options	The menu line at the top of forms. The drop-down menus also use this skin unless a menu skin is defined on the **Menu** sheet of the Jade Skin Maintenance dialog and applied to forms in the **Popup Menu** combo box.
Scroll Bar Skins	Specific horizontal and vertical scroll bar skins to be assigned to the Form skin and to specific **Control** class skins.
Caption Options	The title caption of the forms.
Buttons	The minimize, maximize, and close buttons of the forms.

The **Forms** sheet also has the following options that are not in group boxes.

Option	Element to be Modified
Active Image	The border elements of the forms. You can set an image for each one, to fully customize the look of forms.
Inactive Image	The border elements of the forms when they do not have focus. You can set an image for each one, to fully customize the look of forms.
Default backColor?	When unchecked, allows you to set a new default background color for the forms.
Inner Image	Displays the selected image as the background for forms.
Is brush	Displayed on the **Forms** sheet only when inner image is set. Checking this check box causes the inner image to be tiled on forms.
Popup Menu	The drop-down and popup menus. To use this option, you must first create a menu skin on the **Menus** sheet of the Jade Skin Maintenance dialog.
Image Mask	Allows for a non-rectangular region mask image to be applied to the forms' skin.

Customizing a Menu with Jade Skins

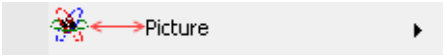
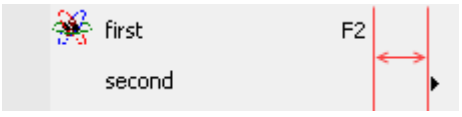
The **Menus** sheet of the Jade Skin Maintenance dialog is used to apply a skin to any drop-down and popup menus on forms. Menu skins do not apply to the menu line of forms, but the menu line's skin is used as the default menu skin until it is set.



The **Menus** sheet has the following options.

Option	Description
BorderStyle	Controls the borders between menu items. You can choose **none**, **single**, **3D sunken**, **3D raised**, or **use border images**.
Set All / Set / Clear	Allows for the setting of image files as the borders between menu items when **BorderStyle** is set to **4 - use border images**.
Font	Sets the font (typeface) of the text in menu items.
Default backColor	Sets the default background color for menu items.
Default backColor selected	Sets the default background color for how menu items are displayed when they are selected.
Default foreColor	Sets the default text color for menu items.
Default disabled foreColor	Sets the default text color for disabled menu items.
Default selected foreColor	Sets the default text color for how menu items are displayed when they are selected.
Draw flat selection	When checked, selected menu items are drawn flat regardless of the border style that is selected. In addition, if the border style is set to **none** and multiple menu items are selected, no line is drawn between them.
Check mark image	Sets the image to be used for check marks in menu items.
Sub-menu arrow	Sets the image to be used to show that a menu item contains a submenu.
Separator image	Sets the image to be used for any separators between menu items.

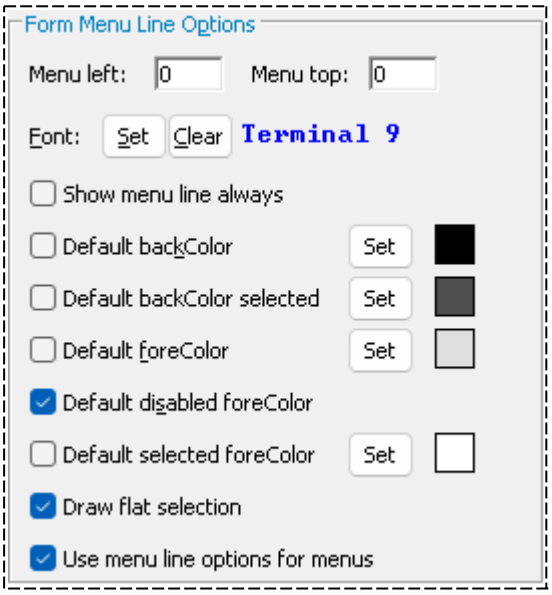
The Pixel Spacings group box on **Menus** sheet has the following text boxes, which enable you to specify the number of pixels used for spacing.

Check Box	Example
Before checkmark	
After checkmark	
After picture	
Before accelerator	
Before right Arrow	
Menu item height	

Exercise 4 – Adding a Forms Skin to an Application Skin

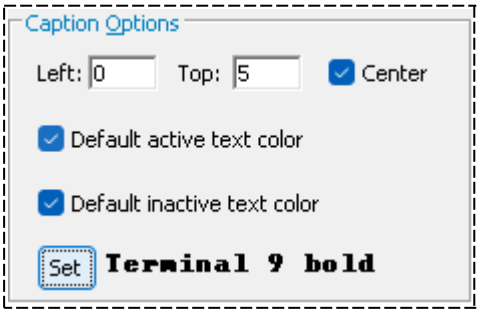
In this exercise, add a forms skin to your DOS application skin.

1. Run the **JadeScript** class **createJadeSkinMaintenance** method to open the Jade Skin Maintenance dialog.
2. Select the **Forms** sheet on the Jade Skin Maintenance dialog.
3. Set the **Default backColor** to **Light Gray**.
4. In the Form Menu Line Options group box, set the following values.



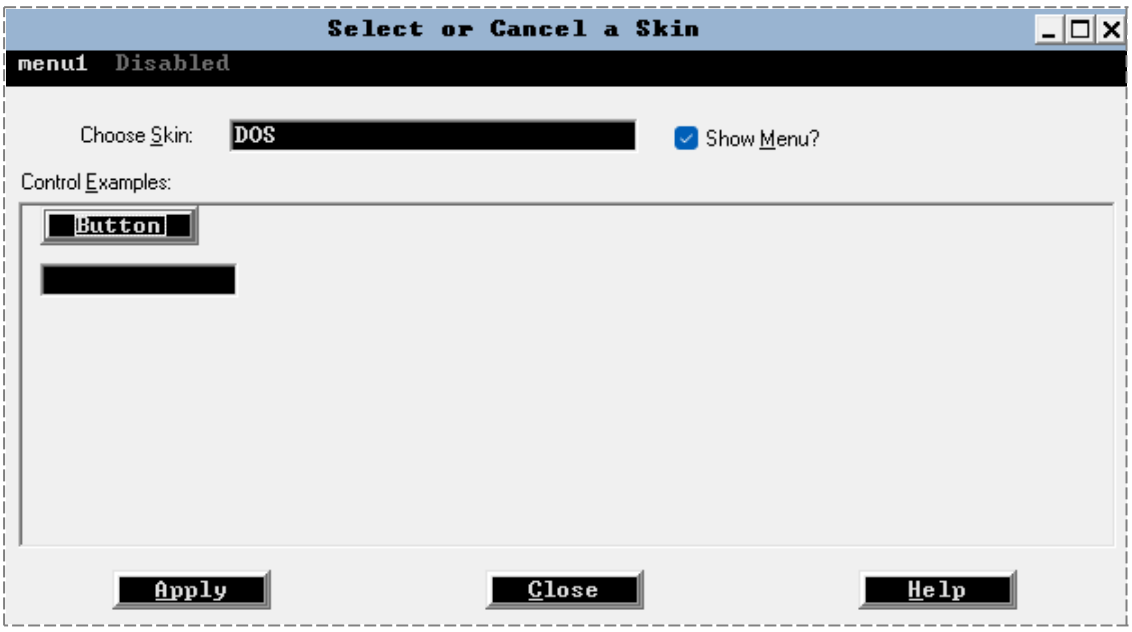
   a. **Font** to **Terminal**, size **9**
   b. **Default backColor to Black**
   c. **Default backColor selected to Dark Gray**
   d. **Default foreColor** to **Light Gray**
   e. **Default selected foreColor** to **White**
   f. Check the **Draw flat selection** check box

5. In the Caption Options group box, set the following values.



   a. **Top** margin to **5**
   b. Check the **Center** check box
   c. **Font** to **Terminal**, size **9**, **bold**

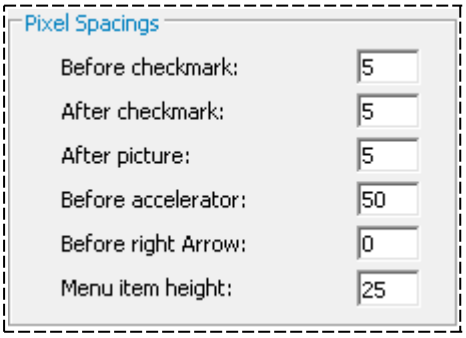
6. Save the forms skin as **frmDOS**.
7. Preview the skin using the **JadeScript** class **JadeSkinSelection** method. It should look like the following.



Exercise 5 – Adding a Menus Skin to an Application Skin

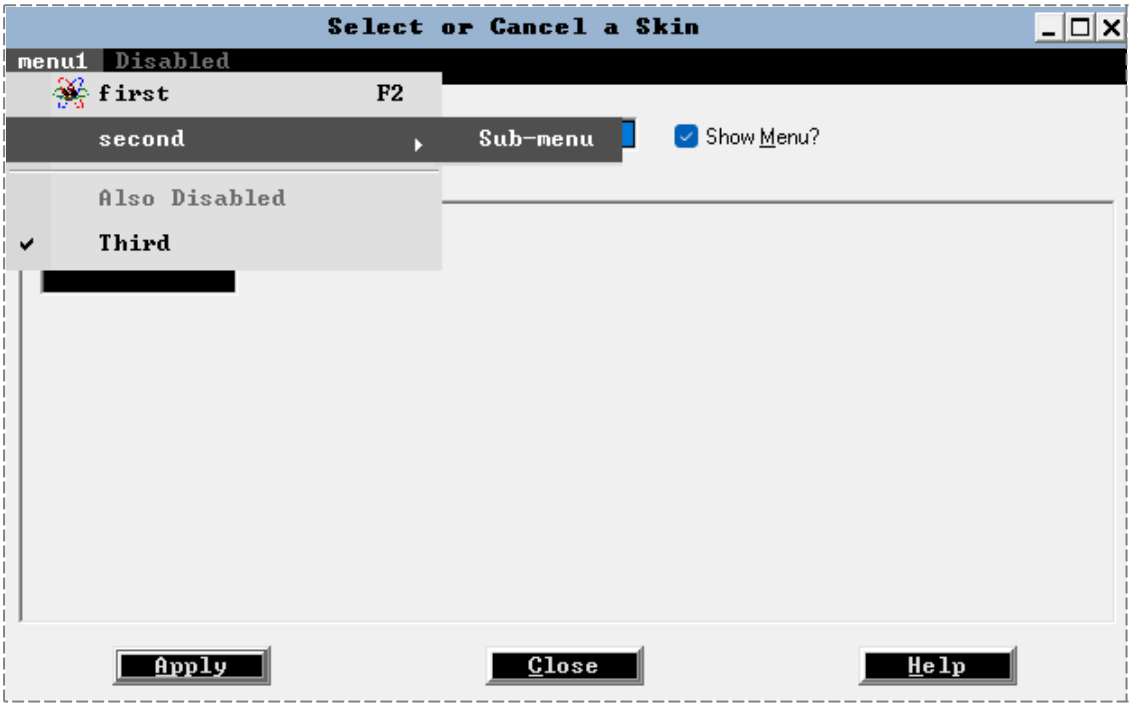
In this exercise, customize the menus skin for the DOS application skin.

1. Run the **JadeScript** class **createJadeSkinMaintenance** method to open the Jade Skin Maintenance dialog.
2. Select the **Menus** sheet on the Jade Skin Maintenance dialog.
3. Set the **BorderStyle** to **0 - none**.
4. In the Pixel Spacings group box, set the values shown in the following image.



5. Set the **Font** to **Terminal**, size **9**.
6. Set the **Default backColor** to **Light Gray**.
7. Set the **Default backColor** selected to **Dark Gray**.
8. Set the **Default foreColor** to **Black**.
9. Set the **Default selected foreColor** to **White**.

10. Check the **Draw flat selection** check box.
11. Save the menus skin as **mnuDOS**.
12. On the **Forms** sheet of the Jade Skin Maintenance dialog, set the **Popup Menu** option to **mnuDOS**.
13. Preview the skin using the **JadeScript** class **JadeSkinSelection** method. It should look like the following.



JadeSkinRoot Class

JadeSkinRoot is the root object that contains collections of all skins (including **Application** skins as well as the **Control** and **Forms** skins that make up an application skin).

As **JadeSkinRoot** is a root object that uses the singleton pattern, it can be accessed safely with **JadeSkinRoot.firstInstance**.

Note The singleton pattern that is used for root objects means that "There can be only one" is enforced. For more details, see "Module 7 - Root Object" of the Jade Platform Developer's course.

The **JadeSkinRoot** class can be used to find specific skins with a dictionary lookup by name, as shown in the following example.

```
initialize() updating;

vars
    jsRoot  : JadeSkinRoot;
    exampleSkin : JadeSkinApplication;

begin
    jsRoot  := JadeSkinRoot.firstInstance;
    exampleSkin := jsRoot.allApplicationSkins["Example"];
    app.setApplicationSkin(exampleSkin);
end;
```

This method sets the application's skin to the skin called **Example** when the application is initialized.

Exercise 6 – Adding an Application Skin to an Application

In this exercise, apply the **DOS** application skin to the **Banking** application.

1. Open **BankingModelSchema** in the Class Browser.
2. Navigate to the **BankingModelSchema** application subclass.

3. Add a method called **applySkin** to the **BankingModelSchema** application class and code it as follows.

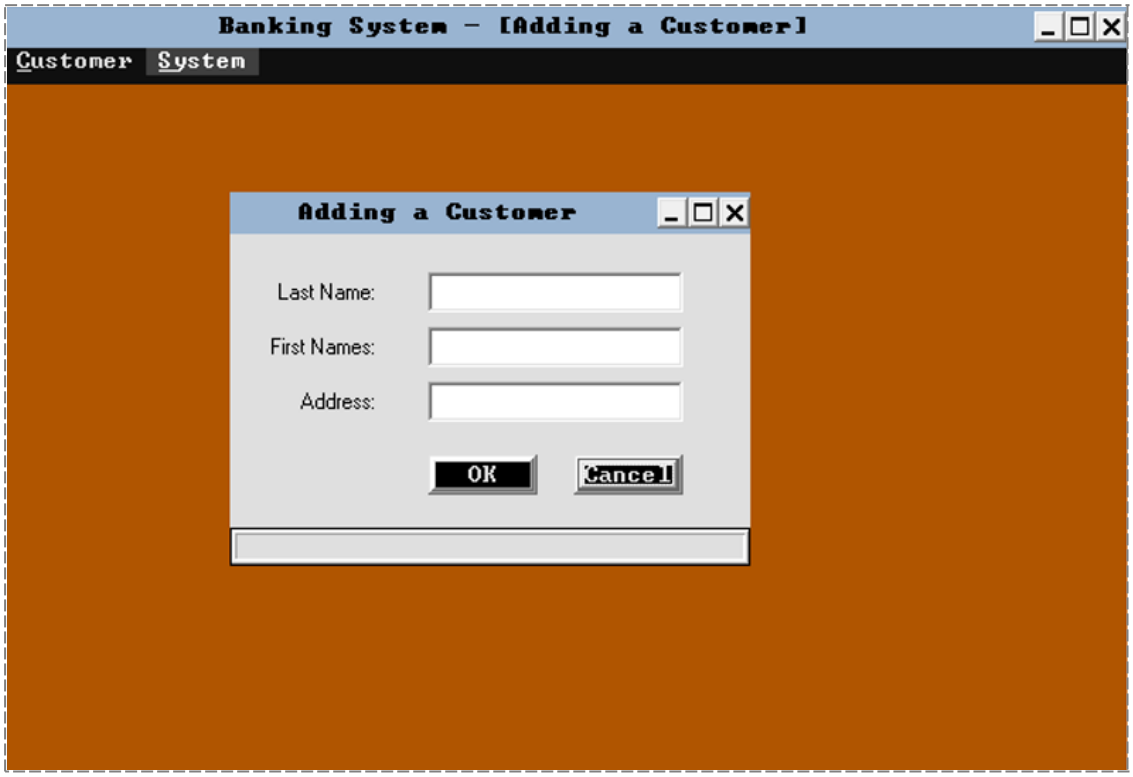
```
applySkin();

vars
    jsRoot  : JadeSkinRoot;
    skin    : JadeSkinApplication;
begin
    jsRoot  := JadeSkinRoot.firstInstance;
    skin    := jsRoot.allApplicationSkins["DOS"];
    app.setApplicationSkin(skin);
end;
```

4. Modify the **initialize** method in the **BankingModelSchema** application class to call the **applySkin** method, as follows.

```
initialize() updating;

begin
    self.applySkin();

    on Exception do self.genericExceptionHandler(exception) global;
    self.myBank := Bank.firstInstance();
    if self.myBank = null then
        beginTransaction;
        create myBank persistent;
        commitTransaction;
    endif;
end;
```

5. Run the **Banking** application and open the Add Customer dialog.

The **Banking** system should have the **DOS** application skin applied to it.



Logical Certifier

The Logical Certifier is a tool for verifying and, if needed, repairing the referential integrity of a Jade database.

Typically, Jade will automatically maintain referential integrity when an inverse relationship is set up within a Jade database and as such, the most common usage of the Logical Certifier is simply to verify that this has been done correctly.

In the unlikely event that somehow an inverse has not been maintained correctly, the Logical Certifier can generate fixes for the various integrity issues an inverse relationship could have.

Referential Integrity

A Jade inverse reference defines a relationship contract between two classes that is enforced on the objects of those classes.

These relationships, which can be one-to-one, one-to-many, or many-to-many, cause Jade to automatically make any references between objects of the contracted classes mutual. For example, consider two classes, **Company** and **Product**, which have the following relationship.

- A **Company** has many **Products**, held in an **allProducts** collection.
- A **Product** has one **Company**, stored in a **myCompany** reference.

As such, we can say that there is a one-to-many relationship between **Company** and **Product**.

With an inverse relationship, we can set a **Company** to automatically add a **Product** to its **allProducts** collection whenever that **Product** sets the **Company** to its **myCompany** reference (or the reverse). With such a relationship set, we can safely say that if a specified **Product** has its **myCompany** set, it is definitely present in the **allProducts** collection of that **Company**.

Broken Referential Integrity

As the use of inverses is both very useful and very reliable, it is common to rely heavily on referential integrity being correct.

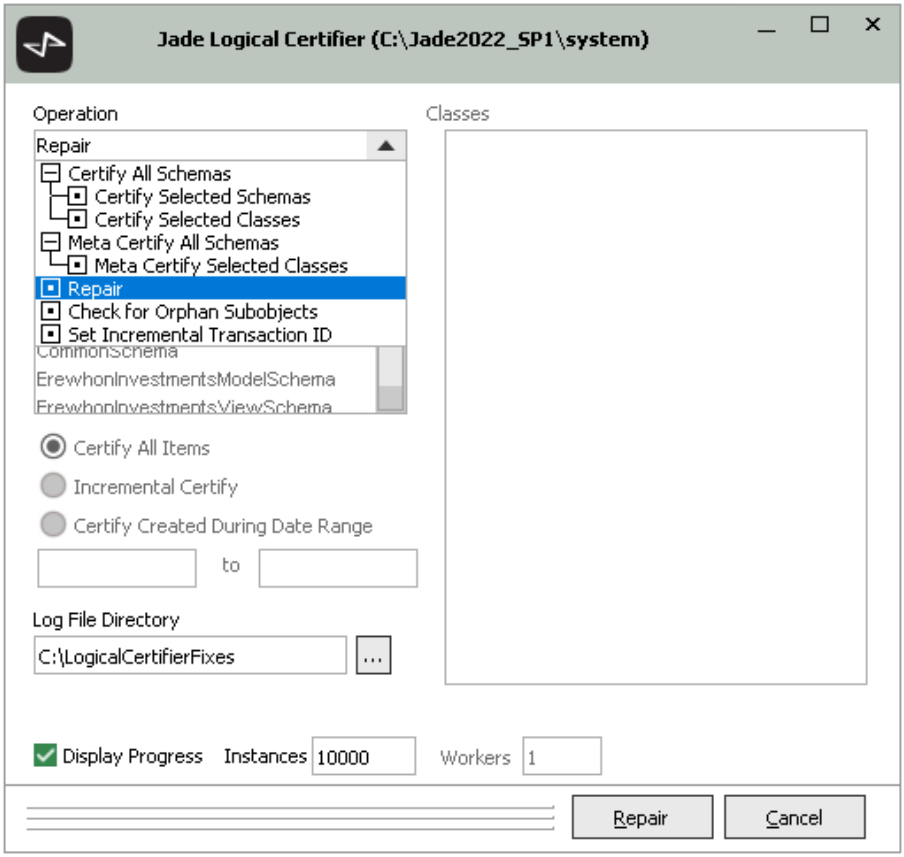
While it is rare for referential integrity to break, any issues have the potential to result in severe consequences; for example, they could result in the accessing of wrong data, the inability to access data at all, or runtime exceptions being generated.

The impact of these events varies from application to application, but it can be significant; for example, if a customer's details end up stored at the wrong key, another customer could inadvertently gain access to his or her account. As such, even though it is unlikely to see any issue, you should regularly perform certification on user data in production systems.

The Logical Certifier

The Logical Certifier is a RootSchema application that you can open by selecting **RootSchema** in the Schema Browser and then right-clicking the **Run Application** toolbar button.

The Jade Logical Certifier dialog is then displayed.



The Logical Certifier can perform the following functions.

Function	Performs…
Certify All Schemas	A certify on the user data within all schemas of the database and saves the results to the specified Log File Directory.
Certify Selected Schemas	A certify on the user data within the selected schema and saves the results to the specified Log File Directory.
Certify Selected Classes	A certify on the user data within the selected classes and saves the results to the specified Log File Directory.
Meta Certify All Schemas	A certify on the metadata of all schemas of the database and saves the results to the specified Log File Directory.
Meta Certify Selected Classes	A certify on the metadata of the selected classes and saves the results to the specified Log File Directory.
Repair	The actions prescribed in the **_logcert.fix** file within the specified Log File Directory.
Check for Orphan Subobjects	A check for orphan subobjects in user data files within all schemas of the database and saves the results to the specified Log File Directory.
Set Incremental Transaction ID	Manually sets the transaction ID that the Jade Logical Certifier uses during an incremental certify operation.

Exercise 1 – Creating a Throwaway Schema

In this exercise, create a simple schema with an inverse relationship between two classes so that low-value data intentionally breaks in the next exercise.

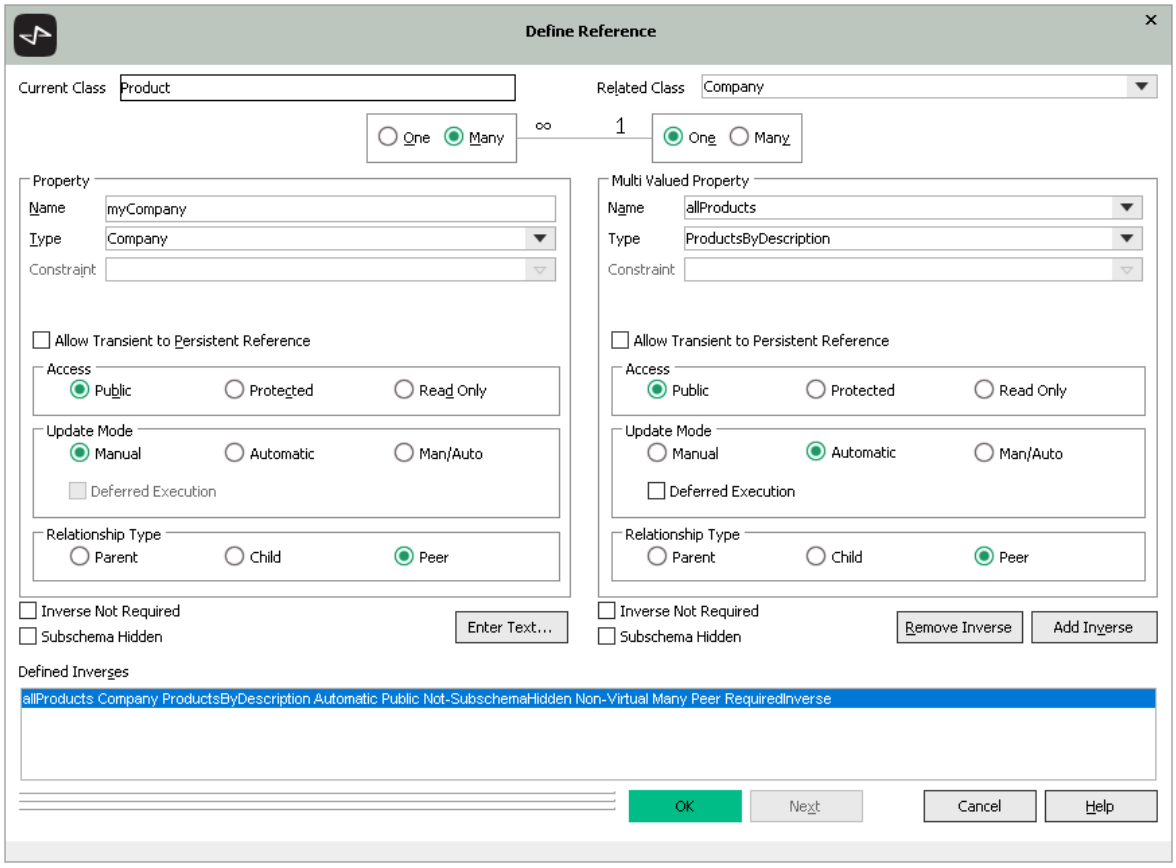
1. Create a schema called **LogCertTester**, with two classes called **Company** and **Product**.

   Caution While we will only be intentionally breaking *this* schema, if you have important data in another schema, it may pay to take a backup or start from a fresh Jade database, just in case.

2. In the **Product** class, define a **public** property called **price** of type **Integer**, and a **public** property called **description** of type **String**.

3. Create a subclass of **MemberKeyDictionary** called **ProductsByDescription**, with membership **Product** and the key of **description**.

4. In the **Product** class, define an inverse relationship with **Company**, as follows.

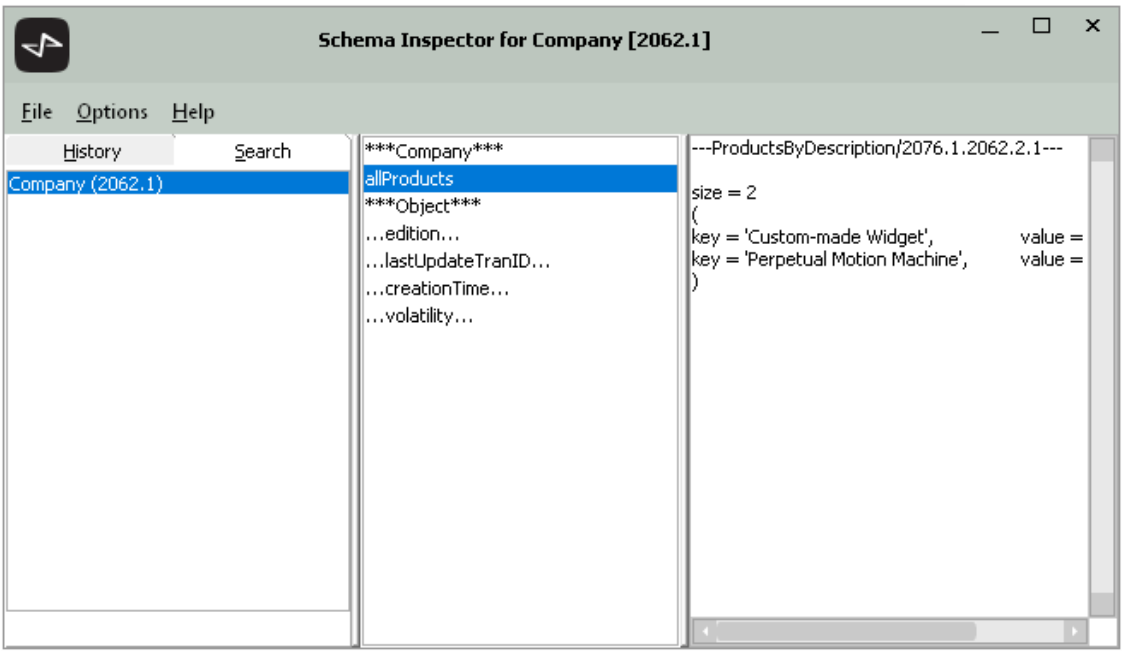


5. Create a **JadeScript** class method called **setup** and code it as follows.

```
setup();

vars
    product : Product;
    company : Company;
begin
    beginTransaction;
    create company persistent;
    commitTransaction;

    beginTransaction;
    create product persistent;
    product.description := "Custom-made Widget";
    product.price := 42;
    product.myCompany := company;
    commitTransaction;

    beginTransaction;
    create product persistent;
    product.description := "Perpetual Motion Machine";
    product.price := 350;
    product.myCompany := company;
    commitTransaction;
end;
```

6. Run the method, checking that the referential integrity has been established by viewing **Company** in the Schema Inspector.

   The **Company** should have both **Products** automatically added to its **allProducts** collection.



Tip Use the Ctrl+I shortcut keys to open a class in the Schema Inspector.

Exercise 2 – Breaking Referential Integrity

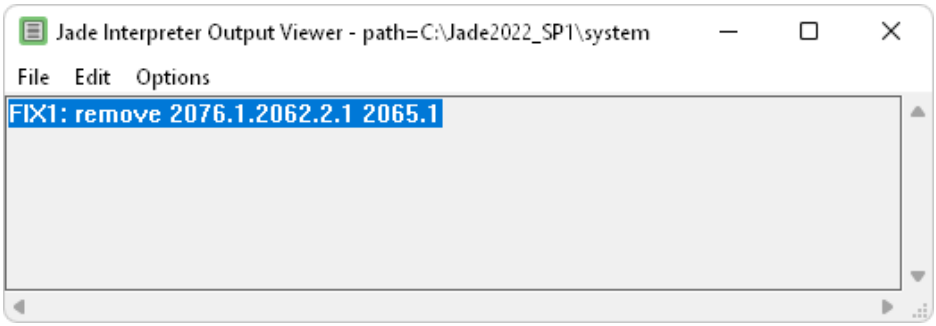
In this exercise, intentionally break the referential integrity of the **allProducts**/**myCompany** inverse, by removing one of the **Products** from the **allProducts** collection.

Normally this would not be possible, as Jade will prevent you from manually modifying an automatic collection involved in an inverse relationship (to ensure referential integrity). However, we can get around this by using the Logical Certifier itself to modify the collection.

1. Create a **JadeScript** class method called **createBreakingFix** and code it as follows.

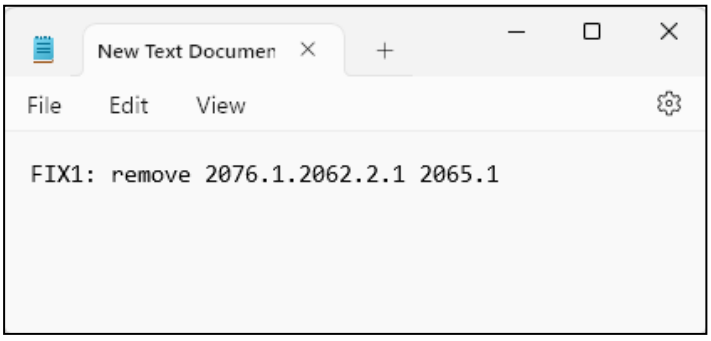
```
createBreakingFix();

vars
    fixLine : String;
begin
    //Remove first instance of Product from company's collection of products
    fixLine := "FIX1: remove ";
    fixLine := fixLine & ProductsByDescription.number.String;
    fixLine := fixLine & ".1." & Company.number.String;
    fixLine := fixLine & ".2.1 ";
    fixLine := fixLine & Product.firstInstance.getOidString;
    write fixLine;
end;
```

2. Run the method and copy the result from the Jade Interpreter Output Viewer to the clipboard.

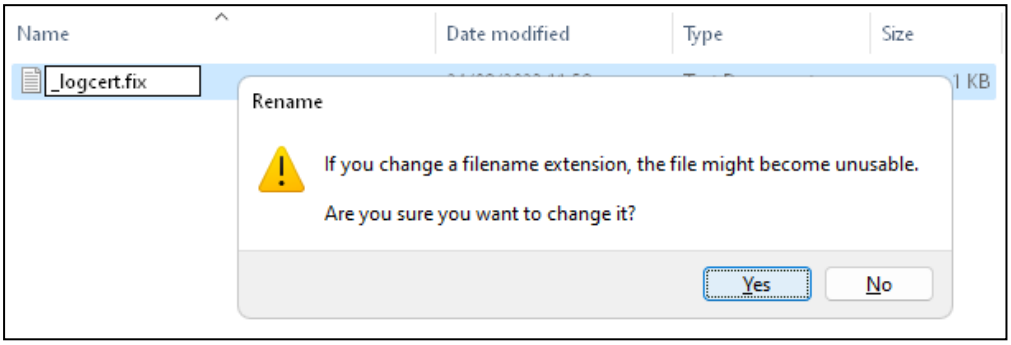


3. In your file system, create a folder called **C:\LogicalCertifierFixes**.

4. Create a text document in this folder, paste the result from the **createBreakingFix** method into it, and then save it.



5. Rename the file to **_logcert.fix**.

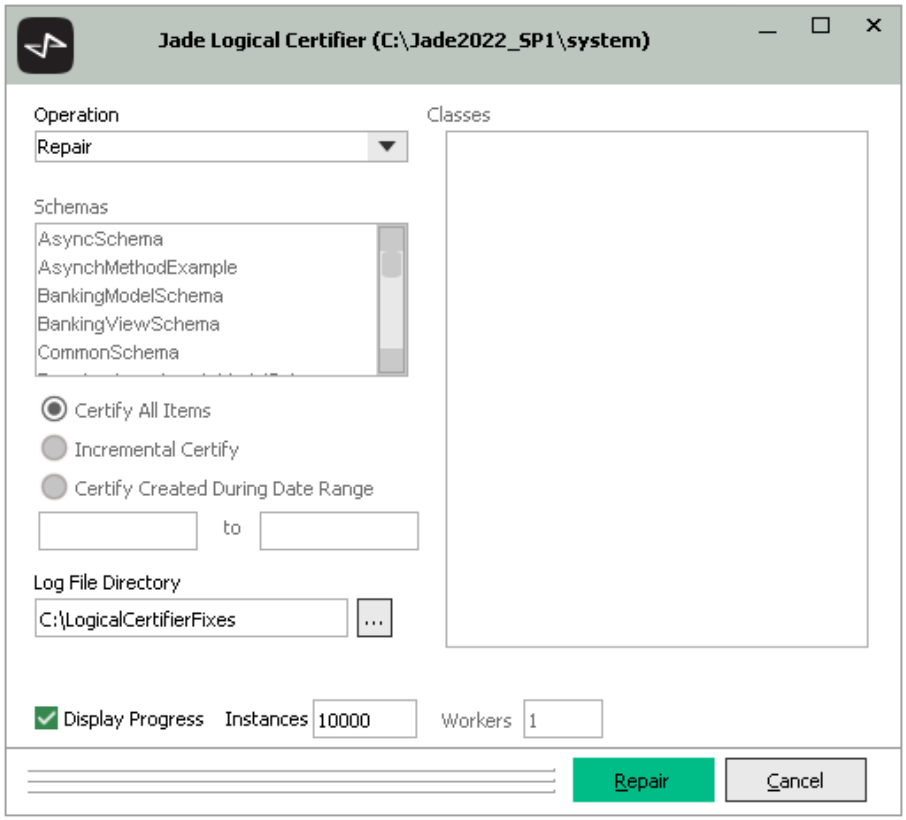


Click **Yes**, to change the file name extension.

Note You would never normally manually create a **_logcert.fix** file, as they are generated by the certify functions of the Logical Certifier. However, in this exercise, we are actively trying to damage our database.

6. In your Jade Platform development environment, select **RootSchema** in the Schema Browser and open the Logical Certifier by right-clicking the **Run Application** toolbar button.

7. Select **Repair** in the **Operation** combo box, enter **C:\LogicalCertifierFixes** in the **Log File Directory** text box, and then click **Repair**.



8. Inspect the **Company** object in the Schema Inspector. Note that although there is now one entry only in the **allProducts** collection, both **Product** objects still have a reference to **myCompany**.

Fixing Missing Items in Collections

One way that an inverse relationship can be broken is if there is a one-to-many relationship between two classes (for example, a **Company** and its **Products**) where a product holds a reference to its **Company** but that Company does not have the **Product** in its collection.

The way that the Logical Certifier fixes this depends on the update mode of the inverse relationship; that is, which class is to be manually updated and which is automatically updated (or if there is a man/auto relationship, where either class can be updated).

The assumption that the Logical Certifier makes is that the manually updated class is correct and the automatically updating class should be the one to change to match. As such, it performs the following fixes for the following cases.

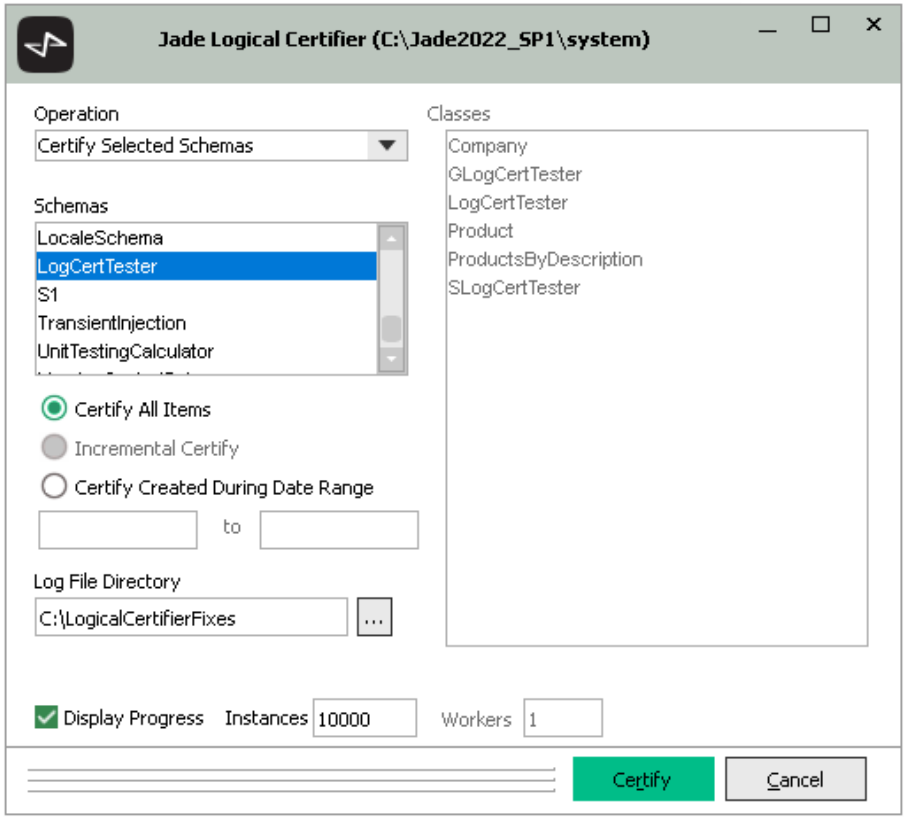
Case	Fix
Manual one to automatic many	Adds the missing object to the collection.
Automatic one to manual many	Keeps the collection as-is and removes the reference to the "one" on the "many" class.
Man/Auto setting	Generates both of the above, but comments them out. You must manually uncomment the fix that is to be done.

Exercise 3 – Repairing Referential Integrity

In this exercise, use the Logical Certifier to repair the damage done in the previous exercise.

1. Open the Logical Certifier.

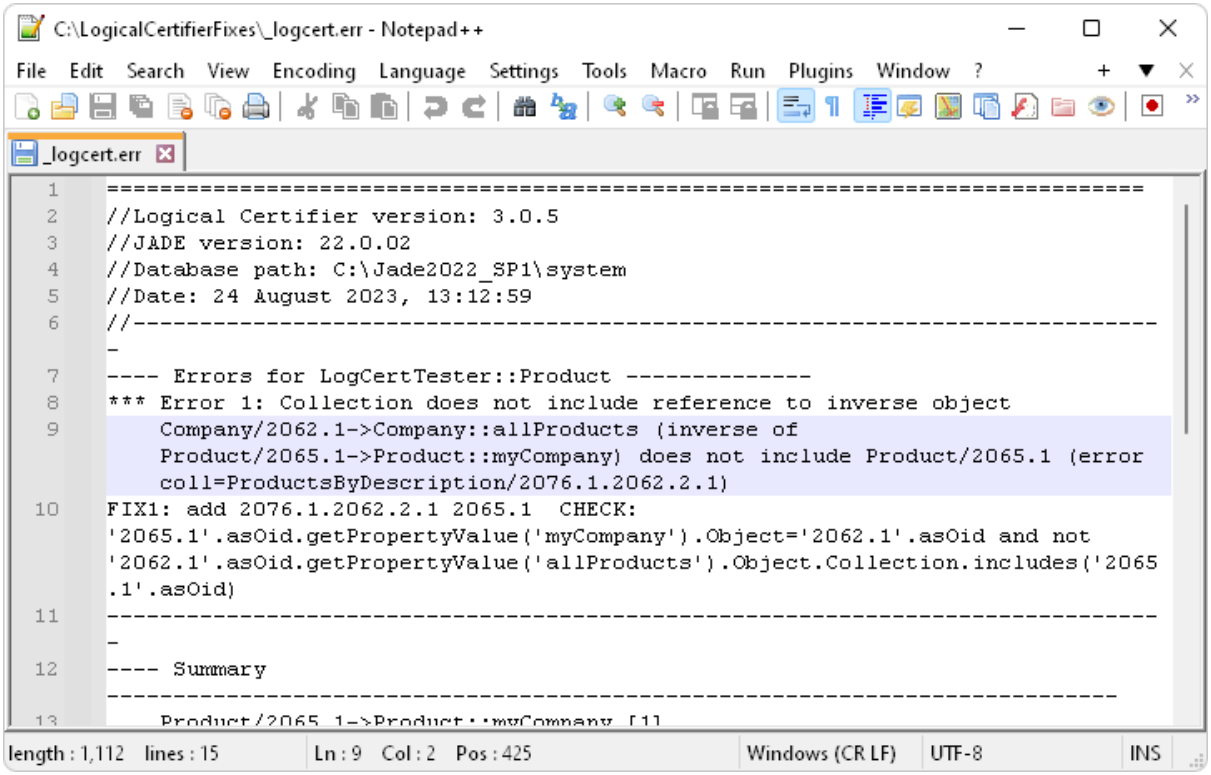
   Select **Certify Selected Schemas** in the **Operation** combo box, select **LogCertTester** in the **Schema** list box, and then enter **C:\LogicalCertifierFixes** in the **Log File Directory** text box.



2. Click **Certify** and **Yes** when prompted to confirm that you want to replace the existing file.

The Certify message box should then be displayed, advising you that the certify operation completed with one error. Click **OK**.

3. To see what error was detected, open the **_logcert.err** file in Notepad++.



4. From the Logical Certifier, select **Repair** in the **Operation** combo box and enter **C:\LogicalCertifierFixes** in the **Log File Directory** text box.

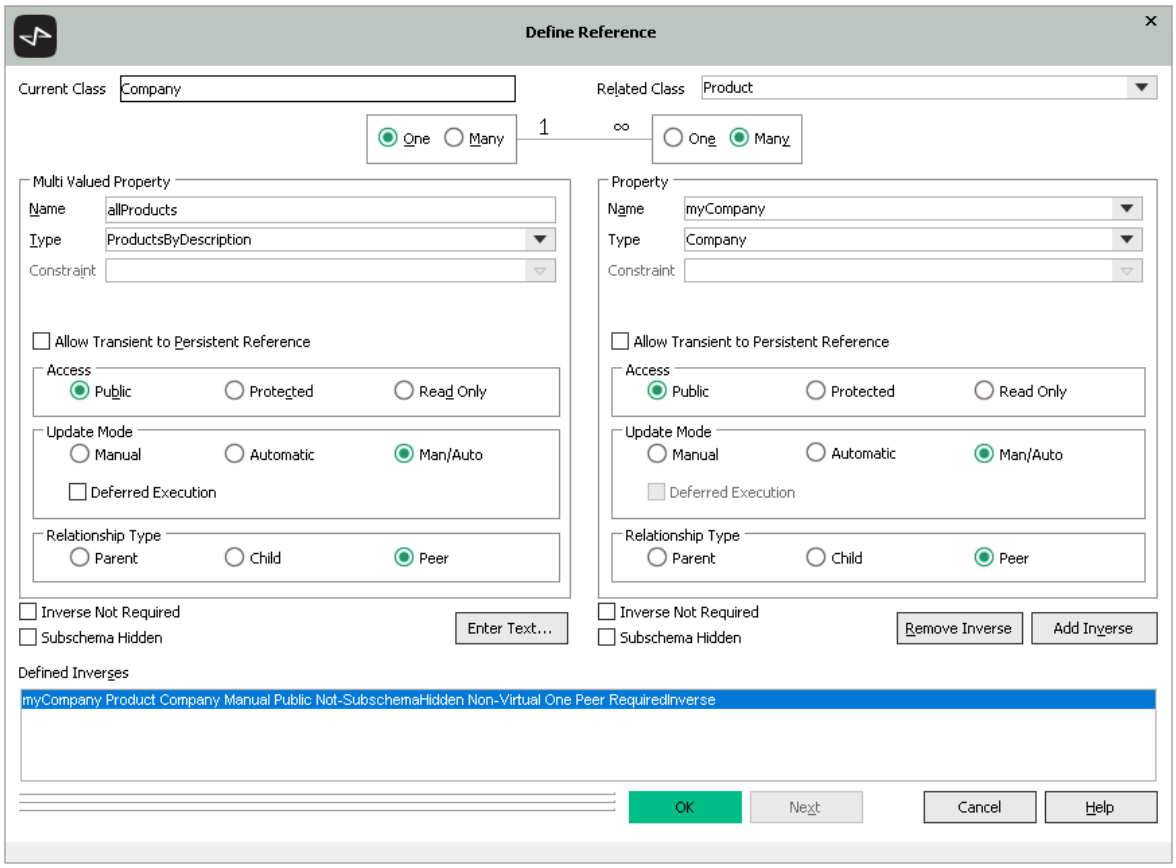
5. Click **Yes** on the two subsequent confirmation dialogs.

6. Open **Company** in the Schema Inspector and then select the **allProducts** collection. You should see that there are now two **Products** in the collection again.

Exercise 4 – Fixing an Ambiguous Referential Integrity Error

In this exercise, set the **Product** / **Customer** relationship to **Man/Auto** and then generate another referential integrity error.

1. Modify the **allProducts** relationship in **Company** to set the update mode to **Man/Auto**, as follows.



2. Run the **JadeScript** class **createBreakingFix** method and then copy the result to the clipboard.

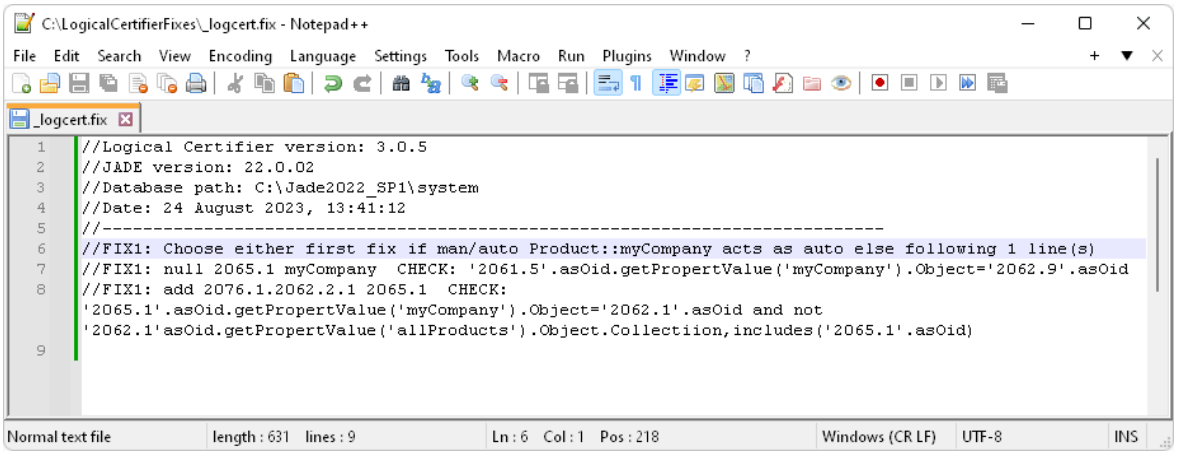
3. Open the **_logcert.fix** file in the **C:\LogicalCertifierFixes** directory using Notepad++, and replace the contents with the result of the **JadeScript** class **createBreakingFix** method that you copied to the clipboard. (Don't forget to save it.)

4. Check that the **allProducts** collection in **Company** now contains one product only (using the Schema Inspector).

5. Select **Certify Selected Schemas** in the **Operation** combo box and enter **C:\LogicalCertifierFixes** in the **Log File Directory** text box of the Jade Logical Certifier dialog.

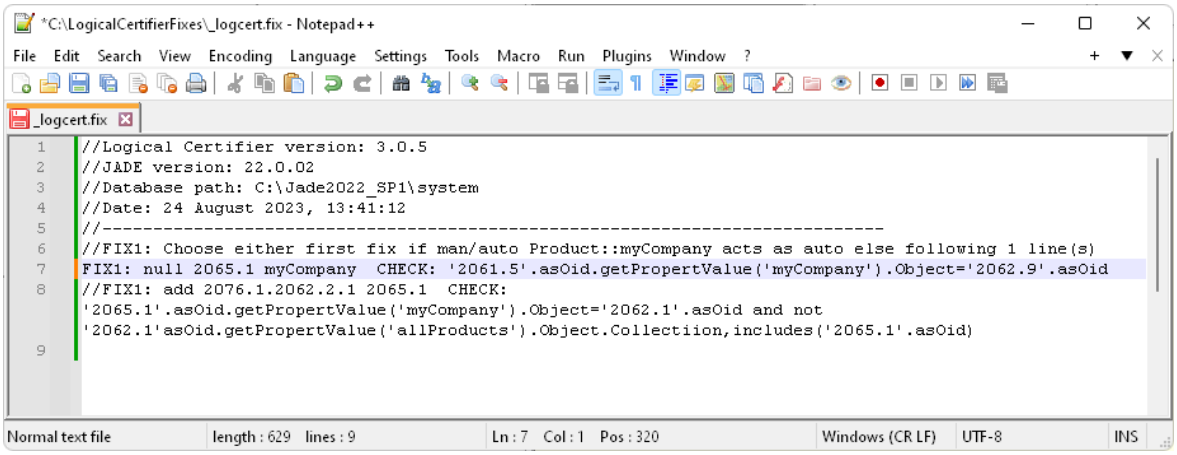
6. Open the **_logcert.fix** file in the **C:\LogicalCertifierFixes** directory using Notepad++.

You will see that two fixes have been proposed; that is, setting **myCompany** to null for the **Product** or adding the **Product** to the **allProducts** collection.

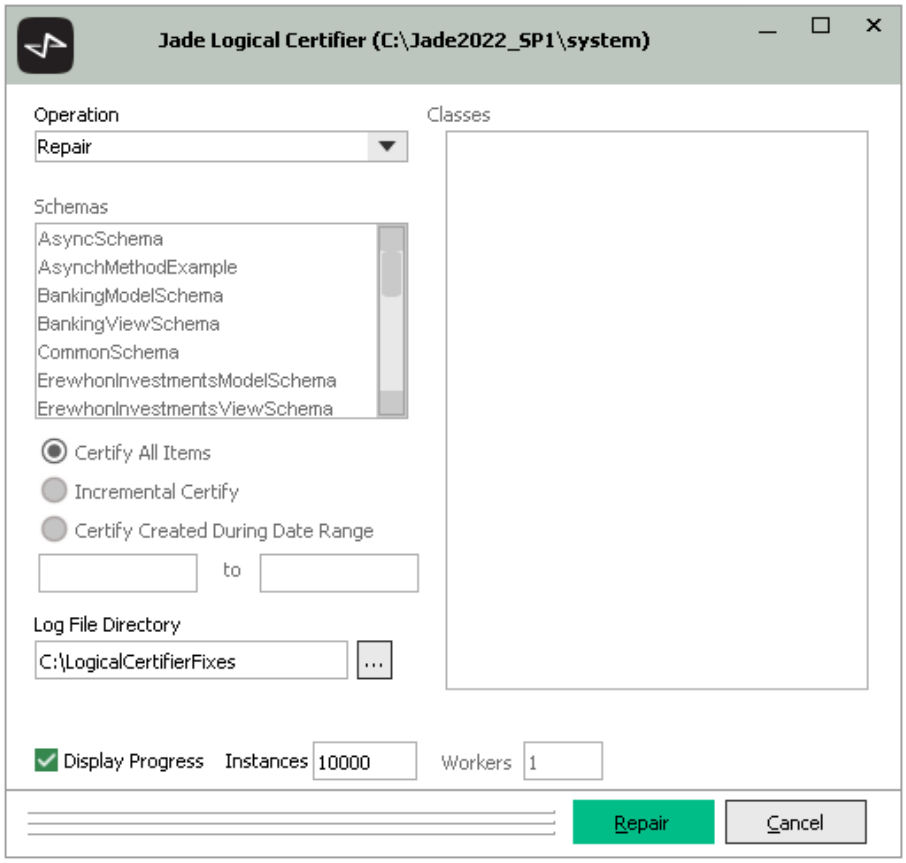


7. To select one of these, simply delete the preceding *//* to uncomment the fix.

   For this exercise, delete the *//* preceding the first proposed fix, as follows.

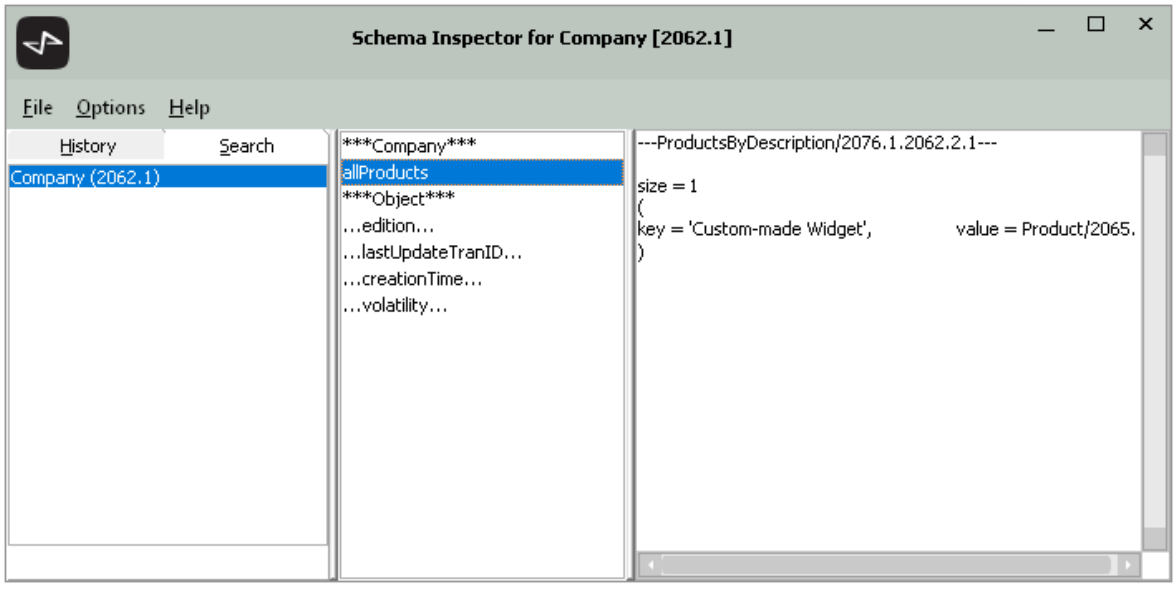


8. Run the **Repair** operation in the Jade Logical Certifier dialog, with **C:\LogicalCertifierFixes** specified in the **Log File Directory** text box.



You will see that it generates no errors, but two warnings. This is simply because the other fix was ignored.

9. Inspect the **allProducts** collection of the **Company**. You will see that it still only has one **Product** in it.



10. Inspect the first **Product** in the Schema Inspector.

You will see that the **myCompany** reference is a **<null> object reference**; that is, the **Product** has no **Company**.

Handling Invalid Member Keys

One of the most damaging things that can happen to a **MemberKeyDictionary** is when there is a mismatch between the key of an entry and the object referenced by that entry.

This will typically happen only if the database is used while in an erroneous state; for example, consider the following sequence of actions.

1. A **Company** has a **Product** in its **allProducts** collection but that **Product**'s **myCompany** reference is null.
2. That **Product** changes its **description** (that is, its member key), but since the **myCompany** reference is null, the member key is not updated in the **allProducts** collection.
3. The **myCompany** reference on the **Product** is set back to the **Company**. The **Company** now has both the old and the new descriptions as two keys pointing to the same object.

This could lead to a customer ordering the wrong product or other unexpected behaviors.

Fortunately, the Logical Certifier can fix this, by rebuilding the collection using the **Collection** class **rebuild** method. This removes the excess entry in the collection and ensures that every member key of the dictionary matches the property of the object referenced in the dictionary.

However, this can fail in the situation where the member key of a **Product** has changed to a key already present in the dictionary. In this case, the step of changing the member key of the dictionary to that of the referenced object's property is impossible, as it would result in a duplicate entry.

Exercise 5 – Creating an Invalid Member Key

In this exercise, generate a mismatch between the key of an entry and the object referenced by that entry, and then observe the effect on behavior.

1. Modify the **JadeScript** class **setup** method as follows, and then run it.

```
setup();

vars
    product : Product;
    company : Company;
begin
    beginTransaction;
    Company.instances.purge();
    Product.instances.purge();
    commitTransaction;

    beginTransaction;
    create company persistent;
    commitTransaction;

    beginTransaction;
    create product persistent;
    product.description := "Custom-made Widget";
    product.price := 42;
    product.myCompany := company;
    commitTransaction;

    beginTransaction;
    create product persistent;
    product.description := "Perpetual Motion Machine";
    product.price := 350;
    product.myCompany := company;
    commitTransaction;
end;
```

   This restores the test database to its original state.

2. Create a **JadeScript** class method called **breakManualSide** as follows, and then run it.

```
breakManualSide();

vars
    fixLine : String;
begin
    // A product does not know its owning company
    fixLine := "FIX1: null " &
               Product.firstInstance.getOidString &
               " myCompany";
    write fixLine;
end;
```

3. Copy and paste the output from the **breakManualSide** method to the **_logcert.fix** file in **C:\LogicalCertifierFixes**.

4. Run the **Repair** operation in the Jade Logical Certifier dialog, with **C:\LogicalCertifierFixes** specified in the **Log File Directory** text box.

5. Create a **JadeScript** class method called **changeDescription** as follows, and then run it.

```
changeDescription();

begin
    beginTransaction;
    Product.firstInstance.description := "Home-Made Widget";
    commitTransaction;
end;
```

6. Inspect **Company** in the Schema Collection Inspector (Ctrl+I).

7. Double-click **Company** to view the first (and only) **Company** in the Inspector.

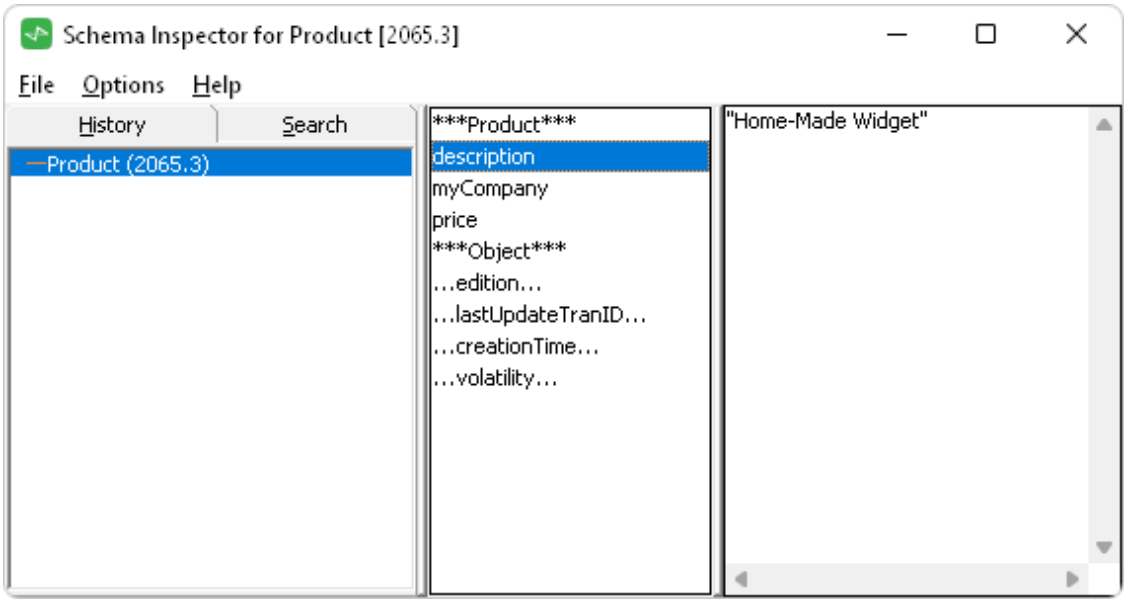
8. Double-click **allProducts** to view the **Company**'s **allProducts** collection.

9. Select the first **Product**. You should see that the description is **Home-Made Widget**.

10. Create a **JadeScript** class method called **findWidget** and code it as follows.

```
findWidget();

begin
    Company.firstInstance.allProducts["Custom-made Widget"].inspect();
end;
```

11. Run the **findWidget** method, which brings up the Schema Inspector for a product. If you look at the description, you will see that the **Custom-made Widget** entry in the **allProducts** collection refers to the widget with the description **Home-Made Widget**.

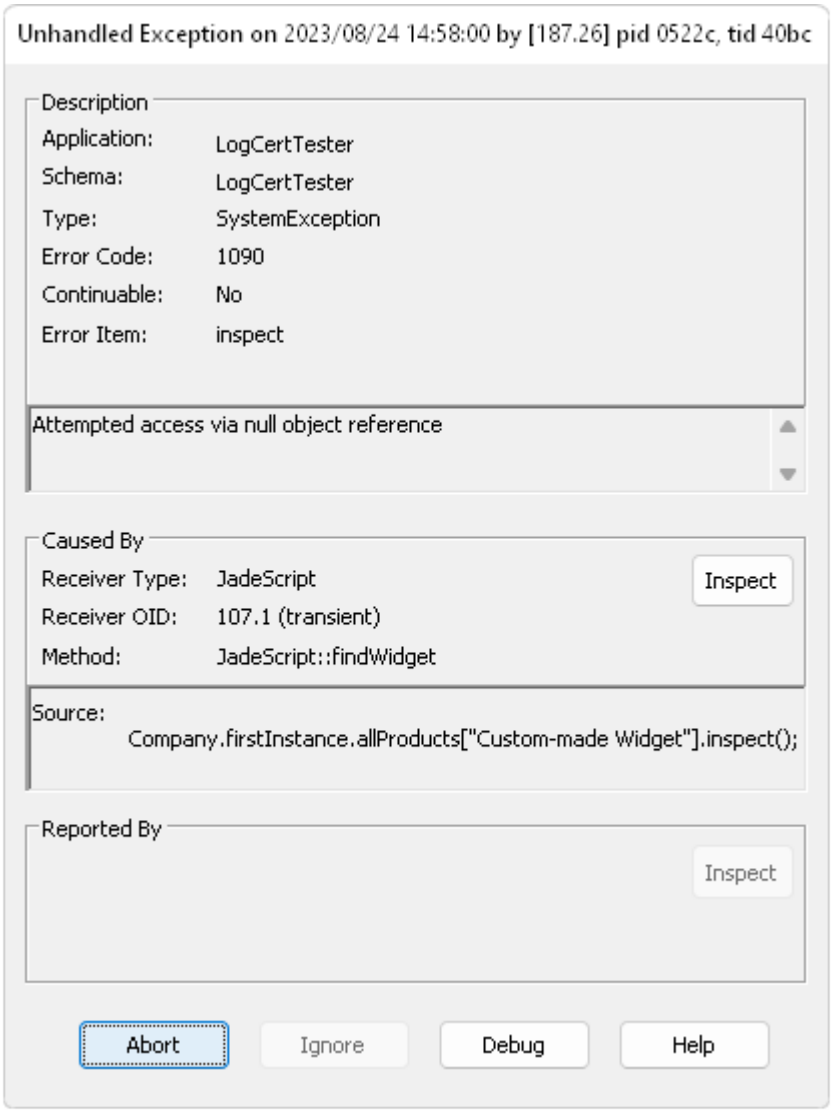


Exercise 6 – Repairing an Invalid Member Key

In this exercise, use the Logical Certifier to repair the invalid member key generated in the previous exercise.

1. Run the **Certify Selected Schemas** operation in the Jade Logical Certifier dialog to generate a fix in **C:\LogicalCertifierFixes**.
2. Run the **Repair** operation in the Jade Logical Certifier dialog, with **C:\LogicalCertifierFixes** specified in the **Log File Directory** text box.
3. Run the **JadeScript** class **findWidget** method.

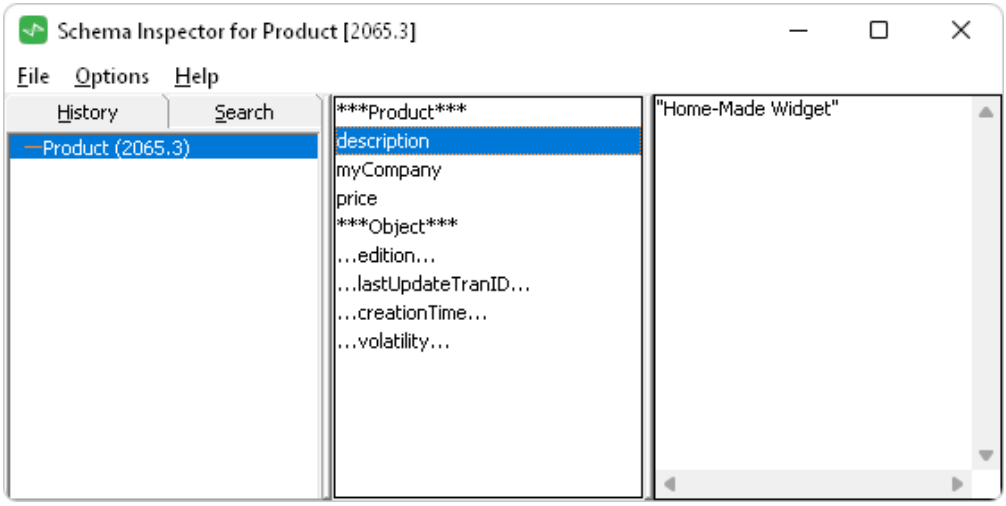
   An unhandled exception 1090 is raised, as **Custom-made Widget** no longer exists as a key in the **allProducts** collection.



4. Modify the **findWidget** method to use the new key, as follows.

```
findWidget();

begin
    Company.firstInstance.allProducts["Home-Made Widget"].inspect();
end;
```

5. Run the **findWidget** method, which should now find the **Home-Made Widget** object again.



Failing a Logical Certifier Repair

The Logical Certifier can automatically repair most referential integrity issues that may arise in a Jade database. However, it is possible to get a state where the recommended fix is not possible. An example of this is when there is an invalid member key within a collection, but setting the invalid member key to the correct key property value would result in a duplicate key exception.

The Logical Certifier avoids raising exceptions, so if it is unable to perform a fix, it logs an error in the **_repair.err** file and skips to the next fix in the **_logcert.fix** file, if there is one.

Exercise 7 – Failing a Repair

In this exercise, generate a referential integrity error that the Logical Certifier is unable to fix without help.

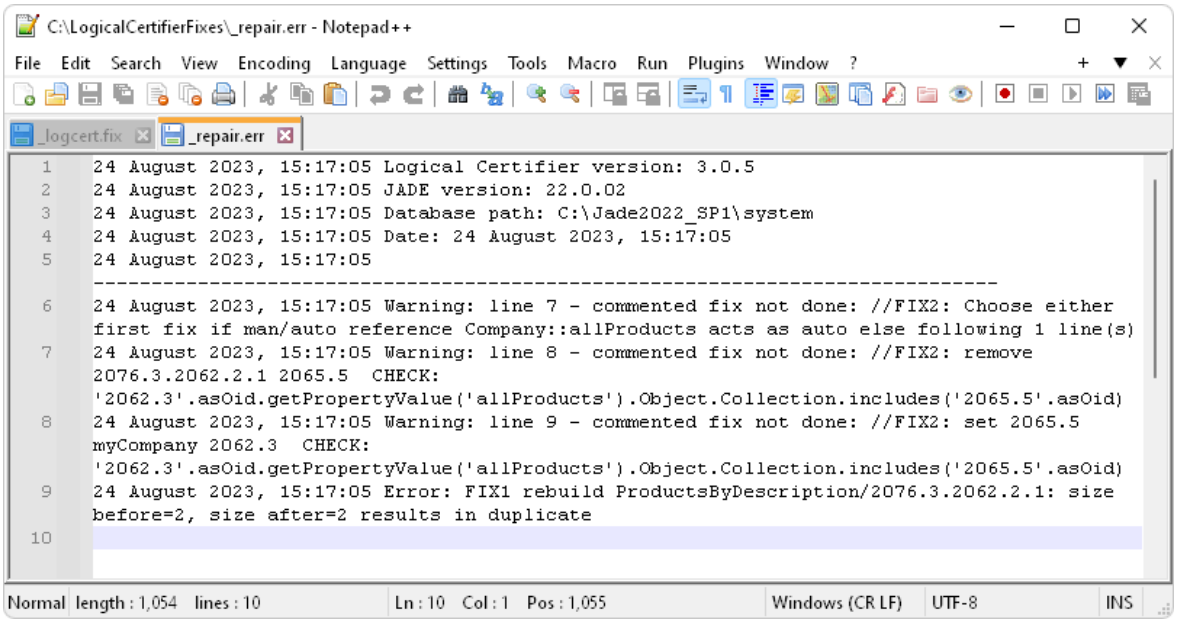
1. Run the **JadeScript** class **setup** method.
2. Run the **JadeScript** class **breakManualSide** method.
3. Copy the output from **breakManualSide** to the **_logcert.fix** file in **C:\LogicalCertifierFixes**.
4. Run the **Repair** operation in the Jade Logical Certifier dialog, with **C:\LogicalCertifierFixes** specified in the **Log File Directory** text box.

5. Modify the **JadeScript** class **changeDescription** method as follows, and then run it.

```
changeDescription();

begin
    beginTransaction;
    Product.firstInstance.description := "Perpetual Motion Machine";
    commitTransaction;
end;
```

6. Inspect **Company** in the Schema Collection Inspector (Ctrl+I).
7. Double-click **Company**, to view the first (and only) **Company** in the Inspector.
8. Double-click **allProducts**, to view the **Company**'s **allProducts** collection.
9. Select the first **Product**. You should see that the description is **Perpetual Motion Machine**.
10. Select the second **Product**. You should see that the description is also **Perpetual Motion Machine**.
11. Run the **Certify Selected Schemas** operation in the Jade Logical Certifier dialog to generate a fix in **C:\LogicalCertifierFixes**.
12. Run the **Repair** operation in the Jade Logical Certifier dialog, with **C:\LogicalCertifierFixes** specified in the **Log File Directory** text box. You should see there was one error and three warnings.

13. Open the **_repair.err** file in **C:\LogicalCertifierFixes**.



You can see that the **FIX1** failed, as rebuilding the collection would result in a duplicate. The **FIX2** was skipped, as the **allProducts** collection is set to **Man/Auto** and therefore there are two possible fixes, both of which are commented out, by default (as in Exercise 4 of this module).

Exercise 8 – Fixing a Failed Repair

In this exercise, manually change the data to allow the repair to complete.

1. Create a **JadeScript** class method called **changeSecondDescription** and code it as follows.

```
changeSecondDescription();

begin
    beginTransaction;
    Product.lastInstance.description := "Temporary Motion Machine";
    commitTransaction;
end;
```

2. Run the **changeSecondDescription** method and then inspect the **allProducts** collection of **Company**.

3. Run the **Certify Selected Schemas** operation in the Jade Logical Certifier dialog to generate a fix in **C:\LogicalCertifierFixes**.

4. Run the **Repair** operation in the Jade Logical Certifier dialog, with **C:\LogicalCertifierFixes** specified in the **Log File Directory** text box. You should see there are now zero (**0**) errors, although there are still three warnings.

5. To verify that the **allProducts** collection now has correct keys, modify the **findWidget** method as follows, and then run it.

```
findWidget();

begin
    Company.firstInstance.allProducts["Perpetual Motion Machine"].inspect();
    Company.firstInstance.allProducts["Temporary Motion Machine"].inspect();
end;
```

You should see both objects open in the Schema Inspector.

Multithreading

In Jade applications, it is sometimes useful to initiate asynchronously executing applications, threads, or processes. These tasks may need to run asynchronously for performance reasons, to separate the execution of specific functions or to allow the processes to run on different machines.

Synchronous versus Asynchronous

Typically, when you call one method from another method, the calling method waits until the called method completes before executing the next line of code, as shown in the following example.

```
method1();

begin
    write "This is the first instruction";
    self.method2();
    write "I will wait until method2 finishes before continuing";
end;
```

However, sometimes the method to be called represents a background process and there is no reason to wait for it to complete before proceeding. In this case, we would start that process in a new thread (by *multithreading*, or having multiple threads) and have it run asynchronously so that it doesn't wait for it to complete before proceeding.

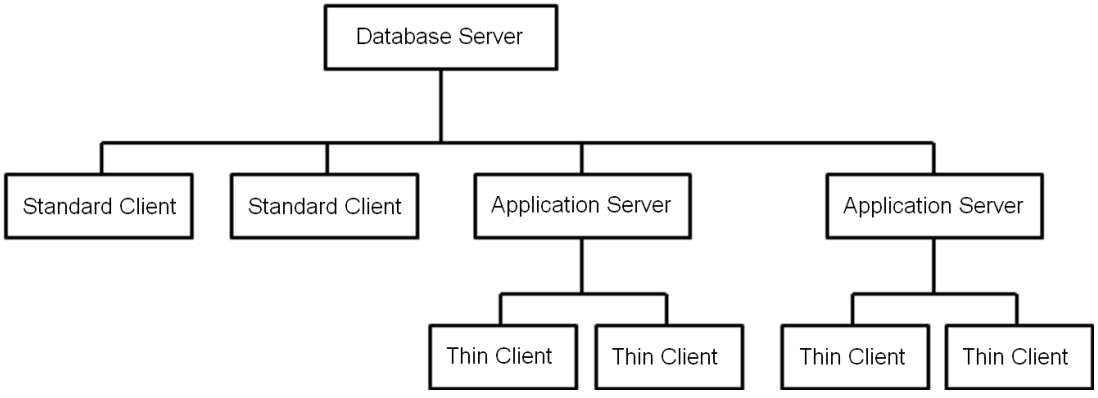
The most usual way to do this in Jade is to start a new application to run the method; for example:

```
exampleMethod();

begin
    write "This is the first instruction";
    app.startApplication("Multithreading", "ExampleApp");
    write "I will NOT wait until method2 finishes before continuing";
end;
```

In this example, **Multithreading** is the name of the schema and **ExampleApp** is an application. The **exampleMethod** method starts the **ExampleApp** application, which will call the **method2** method. However, it will not wait for the **method2** method to complete before proceeding to the next instruction.

Nodes and Processes

In the Jade Platform, a database is broken down into multiple nodes, where each connection to the database contains a database server, an application server, and clients.



The application server and client can be combined into a single standard client (often called a *fat* client).

In a single user system, the database server, application server, and clients are all combined into a single **jade.exe** program.

Note For more information, see "Module 15 – Nodes, Processes, and Caches" in the Jade Platform Developer's course.

Each of the programs that contains an application server or a database server, or both an application server and a database server, represents a Jade node. These can include any of the following executing programs.

- A **jade.exe** program for a standard (fat) client.
- A **jadapp.exe** program for an application server.
- A **jadrap.exe** program for the database server.
- An external program that uses the Jade Object Manager; for example, a .NET application that uses a Jade exposure.

Each node can have several processes running asynchronously at any time.

Initiating Asynchronous Processes

There are five **RootSchema** Jade methods that allow you to start a new application process from an existing Jade process. Any application process that is started using one of these methods runs asynchronously to the process that starts it.

- **Application**::**startApplication**
- **Application**::**startApplicationWithParameter**
- **Application**::**startApplicationWithString**
- **Application**::**startAppMethod**
- **Node**::**createExternalProcess** (which runs an external application asynchronously)

The simplest way to start a new application process is with the **Application** class **startApplication** method. The **startApplication** method takes two string parameters: the name of the schema containing the application and the name of the application.

```
exampleMethod();

begin
    write "This is the first instruction";
    app.startApplication("Multithreading", "ExampleApp");
    write "I will NOT wait until method2 finishes before continuing";
end;
```

If the application to be started requires a parameter for its **initialize** method, you can use the **Application** class **startApplicationWithParameter** method. This method takes one additional parameter, which is a shared transient object. (Although it can be persistent, it is usually a shared transient, because otherwise the other application could merely access it directly from the database.)

Note The parameter required by the **initialize** method must be an object; it cannot be a primitive type such as an **Integer**. If you need to pass through a primitive type, you can set it as a property on the object unless it is exactly a **String**, which is handled by the **startApplicationWithString** method.

```
exampleMethod();

vars
    objParam: ExampleClass;
begin
    beginTransientTransaction;
    objParam := create ExampleClass("Example String") sharedTransient;
    commitTransientTransaction;
    write "This is the first instruction";
    app.startApplicationWithParameter("Multithreading", "ExampleApp", objParam);
    write "I will NOT wait until method2 finishes before continuing";
end;
```

The passed object is then available to the **initialize** method of the application. In this case, the **ExampleParamApp** application has an **initialize** method called **paramMethod**.

Note The **initialize** method of an application is often called **initialize**, and **initialize** is the default method that is called by an application if you do not set one. However, you can call your **initialize** method whatever you like. This is especially useful if you have multiple applications with different **initialize** methods defined on the same schema.

```
paramMethod(str : ExampleClass io);

begin
    app.doWindowEvents(1000);
    write str.myString;
    beginTransientTransaction;
    delete str;
    commitTransientTransaction;
    terminate;
end;
```

Tip Make sure you delete shared transient objects and terminate applications when you have finished with them.

A common use case of passing a parameter to an asynchronous application is to pass a single string. If the parameter you want to pass is exactly one string, you can use the **startApplicationWithString** method of the **Application** class to avoid having to wrap it in an object (as would be needed to use the **startApplicationWithParameter** method).

```
exampleMethod();

begin
    write "This is the first instruction";
    app.startApplicationWithString("Multithreading", "ExampleStringApp", "A Message.");
    write "I will NOT wait until the application finishes before continuing";
end;
```

The **Application** class **startAppMethod** method allows the application to be started with an alternative **initialize** method. As this method provides the most flexibility of any methods that start an application, it also has the most parameters, which are listed in the following table.

Parameter	Type	Description
schemaName	String	Specifies the name of the schema in which the application is located.
appName	String	Specifies the name of the application to start.
methodName	String	Specifies the method that is to be invoked on the application; that is, the method to be called as the initialize method of the application.
methodParam	Object	A shared transient object to be passed to the initialize method of the application.
checkSecurity	Boolean	If set to **true**, the **getAndValidateUser** method is called to validate user codes and passwords. If **false**, the application inherits the security profile from the invoking application.

Exercise 1 – Synchronous versus Asynchronous

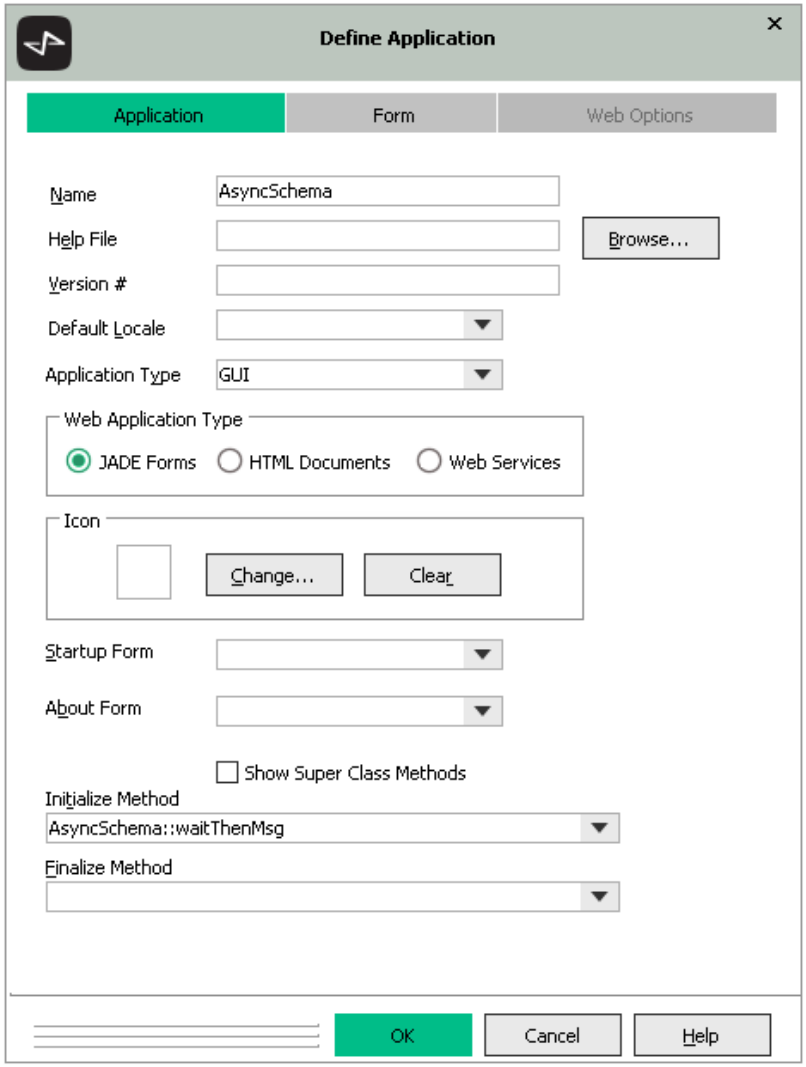
In this exercise, create a form to explore the difference between running a method synchronously versus asynchronously.

1. Create a schema called **AsyncSchema**.
2. In the Class Browser, navigate to the **AsyncSchema** subclass of the **Application** class.

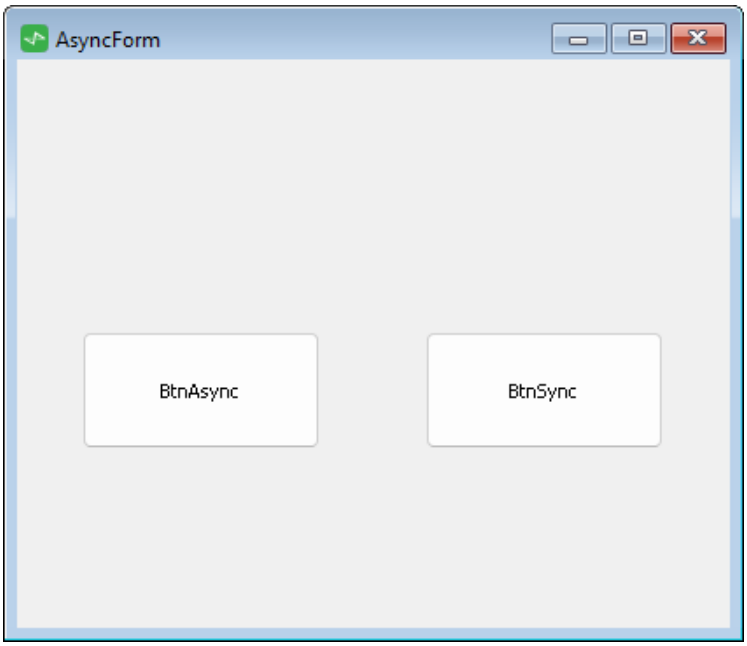
3. Add a method called **waitThenMsg** to the **AsyncSchema** application subclass, and code it as follows.

```
waitThenMsg();

vars

begin
    app.doWindowEvents(3000);
    app.msgBox("Thanks for waiting.", "MsgBox", MsgBox_OK_Only);
end;
```

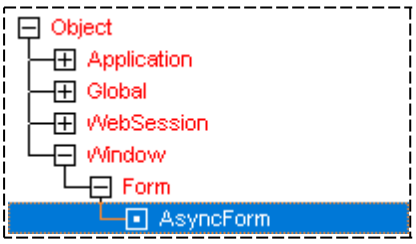
4. Open the Application Browser and create an application, as follows.



5. Open the Jade Painter and create a form called **AsyncForm** with two buttons: **btnAsync** and **btnSync**, as follows.



6. Ensuring that you have first saved **AsyncForm**, navigate to **AsyncForm** in the Class Hierarchy Browser.



7. Modify the **click** method of **btnAsync**, by coding it as follows.

```
btnAsync_click(btn: Button input) updating;

begin
    btnAsync.enabled := false;
    btnSync.enabled := false;

    app.startApplication("AsyncSchema", "AsyncApp");

    btnAsync.enabled := true;
    btnSync.enabled := true;
end;
```

8. Modify the **click** method of **btnSync**, by coding it as follows.

```
btnSync_click(btn: Button input) updating;

begin
    btnAsync.enabled := false;
    btnSync.enabled := false;

    app.waitThenMsg();

    btnAsync.enabled := true;
    btnSync.enabled := true;
end;
```

9. Create a **JadeScript** class method called **createAsyncForm**, coding it as follows.

```
createAsyncForm();

vars
    form : AsyncForm;
begin
    create form;
    form.showModal();
epilog
    delete form;
end;
```

10. Run the **createAsyncForm** method and click on the two buttons.

    How do they behave differently?

Shared Transient Objects

If you have completed the Jade Platform Developer's course, you are likely familiar with the difference between transient and persistent objects. If not, see the "Module 15 – Nodes, Processes, and Caches" and "Module 16 – Transactions and Locking" legacy modules of this course.

Shared transient objects are halfway between transient and persistent objects.

- Persistent objects are stored in the database and are therefore accessible to *all* processes on *every* node of the database.
- Transient objects are stored in the transient cache on a node and are accessible to *one* process on *one* node.
- Shared transient objects are also stored in the transient cache on a node, but they are accessible to *all* processes on that *one* node.

As multiple processes can access the same shared transient object at the same time, changes to shared transient objects must be performed inside transactions, which is like persistent objects, but using the **beginTransientTransaction** and **commitTransientTransaction** instructions.

```
exampleTransaction();

vars
    sharedTransientObj : SharedTransientClass;
begin
    beginTransientTransaction;
    create sharedTransientObj sharedTransient;
    commitTransientTransaction;
end;
```

Shared transient objects are an integral part of multithreading, as they allow for communication between applications within a node.

Notifications and Callbacks

When running multiple threads asynchronously, it is often a requirement to communicate progress from the asynchronous operation back to the caller, usually called a *callback*.

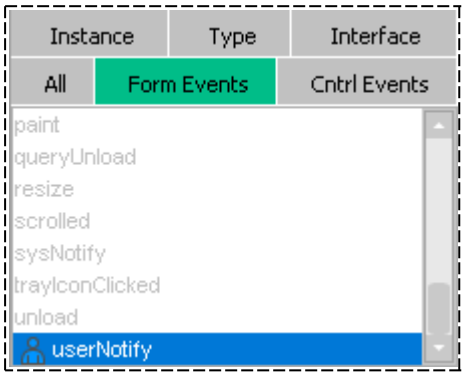
The callback strategy commonly used in Jade systems is to subscribe to user events on a shared transient object, pass that shared transient to the asynchronous process, and then cause events from the asynchronous process.

The shared transient object that is used does not need to be at all complex; in fact, no methods or properties are required on it for it to perform its role in the callback process. Once created, the initiating process should subscribe to notifications from it by using the **beginNotification** method of the **Object** class.

```
btnAsync_click(btn: Button input) updating;

vars
    handler : CallbackHandler;
begin
    beginTransientTransaction;
    create handler sharedTransient;
    commitTransientTransaction;

    beginNotification(handler, Example_Event, Response_Continuous, 0);
end;
```

In this example, **CallbackHandler** is an empty class with no properties or methods, and **Example_Event** is a global constant of type **Integer**, with a value of **1234**.

Once the caller is subscribed to notifications by calling the **beginNotification** method, it still needs to define a behavior to perform when it is notified of that event by that object. For a form, the easiest way to do this is by adding behavior to the **userNotify** method on the **Form Events** folder of the Class Browser.



The following is an example of the code required for the **userNotify** method.

```
userNotify(eventType: Integer; theObject: Object; eventTag: Integer; userInfo: Any) updating;

vars

begin
   /*
    * The behavior to be performed, this can be
    * modifying an element on the form or bringing
    * up a message box etc.
    */
    write eventType.String;
end;
```

To generate the event from the asynchronous process, first ensure that the process has access to the shared transient object, which typically involves passing it in by using the **startApplicationWithParameter** method when that process is first created.

```
btnAsync_click(btn: Button input) updating;

vars
    handler : CallbackHandler;
begin
    beginTransientTransaction;
    create handler sharedTransient;
    commitTransientTransaction;

    beginNotification(handler, Example_Event, Response_Continuous, 0);

    app.startApplicationWithParameter(currentSchema.name, "CallBackApp", handler);
end;
```

In this example, the **CallbackApp** application has an initialize method called **createEvent**. To generate an event on the shared transient object, the **causeEvent** method of the **Object** class is used.

```
createEvent(callback : CallbackHandler);

begin
    app.doWindowEvents(1000);
    callback.causeEvent(Example_Event, true, null);
end;
```

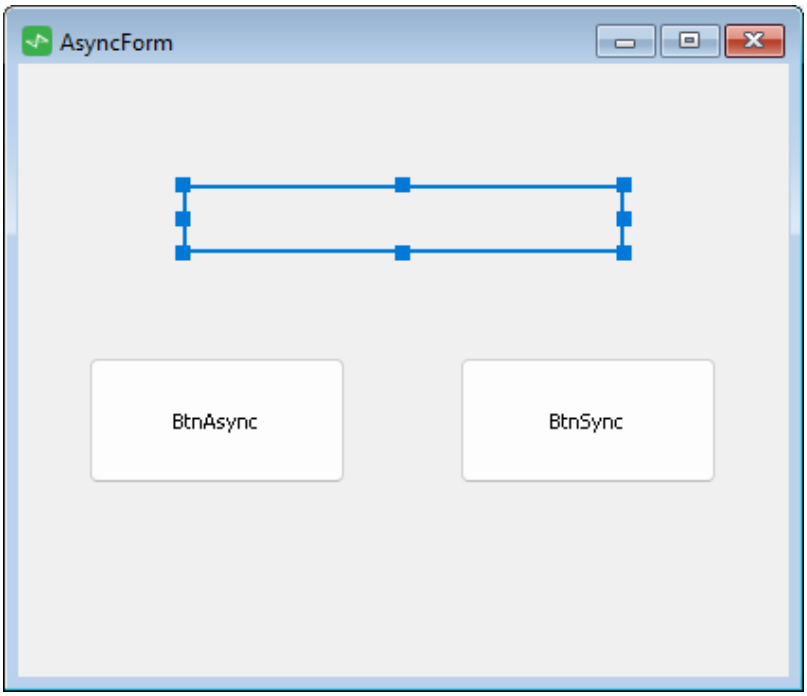
When this event is caused by the asynchronous process on the shared transient **CallbackHandler** object, the initiating process is notified of the event and the **userNotify** method is invoked.

Exercise 2 – Generating a Callback

In this exercise, use a shared transient object to report back as an asynchronous operation begins and finishes.

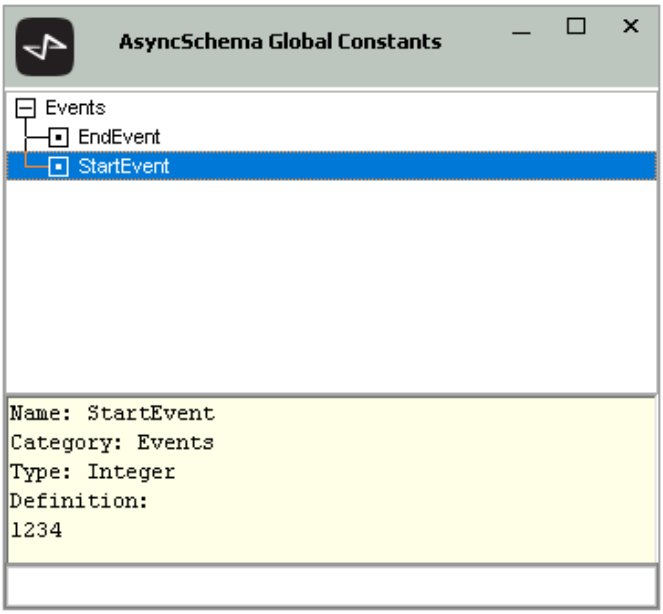
As the **BtnAsync** method creates asynchronous applications, a counter will display how many operations are pending, and that counter will decrement as they complete.

1. Modify the **AsyncForm** in the Jade Painter to add a label called **lblCallback** with an empty caption.



2. Create a class called **CallbackHandler**. You do not yet need to add any methods or properties to this class.

3. Add two global constants (Ctrl+G): **StartEvent** as an **Integer** with a value **1234** and **EndEvent** as an **Integer** with a value **1235**.



4. Create a method called **getHandler** in the **AsyncForm** class, and code it as follows.

```
getHandler() : CallbackHandler;

vars
    handler : CallbackHandler;

begin
    beginTransientTransaction;
    create handler sharedTransient;
    commitTransientTransaction;

    beginNotification(handler, StartEvent, Response_Cancel, 0);
    beginNotification(handler, EndEvent, Response_Cancel, 0);

    return handler;
end;
```

5. Modify the **click** method of the **btnAsync** button control as follows.

```
btnAsync_click(btn: Button input) updating;

begin
    btnAsync.enabled := false;
    btnSync.enabled := false;

    app.startApplicationWithParameter("AsyncSchema", "AsyncApp", self.getHandler());

    btnAsync.enabled := true;
    btnSync.enabled := true;
end;
```

6. Modify the **click** method of the **btnSync** button control as follows.

```
btnSync_click(btn: Button input) updating;

begin
    btnAsync.enabled := false;
    btnSync.enabled := false;

    app.waitThenMsg(self.getHandler());

    btnAsync.enabled := true;
    btnSync.enabled := true;
end;
```

7. Add an **Integer** attribute property called **pendingOperations** to **AsyncForm**.

8. Modify the **userNotify** method in the **Form Events** as follows.

```
userNotify(eventType: Integer; theObject: Object; eventTag: Integer; userInfo: Any) updating;

vars

begin
    if eventType = StartEvent then
        self.pendingOperations := self.pendingOperations + 1;
    elseif eventType = EndEvent then
        self.pendingOperations := self.pendingOperations - 1;
    endif;
    self.lblCallback.caption := "There are "
                              &  self.pendingOperations.String
                              &  " pending operations remaining.";
end;
```

Caution The various elements of the **AsyncForm** such as **btnAsync** and **lblCallback** also have **userNotify** methods, but it is the **userNotify** method of the form itself that we want; that is, the one found in the **Form Events** folder.

9. Modify the **waitThenMsg** method in the **AsyncSchema** subclass of the **Application** class as follows.

```
waitThenMsg(callback : CallbackHandler);

begin
    callback.causeEvent(StartEvent, true, null);
    app.doWindowEvents(3000);
    callback.causeEvent(EndEvent, true, null);
    app.msgBox("Thanks for waiting.", "MsgBox", MsgBox_OK_Only);
end;
```

10. Run the **JadeScript** class **createAsyncForm** method.

    As you click **BtnAsync**, the number of pending operations increases, and as the message boxes appear, the number falls again.

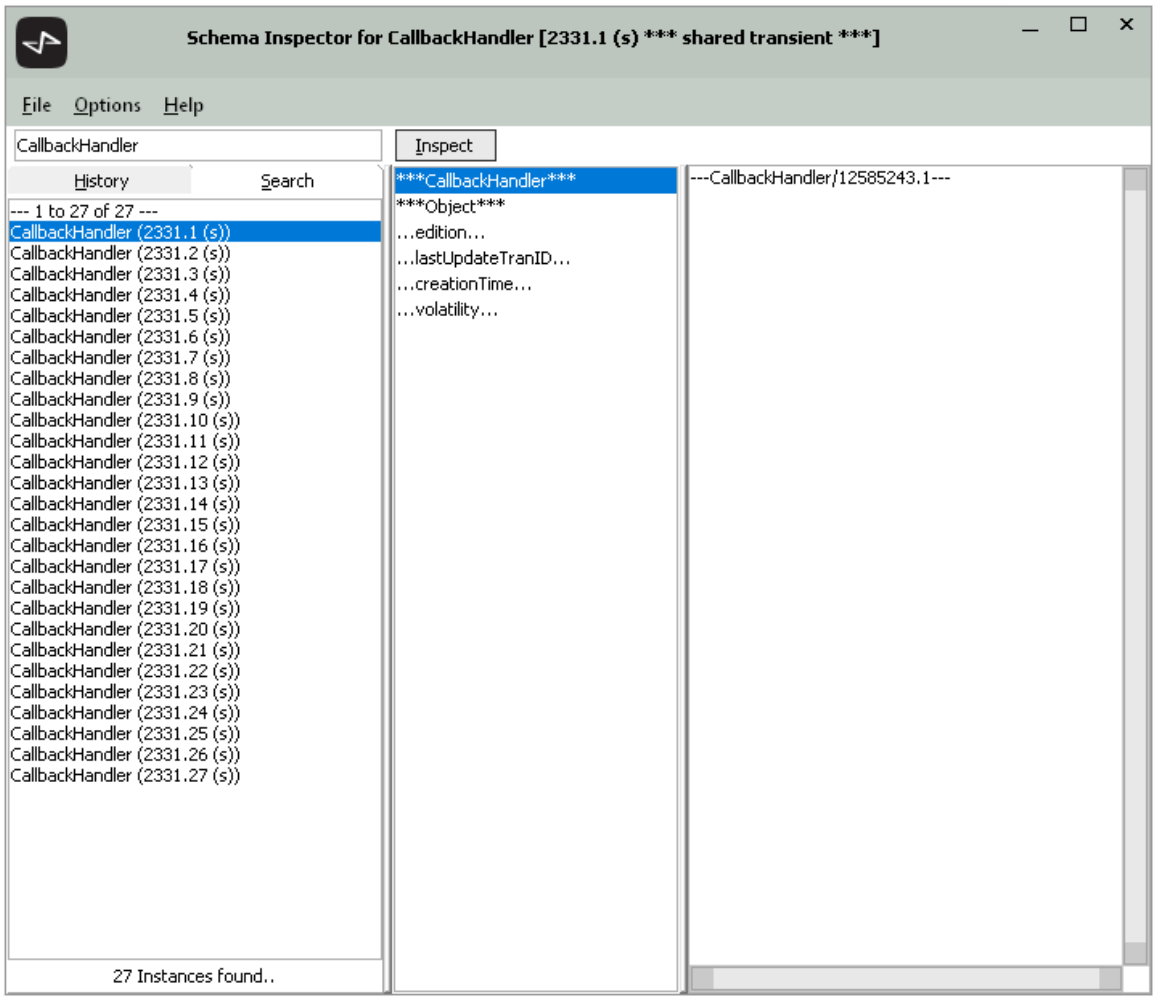
Exercise 3 – Dealing with Shared Transient Leaks

Unlike standard transient objects. which are deleted at the termination of their application, shared transient objects have a lifetime equal to that of their node so they can therefore build up if they are not deleted after use.

In this exercise, identify, fix, and close a shared transient leak.

1. Select **CallbackHandler** in the Class Browser and press the Ctrl+J shortcut keys to invoke the Schema Collection Inspector form.

2. You should see many shared transient instances of **CallbackHandler**, depending on how many times you have created shared transient **CallbackHandler** instances since you last restarted your application server.



3. To remove this build-up of instances of **CallbackHandler**, which is likely filling your transient cache, we could restart the application server (whether a single user Jade system, a standard client, or an application server plus presentation client). However, it is easier to create a **JadeScript** class method to simply remove the excess transient instances manually.

Create a **JadeScript** class method called **deleteTransients** and code it as follows.

```
deleteTransients();

vars
    transients : ObjectArray;
begin
    create transients transient;
    CallbackHandler.allSharedTransientInstances(transients, 0, false);

    beginTransientTransaction;
    transients.purge();
    commitTransientTransaction;
end;
```

4. To prevent the shared transient instances from leaking in future, we should delete them when we are finished with them. The last time we used the callback handler is in the handling of the **EndEvent**. Modify the **userNotify** method as follows.

```
userNotify(eventType: Integer; theObject: Object; eventTag: Integer; userInfo: Any) updating;

vars
    handler : Object;

begin
    if eventType = StartEvent then
        self.pendingOperations := self.pendingOperations + 1;
    elseif eventType = EndEvent then
        self.pendingOperations := self.pendingOperations - 1;
        handler := theObject;
        beginTransientTransaction;
        delete handler;
        commitTransientTransaction;
    endif;

    self.lblCallback.caption := "There are "
                              & self.pendingOperations.String
                              & " pending operations remaining.";
end;
```

Note In the **userNotify** method, **theObject** parameter represents the object that notified the method of the event. We cannot directly delete **theObject**, as it is a constant parameter; however, we can assign it to a new variable and delete that.

5. Run the application again (using the **JadeScript** class **createAsyncForm** method).

   The **CallbackHandler** shared transient instances are now deleted automatically after use.

Asynchronous Method Calls

The strategy with creating a new process for each asynchronous method and using a callback to report progress, described so far in this module, is only one of two ways to implement asynchronicity in the Jade Platform. The other strategy is the use of a dedicated asynchronous worker process and using the **invoke** and **waitForMethods** methods of the **JadeMethodContext** class.

To use the **JadeMethodContext** class to call methods asynchronously, you will need to prepare the following.

1. A worker process in which **app.asyncInitialize** has been called (usually in the **initialize** method of the application; that is, the method that fulfils the **initialize** event, which is often, but not always, called **initialize**).
2. The method to be called, which does not need anything special, as in Jade, the method does not have to be marked as asynchronous; only the process that runs it. However, as it is going to be called from another process, it must not be a method of a transient object.
3. A transient instance of the **JadeMethodContext** class for the asynchronous task.
4. A transient instance of the **JadeMethodContext** class to receive the results of the asynchronous tasks.

When the **JadeMethodContext** instances have been instantiated and the worker process application has been started, the **workerAppName** property of the contexts must be set to the name of the worker process application. This allows the contexts to invoke methods on the worker processes asynchronously, using the **invoke** method. As soon as the **invoke** method has been called, the methods will run on the targeted process.

```
asynchMethodCall();

vars
    context1, context2  : JadeMethodContext;
    awaiter             : JadeMethodContext;
    worker              : Process;
    targetClass         : ExampleClass;
begin
    create context1 transient;
    create context2 transient;

    targetClass := ExampleClass.firstInstance();

    worker := app.startApplication(currentSchema.name, "WorkerApplication");

    context1.workerAppName := "WorkerApplication";
    context2.workerAppName := "WorkerApplication";

    context1.invoke(targetClass, longMethod);
    context2.invoke(targetClass, longMethod);

epilog
    delete context1;
    delete context2;
end;
```

Note The **invoke** method takes an object, a method of that object, and optionally any parameters that the method might have.

If you require a return value from these methods, you can wait for them to finish on the caller by using **process.waitForMethods**. This method takes any number of **JadeMethodContext** instances as parameters, which can be individual object references or collections of **JadeMethodContext**. It returns the **JadeMethodContext** of the first context to finish its method, and you can then use the **getReturnValue** method of that context to obtain the return value of the called method.

```
asynchMethodCall();

vars
    context1, context2  : JadeMethodContext;
    awaiter             : JadeMethodContext;
    worker              : Process;
    targetClass         : ExampleClass;
begin
    create context1 transient;
    create context2 transient;

    targetClass := ExampleClass.firstInstance();

    worker := app.startApplication(currentSchema.name, "WorkerApplication");

    context1.workerAppName := "WorkerApplication";
    context2.workerAppName := "WorkerApplication";

    context1.invoke(targetClass, longMethod);
    context2.invoke(targetClass, longMethod);

    while true do
        awaiter := process.waitForMethods(context1, context2);
        if awaiter = null then
            break;
        else
            write awaiter.getReturnValue();
        endif;
    endwhile;

epilog
    delete context1;
    delete context2;
end;
```

Note If there are no more methods executing, the **waitForMethods** method returns null.

Exercise 4 – Invoking a Method using JadeMethodContext

In this exercise, use the **invoke** method of **JadeMethodContext** class to call a method asynchronously. You will perform the following actions to set up and use the **invoke** method.

1. Create a new class with a method that takes some time to execute.
2. Create a new worker application that can perform asynchronous tasks.
3. Modify the **AsyncForm** to start this worker on loading and terminate it on closing.

4. Modify the **AsyncForm** to allow for the creation of instances of the new class and the asynchronous execution of its method.

To invoke a method using **JadeMethodContext**, perform the following steps.

1. In the **Object** class, add a subclass called **Turtle** with a method called **moveSlowly**, coded as follows.

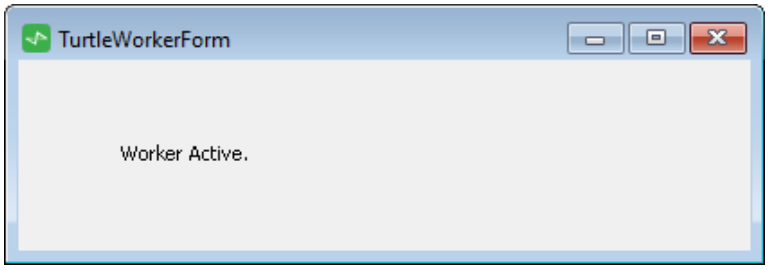
```
moveSlowly() : Turtle;

begin
    app.doWindowEvents(5000);
    app.msgBox("I have finally arrived...", "A Turtle's Journey", MsgBox_OK_Only);
    return self;
end;
```

2. In the **AsyncSchema** subclass of the **Application** class, add the following methods.

```
turtleWorkerInit();

begin
    app.asyncInitialize();
end;
```

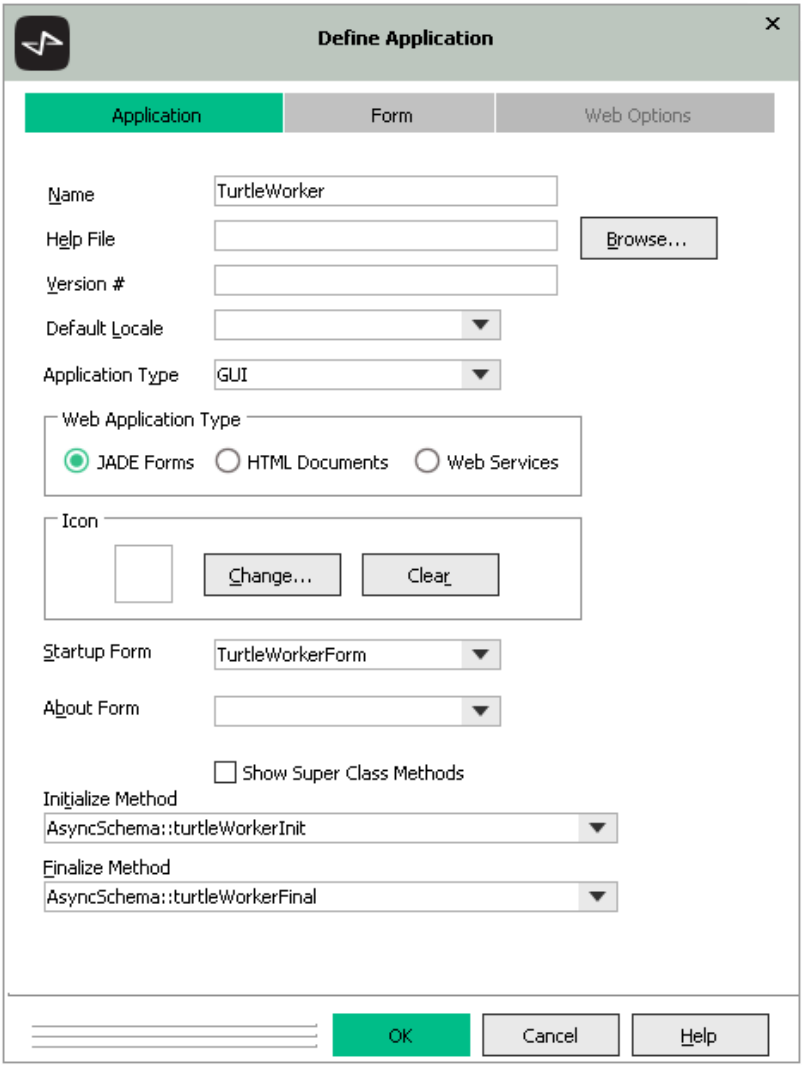
```
turtleWorkerFinal();

begin
    app.asyncFinalize;
end;
```

3. Open the Jade Painter and create a new form called **TurtleWorkerForm**, as follows.

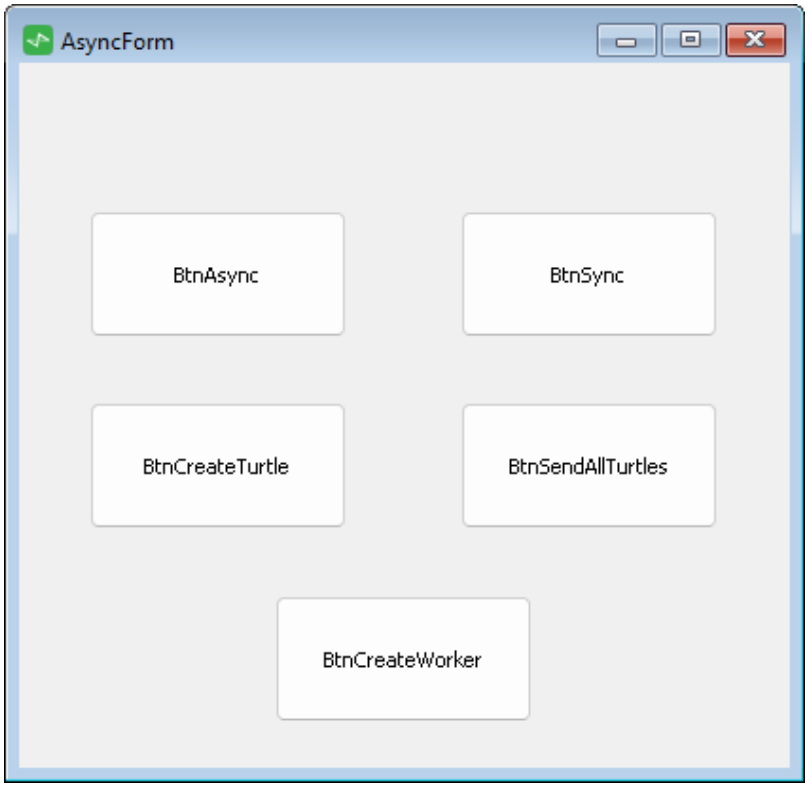


Note This form does not have any usable controls, as it is used only to give a visual indicator of active workers and to provide a simple mechanism to terminate the worker (closing the window). Normally, workers would always be non-GUI applications.

4. Open the Application Browser and create an application called **TurtleWorker**, as follows.



5. In the Jade Painter, modify the **AsyncForm** to add three new buttons, as follows.



6. Modify the **click** method of the **btnCreateTurtle** button control, as follows.

```
btnCreateTurtle_click(btn: Button input) updating;

vars
    babyTurtle : Turtle;
begin
    beginTransaction;
    create babyTurtle persistent;
    commitTransaction;
end;
```

7. Modify the **click** method of the **btnCreateWorker** button control, as follows.

```
btnCreateWorker_click(btn: Button input) updating;

vars

begin
    app.startApplication(currentSchema.name, "TurtleWorker");
end;
```

8. Create a method called **sendTurtle** on the **AsyncForm** class, coded as follows.

```
sendTurtle(turtle : Turtle input);

vars
    context : JadeMethodContext;
begin
    create context transient;
    context.workerAppName := "TurtleWorker";
    context.invoke(turtle, moveSlowly);
epilog
    delete context;
end;
```

9. Modify the **click** method of the **btnSendAllTurtles** button control, as follows.

```
btnSendAllTurtles_click(btn: Button input) updating;

vars
    turtle : Turtle;
begin
    foreach turtle in Turtle.instances do
        self.sendTurtle(turtle);
    endforeach;
end;
```

10. Run the form using the **JadeScript** class **createAsyncForm** method. Try creating various numbers of turtles and workers. How long does it take for the message boxes to be displayed?

Turtles	Workers	Time Taken
1	1	
2	1	
5	1	
5	2	
10	2	
10	5	

Exercise 5 – Waiting for Asynchronous Operations

Currently, whenever the **btnSendAllTurtles** button is clicked, the **Turtle** class will pop up a message box but remain in the database. By using the **waitForMethods** method, you can have the **click** method of the **btnSendAllTurtles** button control delete the **Turtle** class from the database when it has finished its **moveSlowly** method.

In this exercise, use the **Process** class **waitForMethods** method to wait for the completion of the asynchronous methods.

1. Modify the **moveSlowly** method of the **Turtle** class, as follows.

```
moveSlowly() : Turtle;

begin
    app.doWindowEvents(5000);
    app.msgBox("I have finally arrived...", "A Turtle's Journey", MsgBox_OK_Only);
    return self;
end;
```

2. Modify the **sendTurtle** method on the **AsyncForm** class, as follows.

```
sendTurtle(turtle : Turtle input) : JadeMethodContext;

vars
    context : JadeMethodContext;
begin
    create context transient;
    context.workerAppName := "TurtleWorker";
    context.invoke(turtle, moveSlowly);
    return context;
end;
```

3. Modify the **click** method of the **BtnSendAllTurtles** button control, as follows.

```
btnSendAllTurtles_click(btn: Button input) updating;

vars
    turtle      : Turtle;
    contexts    : ObjectArray;
    context     : JadeMethodContext;
begin
    create contexts transient;
    foreach turtle in Turtle.instances do
        contexts.add(self.sendTurtle(turtle));
    endforeach;

    while true do
        context := process.waitForMethods(contexts);
        if context = null then
            break;
        else
            beginTransaction;
            delete context.getReturnValue().Turtle;
            commitTransaction;
        endif;
    endwhile;
epilog
        delete contexts;
end;
```

4. Run the form using the **JadeScript** class **createAsyncForm** method.

   Try creating a few turtles by clicking **BtnCreateTurtle**, using **BtnSendAllTurtles**, and then inspecting the instances of **Turtle** in the Schema Collection Inspector.

   You should see that the turtle objects are deleted only after you close the message boxes.

Report Writer

The Jade Platform Report Writer is a tool that allows end-users and application developers to develop simple reports of a Jade database from a graphical, drag-and-drop interface.

The report writer contains the following applications.

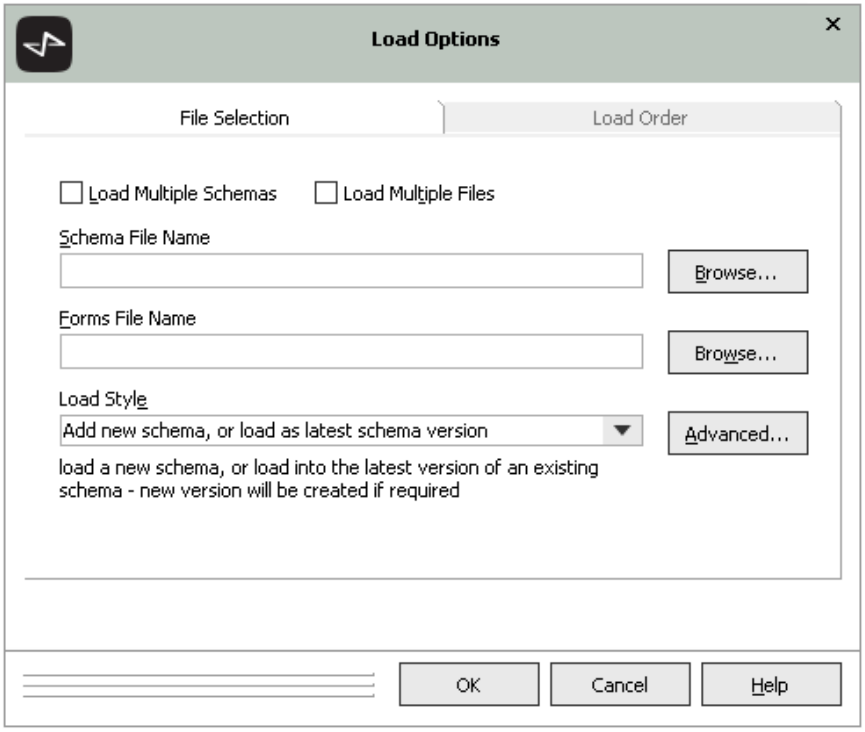
- A configuration application for creating views, which specify what is visible to the report designers
- A designer application for building reports based on the created views

Importing the Report Writer

The Report Writer files are automatically included with the Jade Platform itself, so you do not need to download any additional files to use the Report Writer. To use the Report Writer, you need only to load the **JadeReportWriterSchema** as a subschema of the Jade **RootSchema**.

The **JadeReportWriterSchema** is found in the **reportwriter** folder in the directory into which you installed the Jade Platform (for example, **C:\JadeCourse\reportwriter**).

To load the **JadeReportWriterSchema**, select **RootSchema** in the Schema Browser and then select the **Load** command from the Schema menu.



Click **Browse** at the right of the **Schema File Name** text box and then select **JadeReportWriterSchema.scm** from the **reportwriter** folder. Click **OK**, to load the Report Writer as a subschema of **RootSchema**.

Creating a Reporting View

Reports designed in the Report Writer Designer application are based on a view of the database.

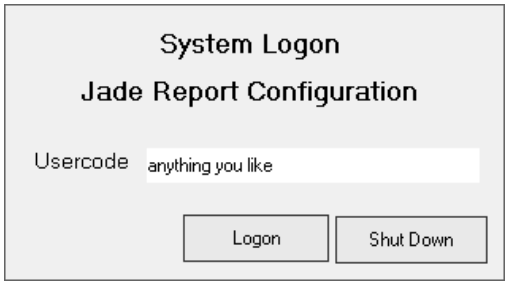
These reporting views specify which classes, and which features of those classes, are visible to the reports.

Note A feature can be a property or a method. However, if it is a method, the return type must be a primitive type (for example, **Integer** or **String**) rather than an object.

To create a reporting view, use the **ReportWriterConfiguration** application in the **JadeReportWriterSchema**. You can open this using either of the following actions.

- Select the **Run Application** toolbar button in the Jade Platform development environment, select **ReportWriterConfiguration**, and then click **OK**.

  The following dialog is displayed. You can enter any anything you like as the user code.

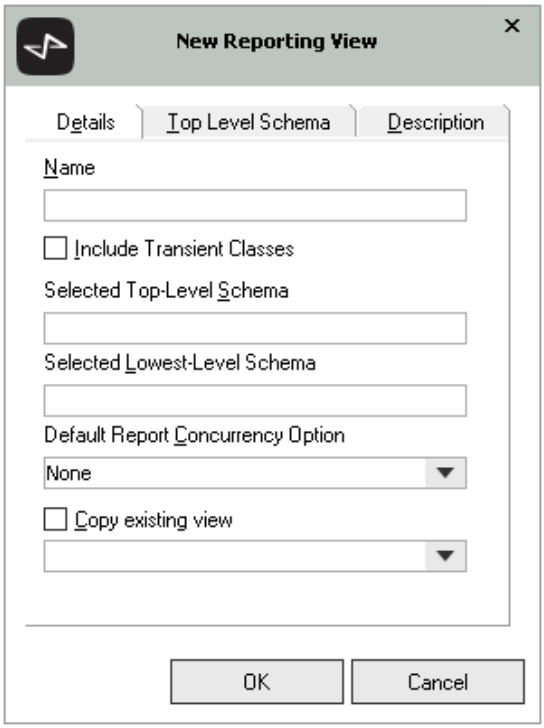


- Write a **JadeScript** class method in the schema that is to be reported on, as follows.

```
startConfiguration();

vars
    rwManager : JadeReportWriterManager;
begin
    create rwManager transient;
    rwManager.startReportWriterConfiguration("a user name", null);
epilog
    delete rwManager;
end;
```

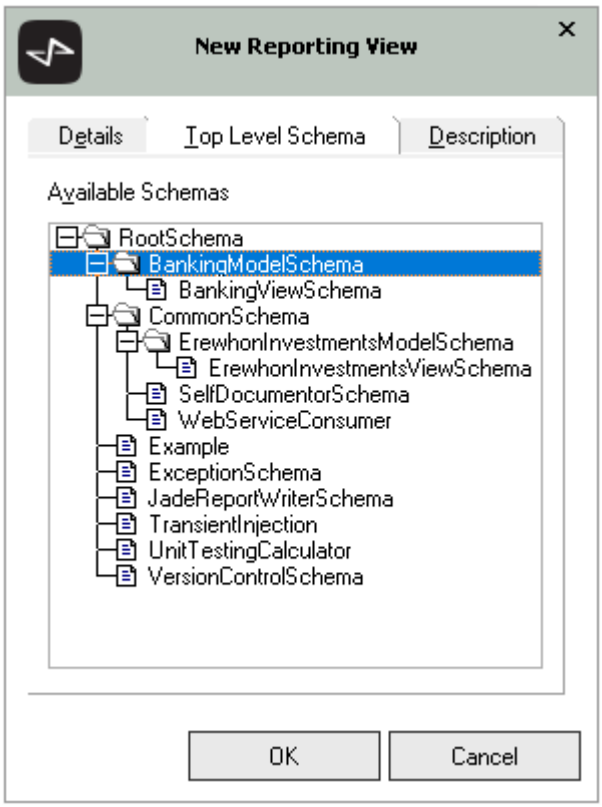
Note Later in this module, we will set up different levels of security based on the entered user name, but for now any user name is accepted.

To create a new reporting view, select the **New** command from the View menu. The New Reporting View dialog is then displayed in the Report Configuration window.



You can specify a name for the view, the top-level schema, the lowest-level schema, and the default concurrency option. Alternatively, you can copy settings from an existing view, if you have one.

When you click on the **Selected Top-Level Schema** text box, a hierarchy of available schemas is displayed on the **Top Level Schema** sheet. Select the schema that you want to be the highest-level schema to include in the view, and then return to the **Details** sheet to finish filling out the form.



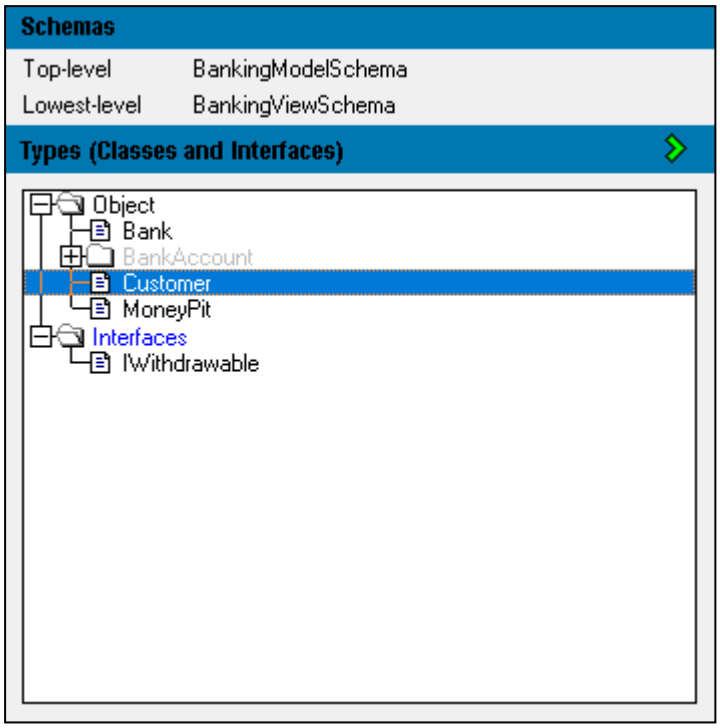
When you have selected the schema, click **OK** to create the view. The **Types & Features** sheet of the Report Configuration window is then displayed.

Selecting Types and Features

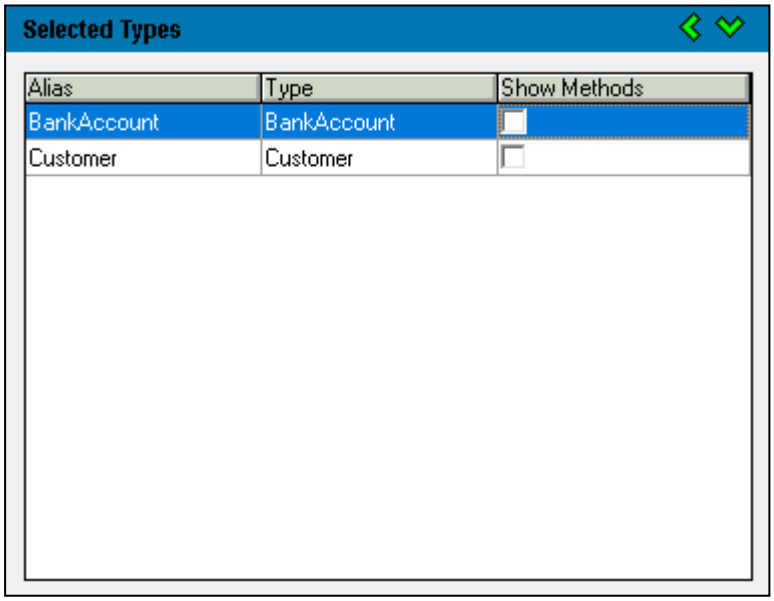
Use the **Types & Features** sheet of the Report Configuration window to select which types (classes and interfaces) and features (methods and properties) will be included in your reporting view.

This sheet contains four main panes, which are manipulated to describe the scope of the reporting view, which are:

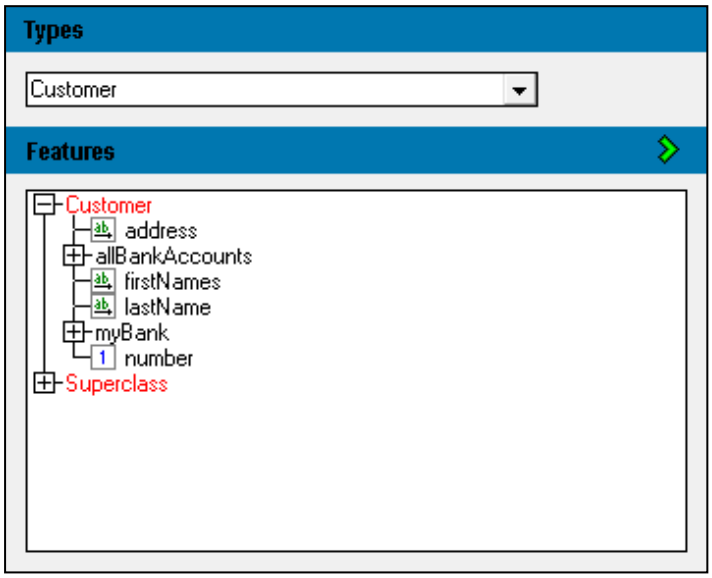
- **Types (Classes and Interfaces)**, displaying all classes contained within all schemas between the top-level schema and the lowest-level schema, inclusive.



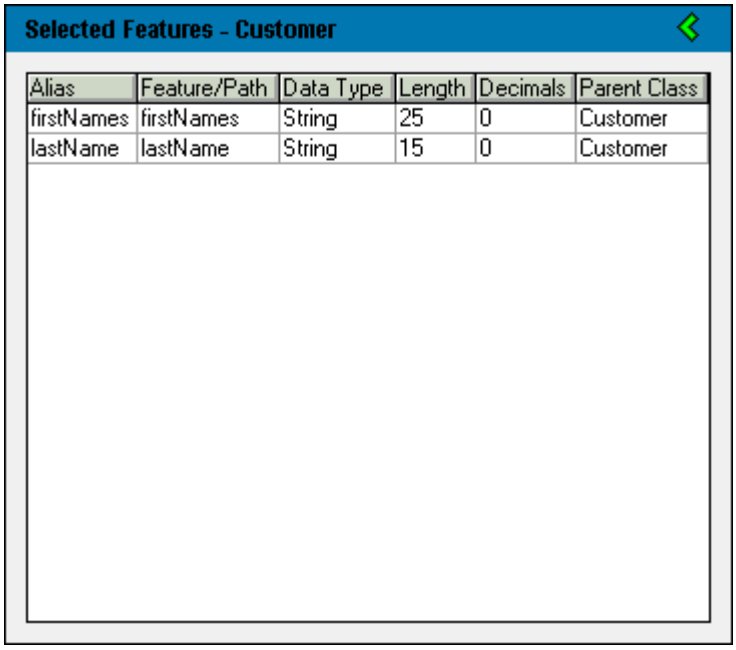
- **Selected Types**, displaying a list of all classes that have been selected to be part of the reporting view.



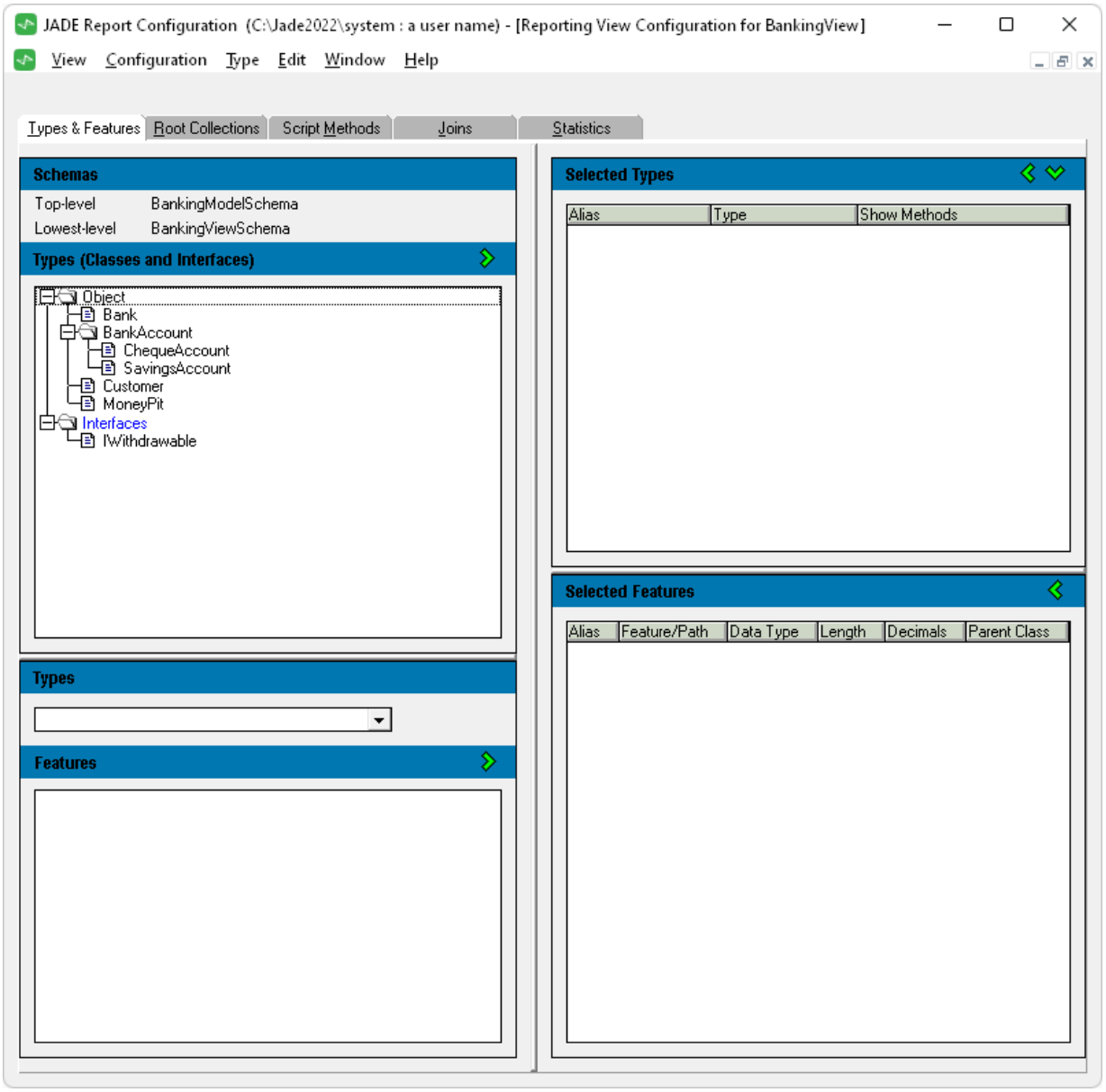
- **Features**, displaying a list of all available features of the current type. To change the current type, select it in the **Selected Types** pane or select a type from the **Types** combo box above the **Features** pane.



- **Selected Features**, displaying a list of features that have been selected for the current type.



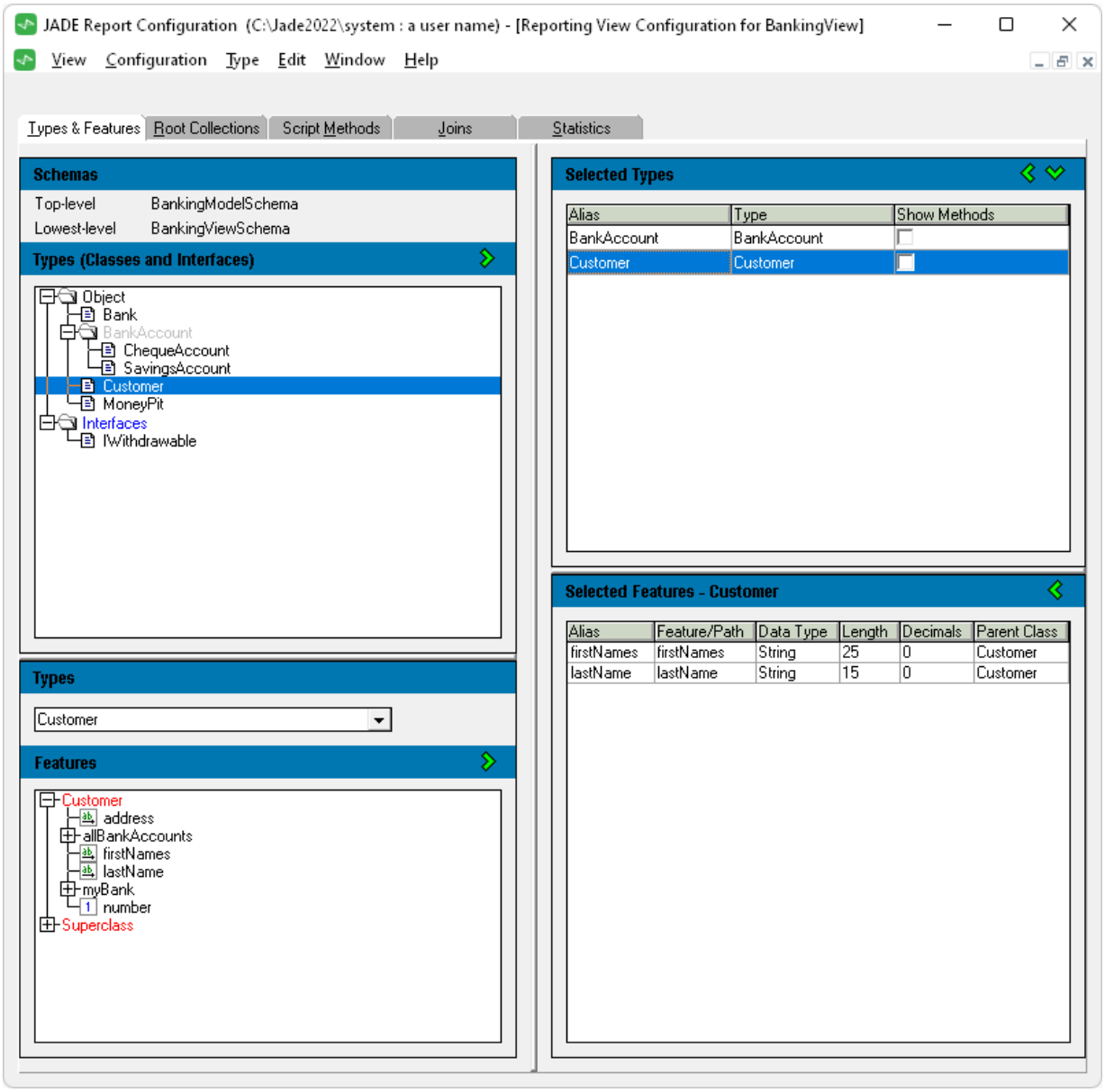
The following is an example of all four panes on the Report Configuration window.



Each of the following actions adds a class to the selected types.

- Select the class and then click the green arrow in the **Types (Classes and Interfaces)** pane header.
- Click and drag the class from the **Types (Classes and Interfaces)** pane to the **Selected Types** pane.
- Double-click the class in the **Types (Classes and Interfaces)** pane.

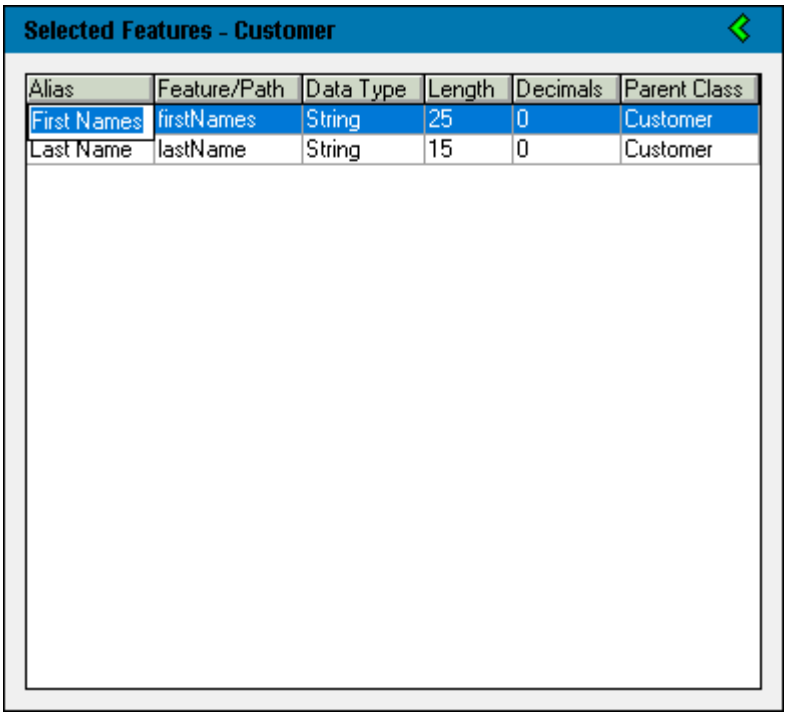
The following is an example of classes added to selected types.



Each of the following actions adds a feature to the selected features.

- Select the feature and then click the green arrow in the **Features** pane header.
- Click and drag the feature from the **Features** pane to the **Selected Features** pane.
- Double-click the feature in the **Features** pane.

To set an alias for a feature, double-click the feature name in **Alias** column of the table in the **Selected Features** pane and then specify the required name.



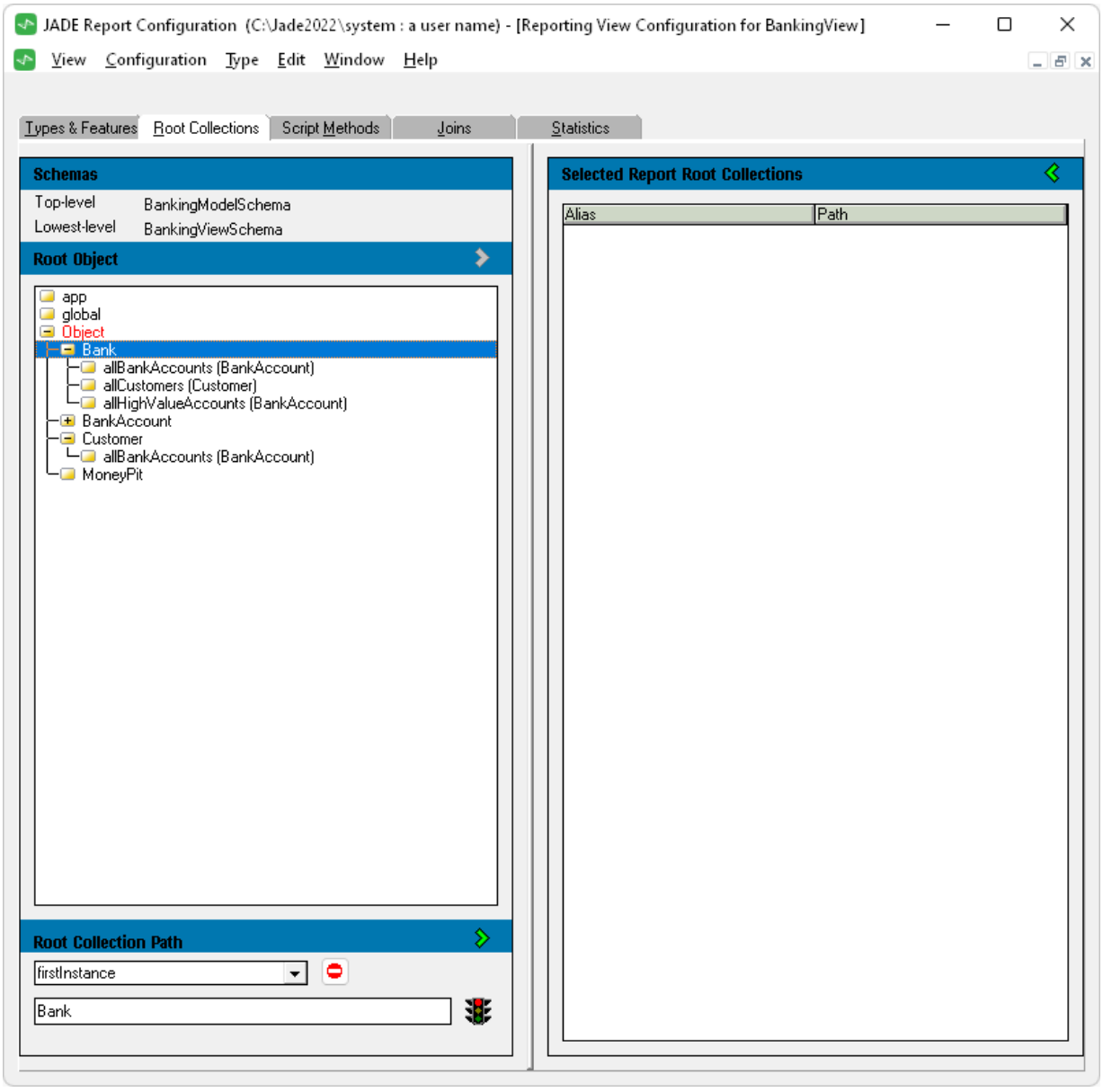
Note There is no need to save or confirm your actions. The reporting view is automatically updated as you make changes.

Root Collections

Reports are based on root collections, which provide the primary source of data for your reports. These root collections must be existing collections containing objects that have been selected in the **Types & Features** pane of the Report Configuration window.

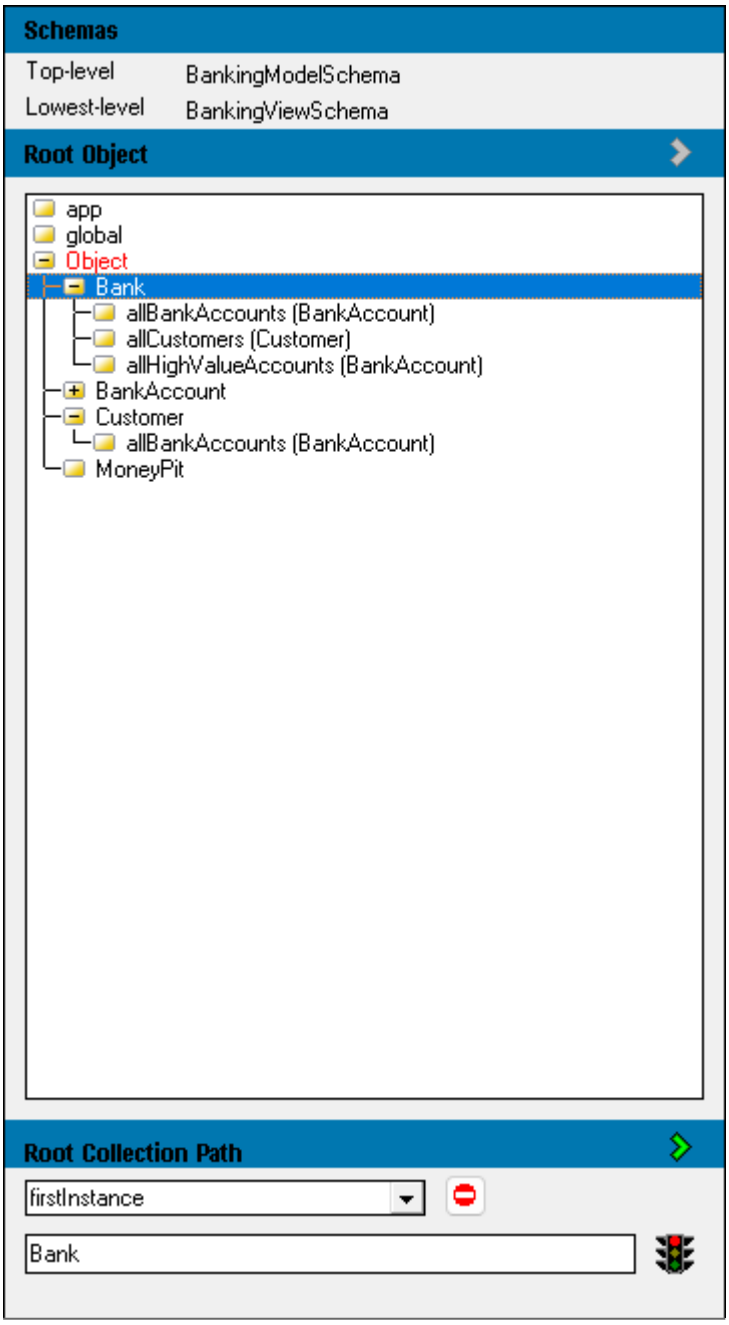
To set the root collections for your reporting view, use the **Root Collections** sheet of the Report Configuration window.

The **Root Collections** sheet contains a representation of all collections in the schemas between the top-level schema and lowest-level schema, inclusive.



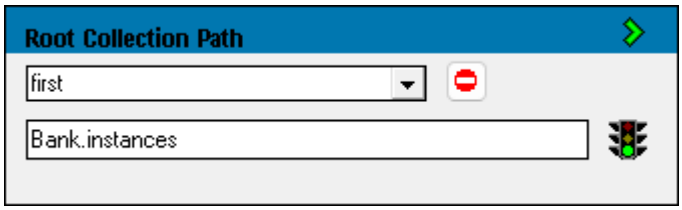
Note Classes themselves are represented by *class-name.instances* (for example, **Client.instances**, and therefore can be selected as a root collection. However, it is more usual to use a collection that has been defined explicitly in the schema.

When selecting a class or collection, the **Root Collection Path** pane will show which reference is currently highlighted, with a traffic light symbol showing whether it is valid as a root collection.

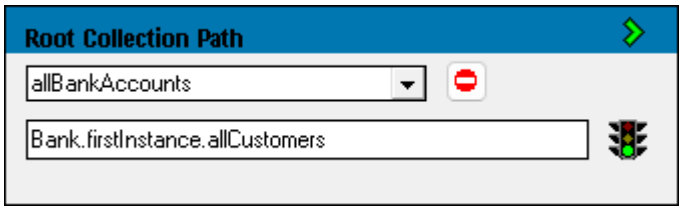


As the **firstInstance** property of the **Bank** class is a singular **Bank** object rather than a collection, this path is invalid, and the traffic light shows red.

As **Bank.instances** is a collection of all the **Bank** objects, the path is valid, and the traffic light shows green. However, this is likely not a useful collection to report on.



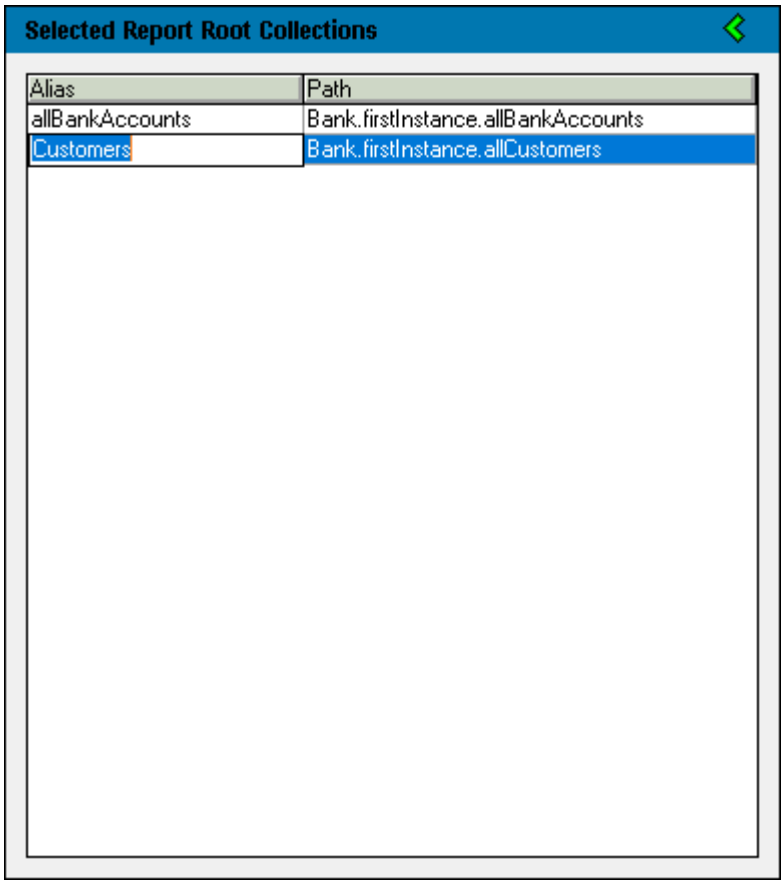
As **Bank.firstInstance.allCustomers** is a collection containing **Customer** objects, the path is valid, and the traffic light shows green.



When you have selected a valid collection to add as a Root Collection, click the green arrow button at the right of the header, to add it to the **Selected Report Root Collections** pane.

Alternatively, you can add it by double-clicking it in the **Root Object** pane.

As with types and features, you can set an alias if you do not want to use the variable name of the collection.



Exercise 1 – Creating a Bank Reporting View

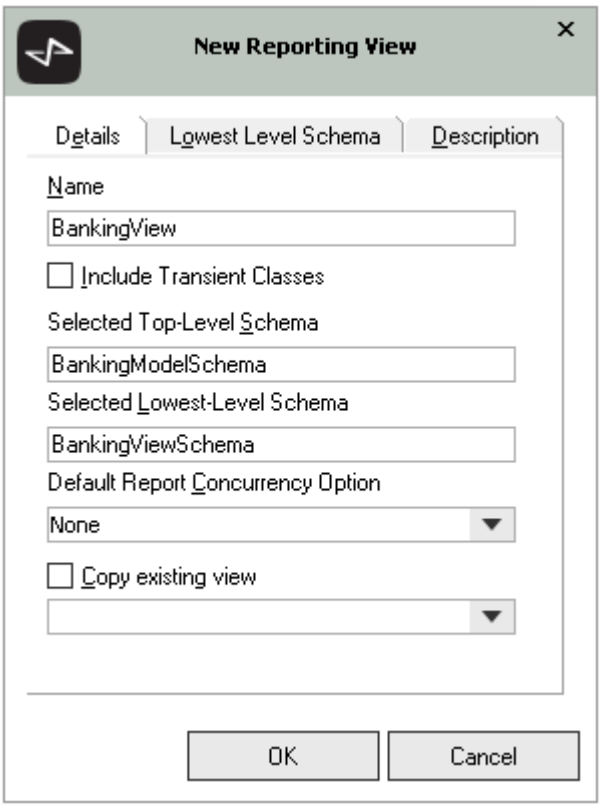
In this exercise, add a reporting view to the Banking system created in the Jade Platform Developer's course.

1. Open the Report Configuration window by coding and running the following **JadeScript** class method.

```
startConfiguration();

vars
    rw : JadeReportWriterManager;
begin
    create rw transient;
    rw.startReportWriterConfiguration("User", null);
epilog
    delete rw;
end;
```

2. Select the **New** command from the View menu, to create a new reporting view.

3. Enter **BankingView** as the name, **BankingModelSchema** as the top-level schema, and **BankingViewSchema** as the lowest-level schema. Leave the default report concurrency option as **None**.

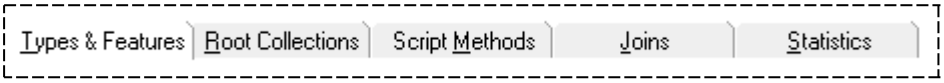


4. Click **OK**.

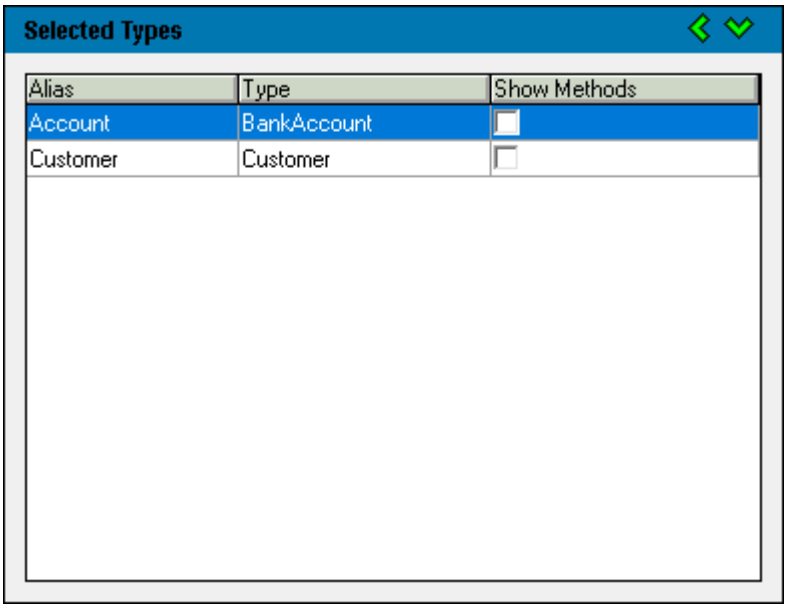
Exercise 2 – Selecting Types and Features

In this exercise, set the classes, methods, and properties to be included in the newly created reporting view.

1. Ensure that the **Types & Features** sheet is selected. (It should be, by default.)

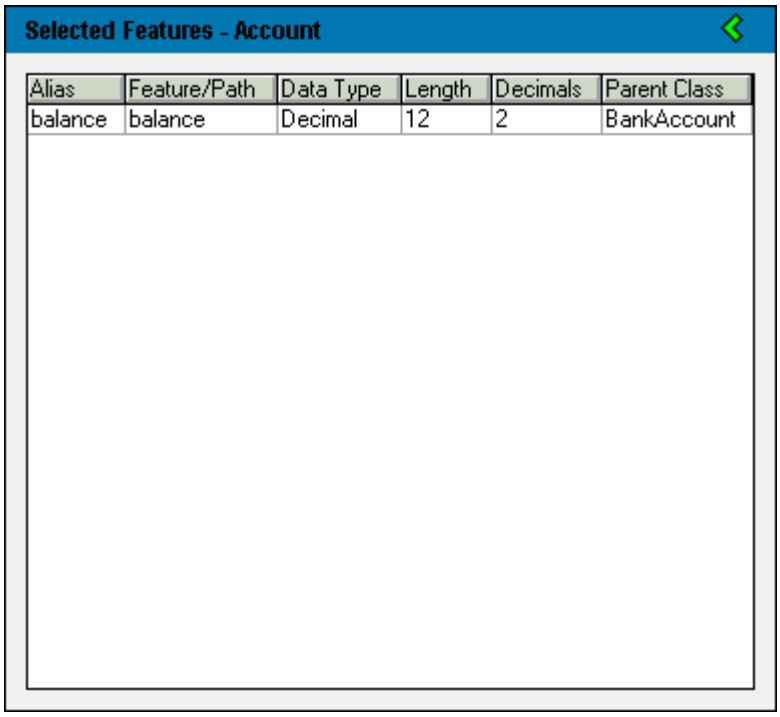


2. Select the **BankAccount** and **Customer** classes and add them to the **Selected Types** pane.
3. Change the alias for **BankAccount** to **Account**.
4. Your **Selected Types** pane should show the following.

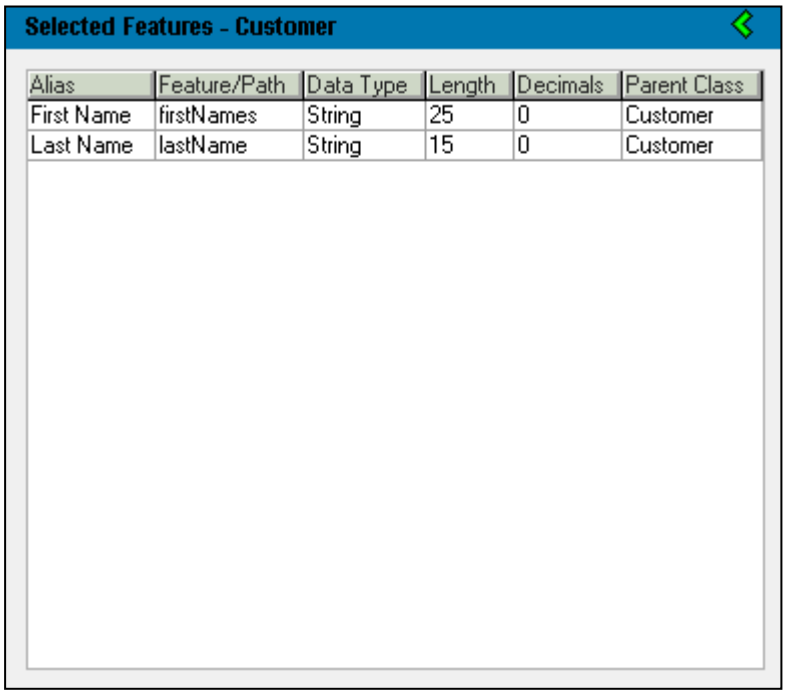


5. For the **Account (BankAccount)** type, add the **balance** feature.

Tip To show the features of **BankAccount**, make sure it is selected in the **Selected Types** pane, or select it from the list in the **Types** pane.



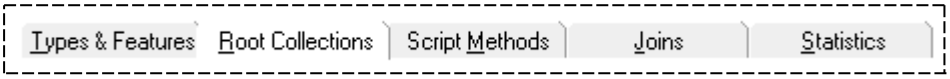
6. For the **Customer** type, add the **firstNames** and **lastName** features.
7. Change the alias of **firstNames** to **First Name** and the alias of **lastName** to **Last Name**.



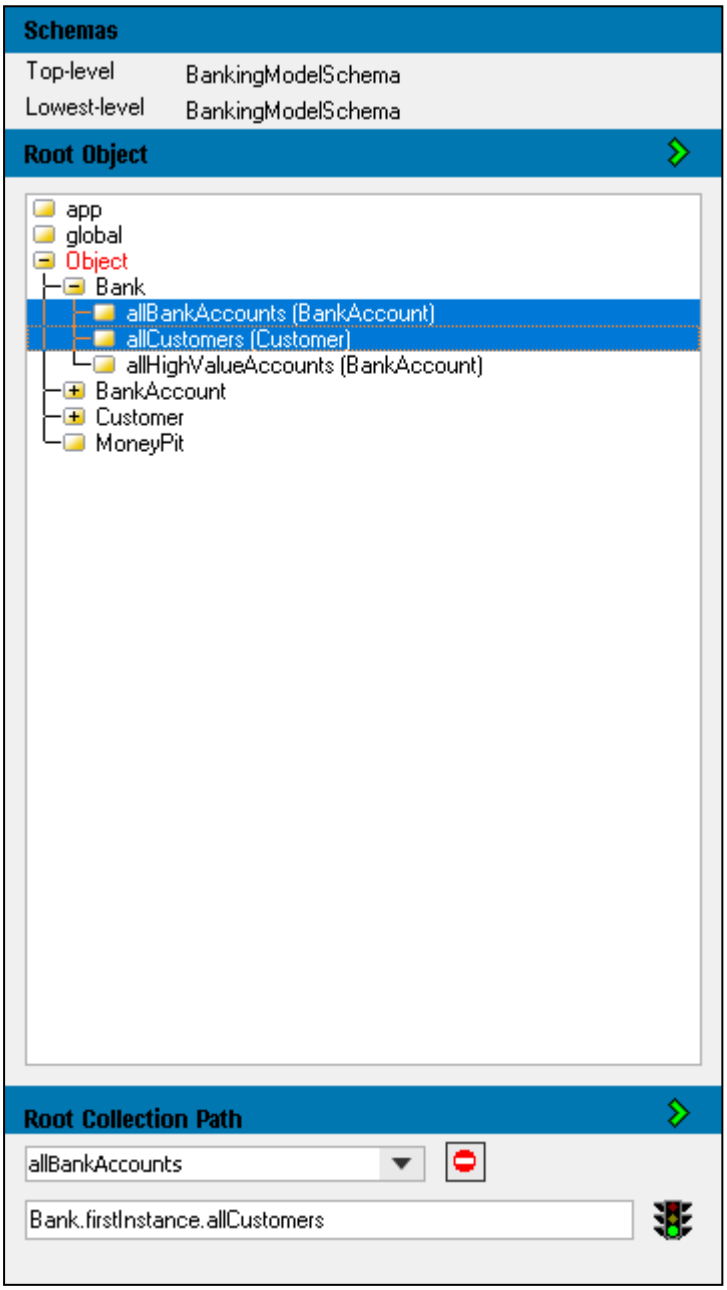
Exercise 3 – Adding Root Collections

In this exercise, add a pair of root collections to the reporting view, to specify the scope of the report.

1. Ensure the **Root Collections** sheet is selected.

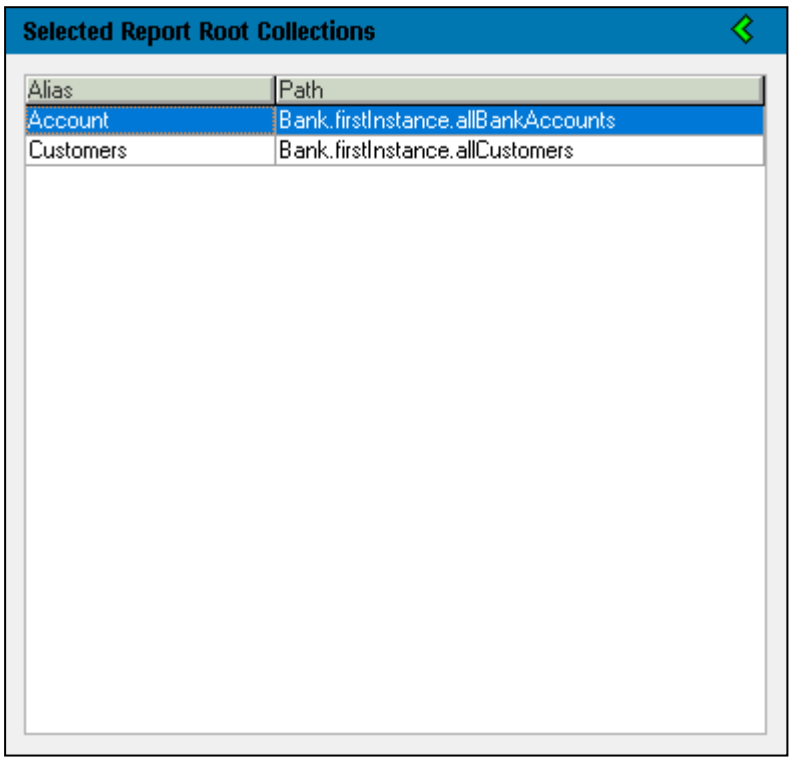


2. Select the **allBankAccounts** and **allCustomers** collections in the **Bank** class.



3. Click the green arrow button at the right of the header, to add them to the **Selected Report Root Collections** pane.

4. Change the alias of **allBankAccounts** to **Account**.
5. Change the alias of **allCustomers** to **Customers**.
6. The **Selected Report Root Collections** pane should appear as follows. (The order of **Customers** and **Accounts** in the table does not matter.)



7. Select the **Close** command from the View menu, and then close the Report Configuration window.

Creating a Report Design

The Report Writer Designer application enables you to define the layout and content of reports based on the data provided by a reporting view of a Jade database. It provides a graphical interface, allowing you to drag and drop text, data from the database, and graphical elements directly onto the report.

While both the Report Configuration and the Report Designer applications can be run directly from the Jade Platform development environment, the Report Designer application has limited functionality when run directly from the development environment, that is, you will be able to create reports but not print, extract, or preview them.

To have full functionality, the Report Designer application should be integrated into the user system from which it reports. The simplest way to do this is from a **JadeScript** class method, as follows.

```
startDesigner();

vars
    rwManager : JadeReportWriterManager;
begin
    create rwManager transient;
    rwManager.startReportWriterDesigner("a user name", null);
epilog
    delete rwManager;
end;
```

You can start a new report from the Welcome dialog or from the **New Report** command on the File menu of the Report Designer window.

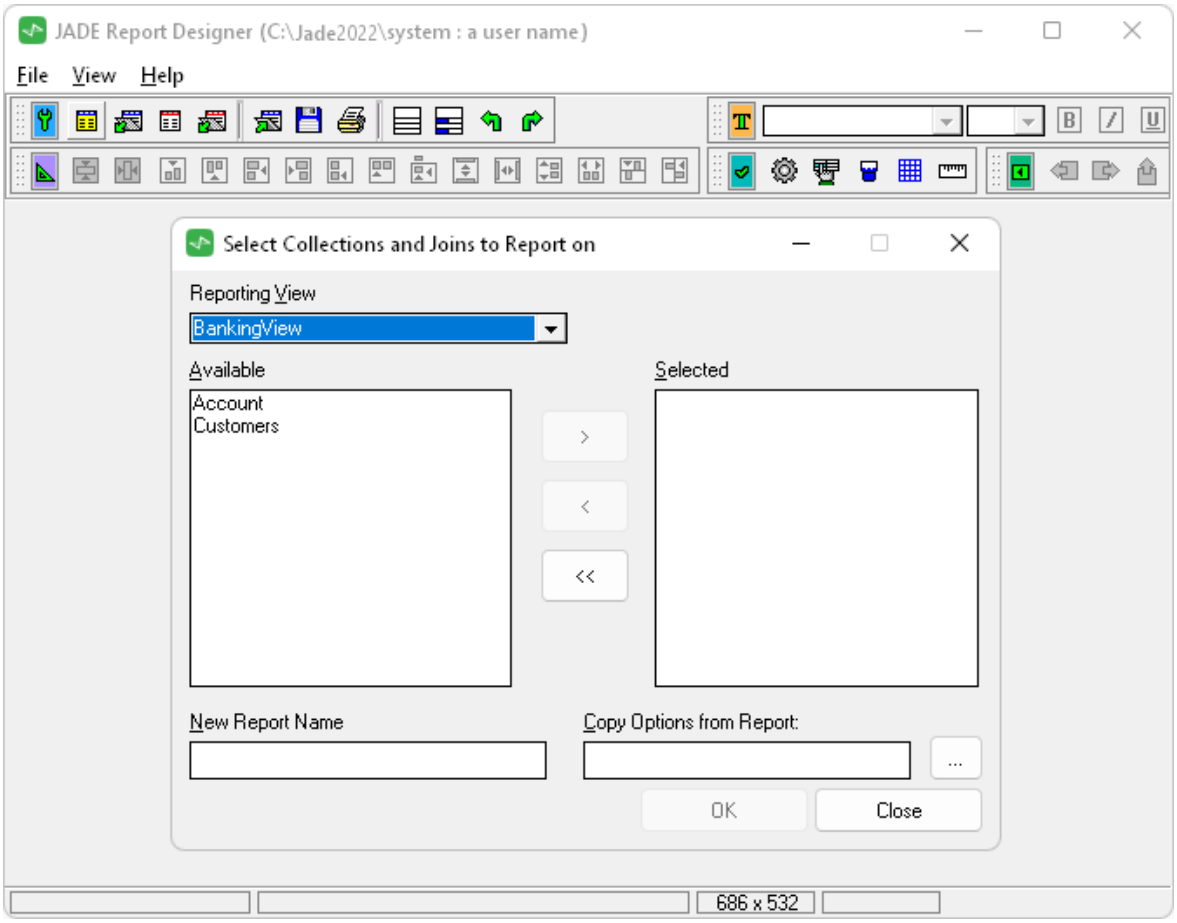
The Welcome dialog is then displayed. Click the **New Report** button at the left of the Welcome dialog and then click **OK** button.

If you want to apply properties from a previously created report, select an existing report in the **Open a Report** list box and then click the **Open a Report** button at the left of the dialog, before clicking **OK**.

To create a report from within the Report Writer Designer application, use any of the following actions.

- Select the **New Report** command in the File menu.
- Click **New Report** toolbar button.
- Press the Ctrl+N shortcut keys.

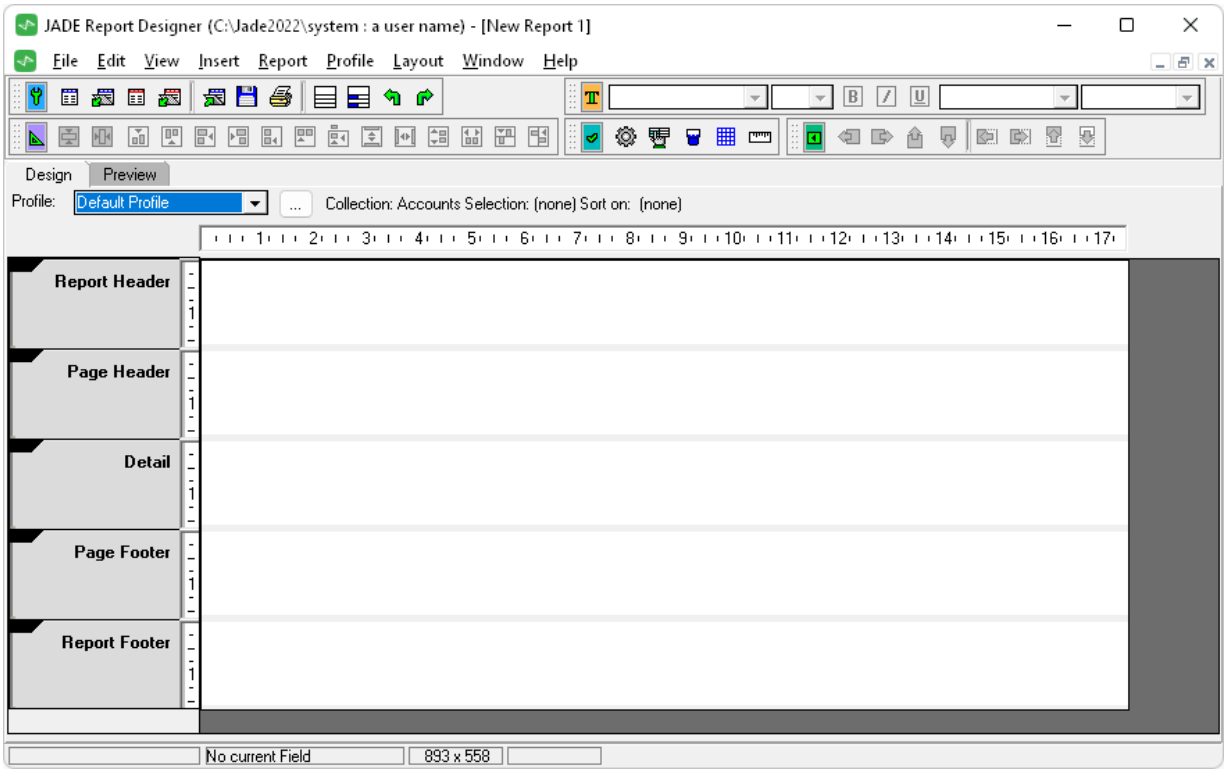
The Select Collections and Joins to Report on dialog is then displayed in the Report Designer window.



If you select a reporting view that has been previously defined, when you click **OK** to begin editing the report, you will need to specify whether to include any joins. (To not use any joins, uncheck the **Join Keys Required?** check box.)

Note While joins are a mainstay of relational databases, they are typically not required in Jade object-oriented databases if good object-oriented principles have been used in the design.

When the Select Collections and Joins to Report on dialog is closed, the Jade Report Designer window, which is the interface for creating reports, is displayed.



The Jade Report Designer window contains the following default sections.

Section	The content in this section is displayed...
Report Header	Once, at the very beginning of the report. Use this section to print the title of the report and any other details you want to be displayed on the front of your report.
Page Header	Once on each page, at the top of each page. If the report header is not printed on its own page, the page header is printed below the report header on the first page of your report. Use this section to print details that you want to appear at the top of every page; for example, date, page number, or field headings.
Detail	For each of the input records provided by the specified root collection or collections. It fills up the space between the page headers and page footers until it presents all elements in the specified root collection or collections, using as many pages as necessary.
Page Footer	Once on each page, at the bottom of each page. Use this section to print details that you want to appear at the bottom of every page; for example, a report identification or date and page number if those are not displayed in the page headers.
Report Footer	Once, at the very end of the report. Use this section to print information such as a grand total of numeric data or other static data that you want to print at the end of the report.

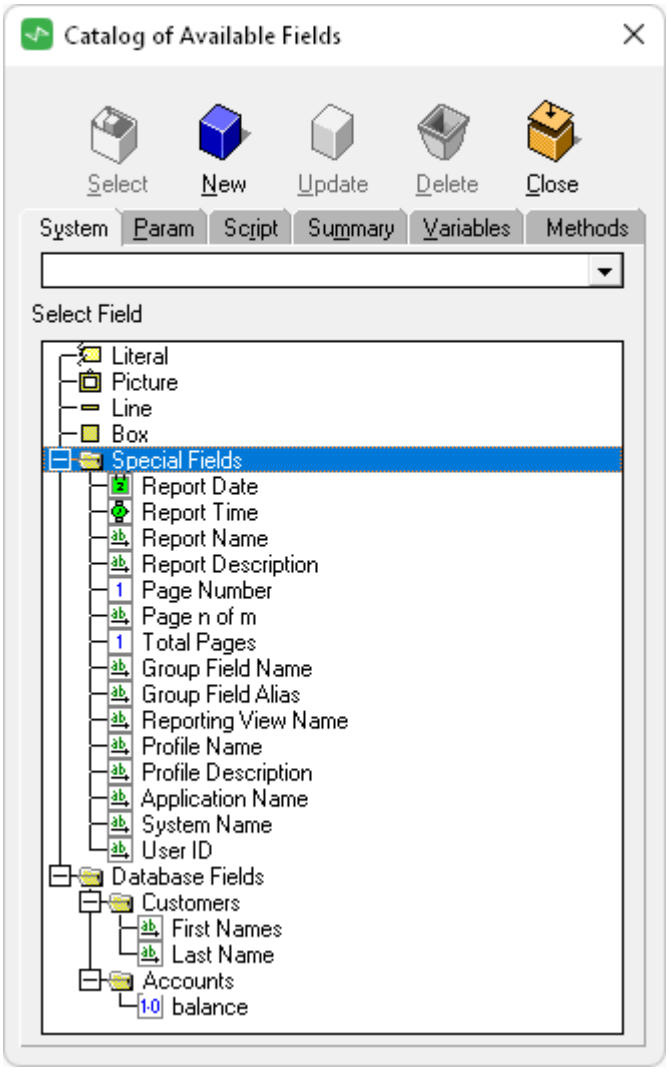
Catalog of Available Fields

The Catalog of Available Fields dialog is used to insert fields into the report. You can open it by performing any of the following actions.

- Select the **Catalog** command from the View menu.
- Click the **Toggle Catalog** button on the **Quick Launch Tools** toolbar.

- Press the F6 shortcut key.

An example of the Catalog of Available Fields dialog is shown in the following image.



This is the major tool you should use to paint fields on your report layout. It also enables you to create, add, and update parameter fields, script fields, summary fields, and method fields and to display the usage of database field items, parameter fields, script fields, summary fields, and method fields.

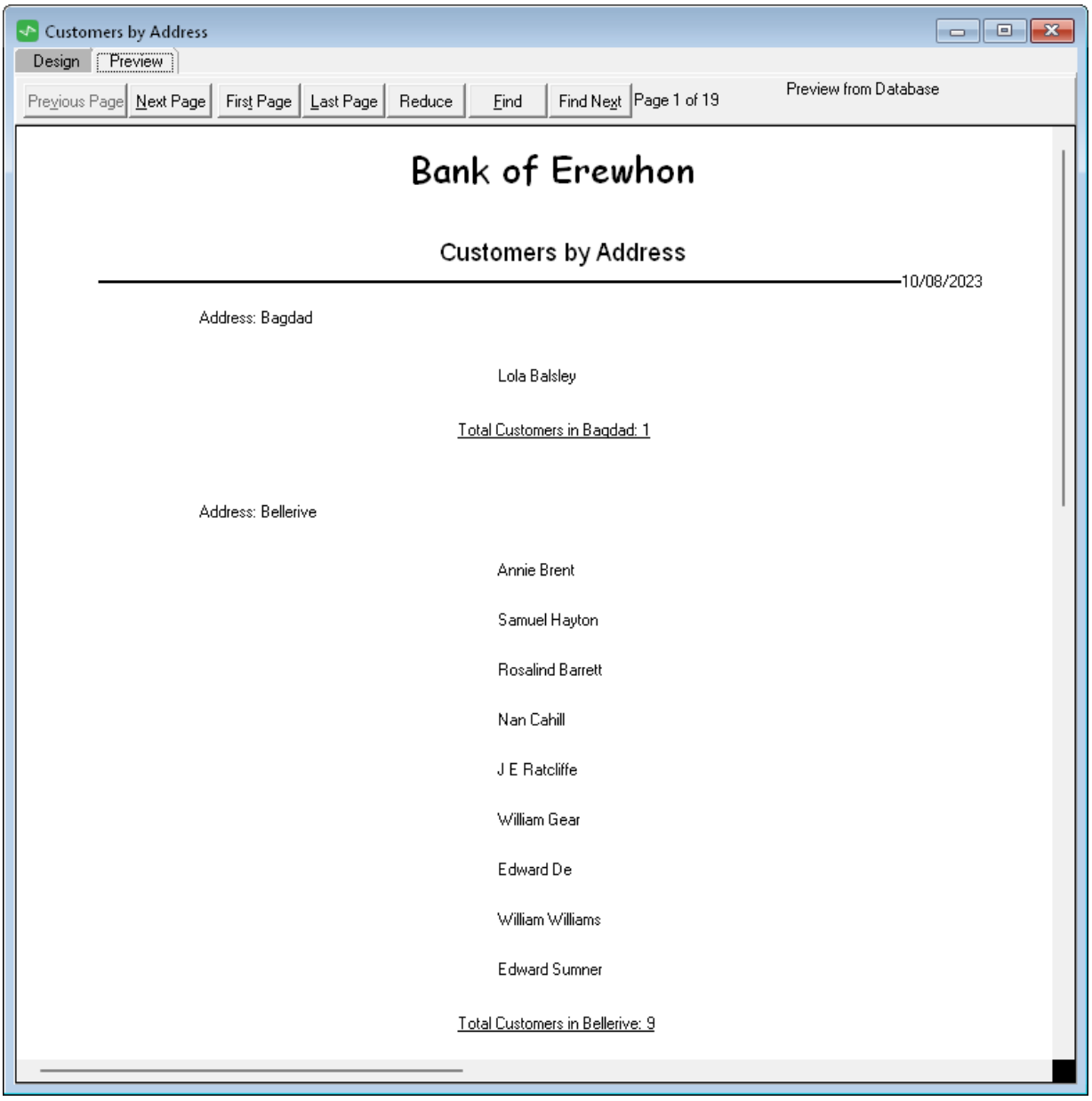
The catalog shows the elements that can be added to the report. These include **Literal** (static text that is not based on any database data), **Line** (a useful divider), **Special Fields** such as the current date and page numbers, and **Database Fields**.

To insert these fields into the report, click and drag them onto the report.

Viewing the Jade Report

You can preview Jade Reports directly in the Report Designer window, print to a printer or to PDF, or export to a variety of formats.

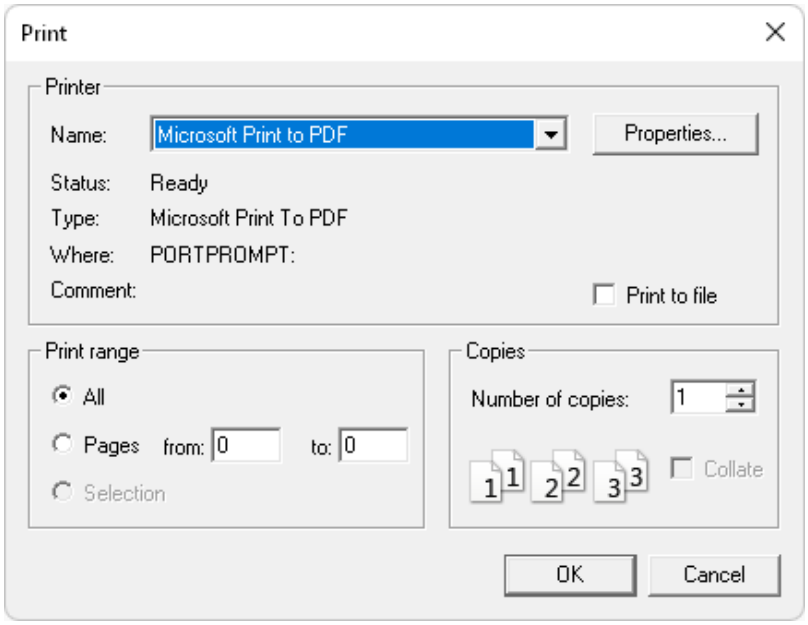
To preview a Jade Report from the Report Designer window, simply select the **Preview** tab of the Report Designer window. The output of the result is displayed in the current window.



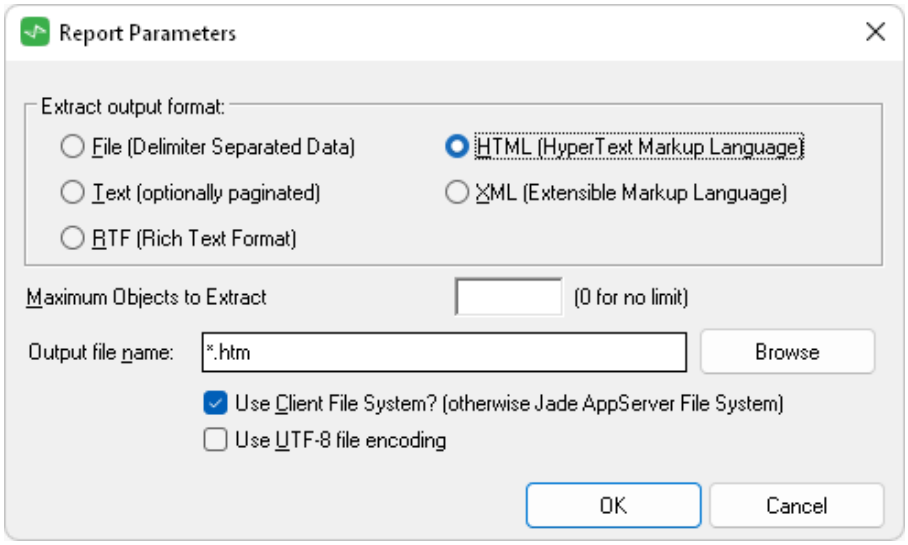
To print a Jade Report to printer or PDF file, open the print menu using one of the following actions.

- Select the **Print** option from the File menu.
- Click the **Print** toolbar button.
- Press the Ctrl+P shortcut keys.

The common Print dialog is then displayed, from which you can select where to print, along with the number of copies and the range of pages to print.



To export the report to another format (for example, a .**csv**, .**txt**, or .**htm** file), open the Report Parameters dialog by selecting the **Extract Data** command from the File menu.



From this dialog, you can select the output format from the following options.

Format	Extension	Descripion
File (Delimiter Separated Data)	.csv	This format creates a Comma-Separated Values file (.**csv**) but does not require a comma specifically as the delimiter. You can select any symbol to use as a delimiter if commas are likely to be found in your data.

Format	Extension	Descripion
HTML	.html	This format applies HTML tags to your data to format it for display on the Internet as an .**htm** file.
Text (optionally paginated)	.txt	This extracts your report as simple text. Any formatting is lost, and layout is approximated using spaces.
XML	.xml	This format applies XML tags to your data to format it for display on the Internet as an .**xml** file.
RTF	.rtf	This extracts your report as a Rich-Text Format file, which encodes formatted text and graphics in a universal format.

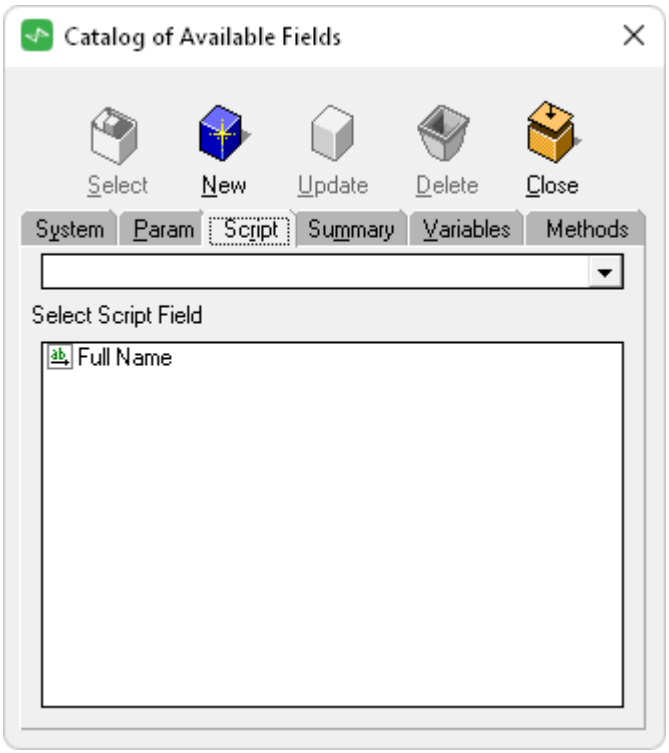
The **Maximum Objects to Extract** text box lets you set a limit on the size of the extraction for very large databases (for example, you may want to extract only the first ten thousand customers).

The **Use Client File System** check box is meaningful only when running the Report Writer from a presentation (thin) client and that client is on a different file system to the application server. For single user and standard (fat) client systems, this will not be the case. However, if you want to extract a report from a remote presentation client to the file system of the application server, this check box enables you to do so.

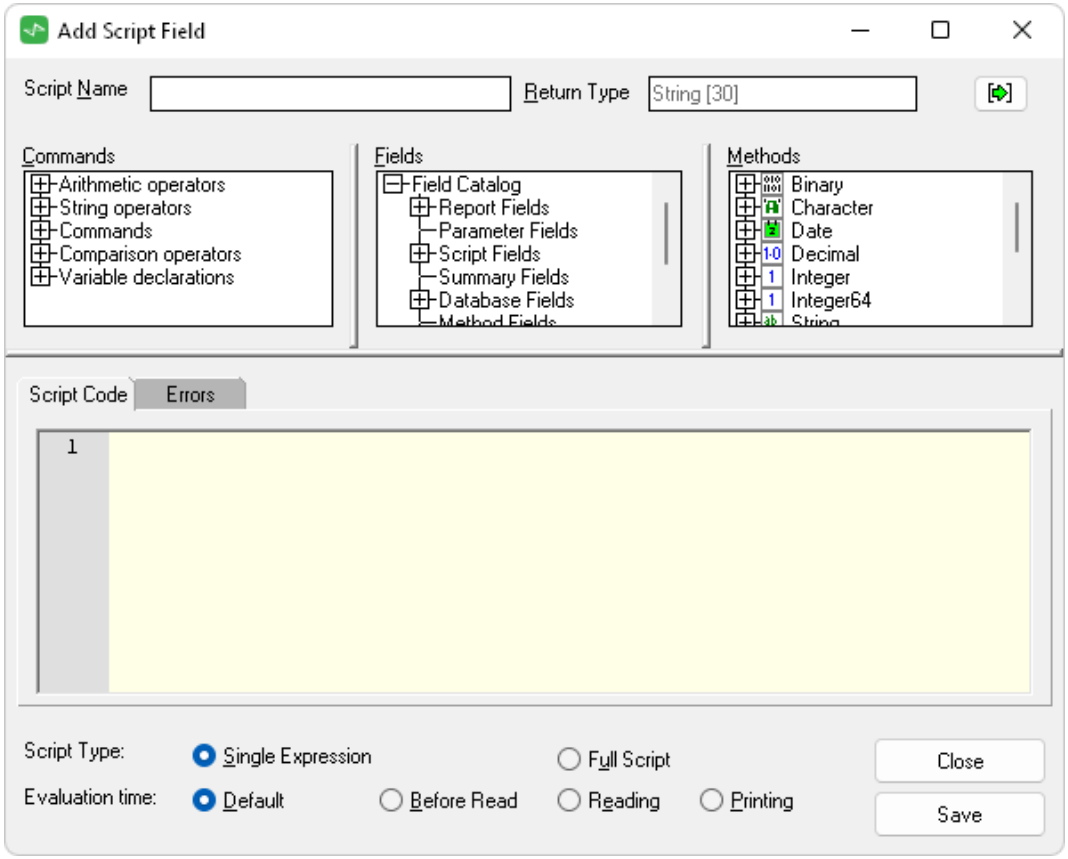
The **Use UTF-8 file encoding** check box lets you use UTF-8 encoding in the output report. (This check box is disabled if you select the **RTF** option button in the Extract output format group box.)

Report Scripts and Combining Data

When designing a report, it is often useful to create a reporting field that is composed of multiple database fields. To do this, the Catalog of Available Fields dialog has the **Script** sheet.



To add a new script to the Catalog, click the **New** button while the **Script** sheet is displayed. The Add Script Field dialog is then displayed.



While the **Full Script** option button for **Script Type** allows you to create powerful **JadeScript** class methods directly from your report, for simple combining of database fields, it is faster and easier to use **Single Expression** option button.

The Add Script Field dialog combines the ability to write Jade syntax in the **Script Code** text box with easy-to-use GUI controls for users who are less familiar with Jade code.

Tip While the **Full Script** option button is selected in **Script Type**, the **Script Code** text box behaves like a **JadeScript** class method.

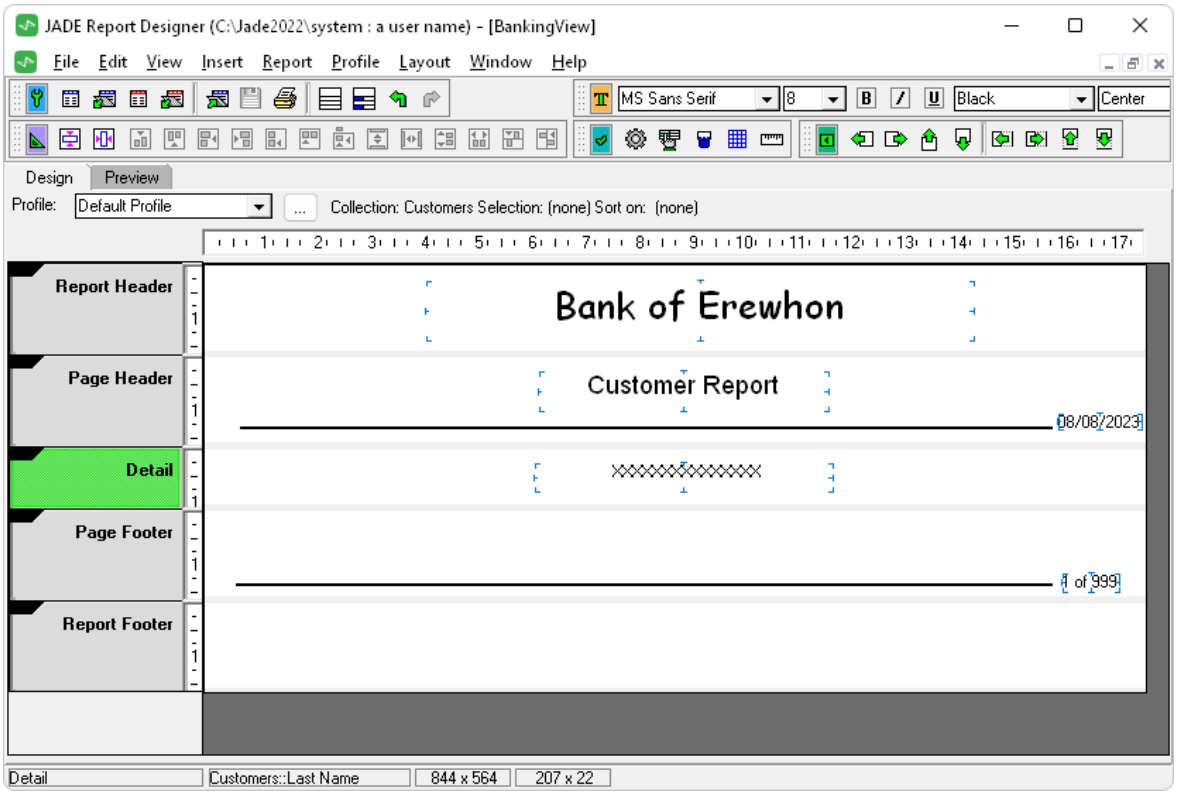
Exercise 4 – Creating a Report

In this exercise, create a report based on the banking system view created in the earlier exercises in this module.

1. Add the following **JadeScript** class method to the **BankingViewSchema**.

```
openDesigner();

vars
    rw : JadeReportWriterManager;
begin
    create rw transient;
    rw.startReportWriterDesigner("User", null);
epilog
    delete rw;
end;
```

2. Run the **JadeScript** class **openDesigner** method and then select **New Report**.
3. Add **Customers** to the **Selected** list box, name the report **BankingView**, and then click **OK**.
4. Create the following form.



   a. Add **String** literals such as **"Bank of Erewhon"** or **"Customer Report"** by dragging a literal from the **System** sheet of the Catalog of Available Fields dialog to the report.

   b. Draw a line by dragging **Line** from the **System** sheet of the Catalog of Available Fields dialog to the report.

      The cursor will become a line-drawing cursor that you can click and drag across the report to draw lines.

c. Add the current date to the report by dragging a **Report Date** from **Special Fields** on the **System** sheet of the Catalog of Available Fields dialog to the report.

d. The **"XXXXXXXXXXXXXXXXXX"** element represents data from the database. Add data from the database by dragging one of the elements from **Database Fields** on the **System** sheet of the Catalog of Available Fields dialog.

   For this example, use **Last Name** in the **Customers** folder.

e. Add the page number by dragging **Page n of m** from **Special Fields** on the **System** sheet in the Catalog of Available Fields dialog.

Tip If the Catalog of Available Fields dialog is not visible, enable it by pressing F6, clicking the **Toggle Catalog** button on the **Quick Launch Tools** toolbar, or by selecting the **Catalog** command from the View menu.

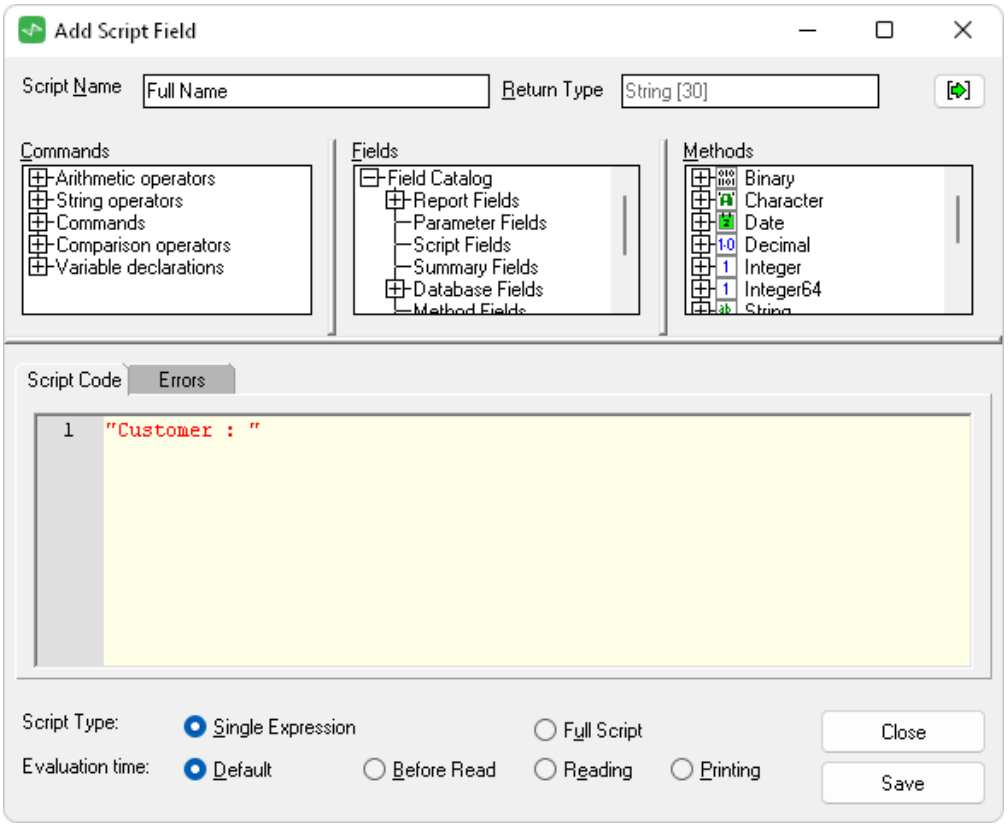
5. To view a preview of the report, select the **Preview** tab to display the **Preview** sheet of the Jade Report Designer window.

Exercise 5 – Report Scripts

In this exercise, use a single expression script to create a composite of multiple database fields and a **String** literal to improve the details reported on each customer in the BankingView report. The goal is to output **"Customer : *firstName lastName*"**.

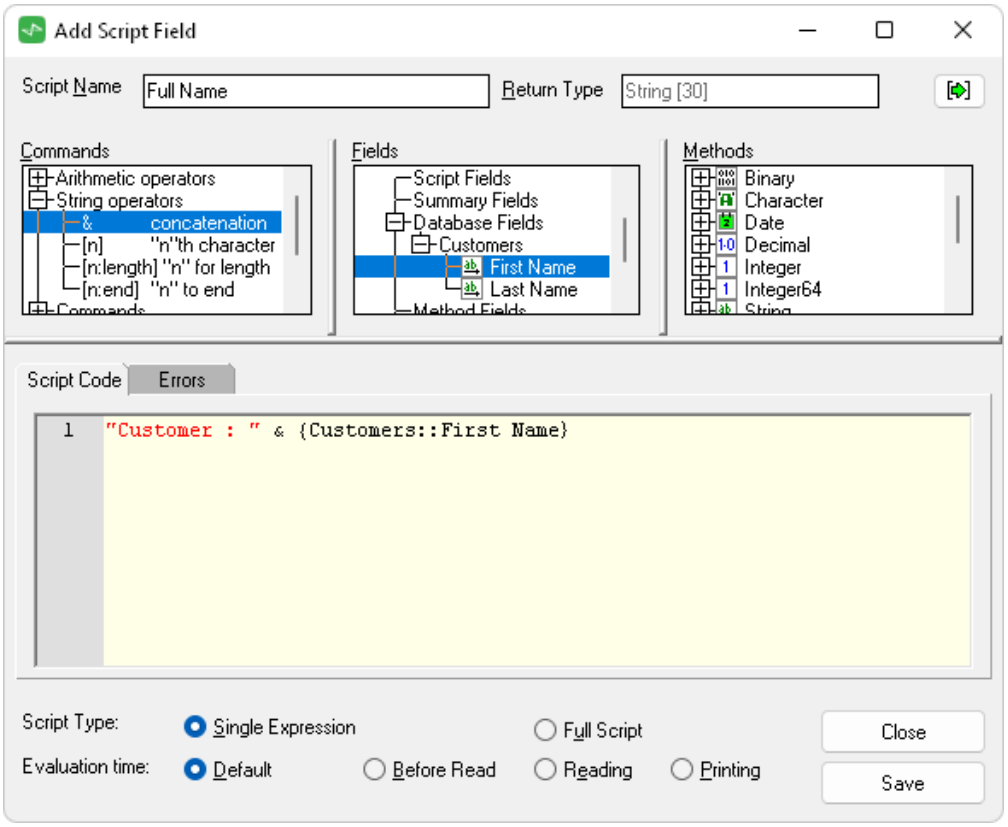
1. Select the **Script** sheet of the Catalog of Available Fields dialog.
2. Click **New**, to create a new script.
3. Call the script **Full Name**.

4. In the **Script Code** text box, specify **"Customer : "**.Your form should look like the following.



5. Double-click the **& concatenation** option under the **String operators** entry in the list of **Commands**. This will add a **&** character to the script, which concatenates two strings together.

6. Double-click the **First Name** field in the **Customers** option within **Database Fields** in the list of **Fields**.

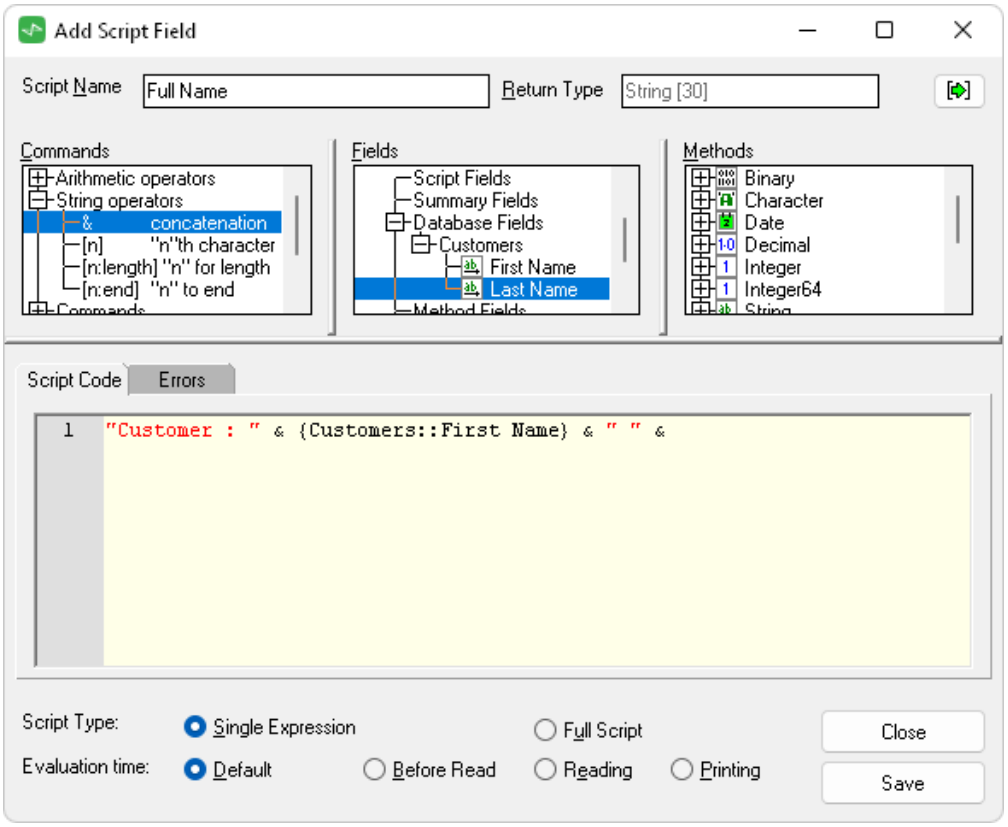


7. Add another string concatenation to the expression. You can select it from the **Commands** list or simply type **&**.

Note If you simply add **Last Name** now, **First Name** and **Last Name** will be concatenated together; for example, **"Customer : BarbaraBayton"**.

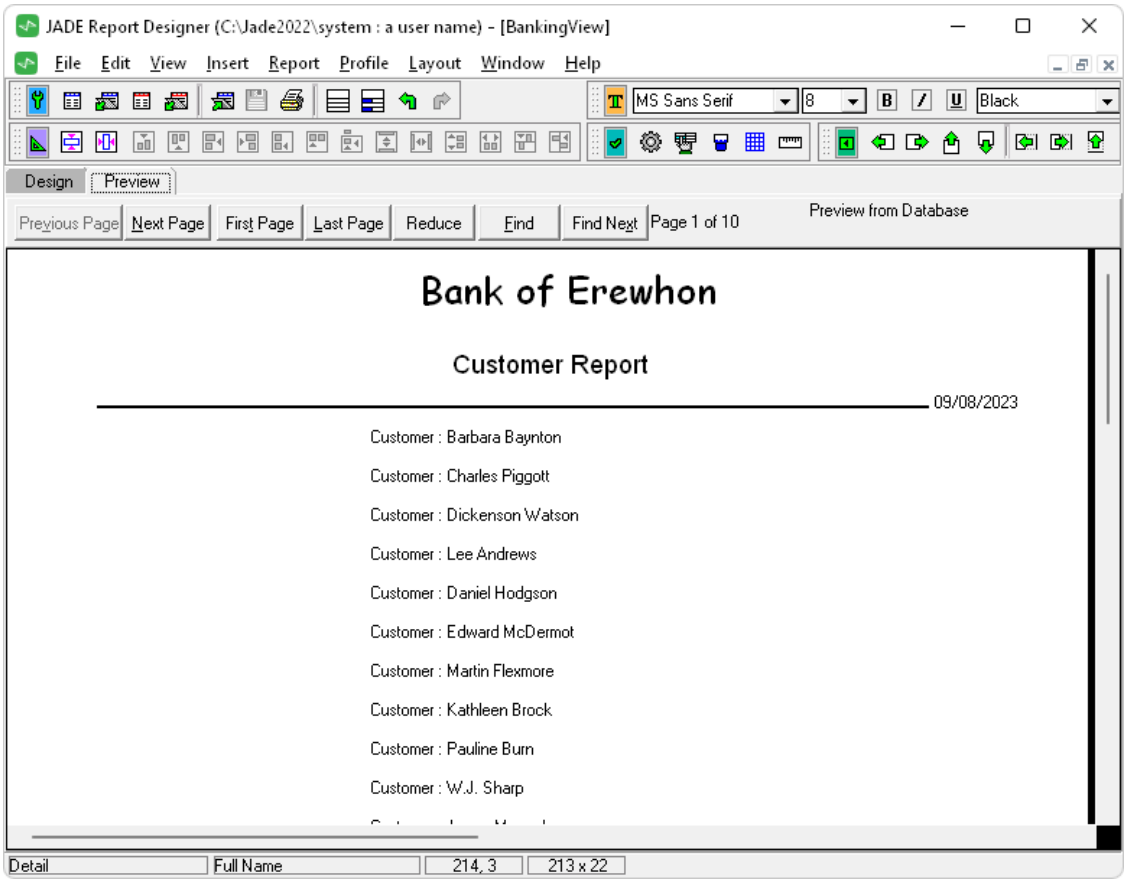
8. To add a space after the **First Name** and before the **Last Name**, type **" "** and another **&**.

The form should now look like the following.



9. Finally, select **Last Name** from the **Fields** list and then click **Save**.

10. To see your changes, select the **Preview** tab to display the **Preview** sheet of the Report Designer window.

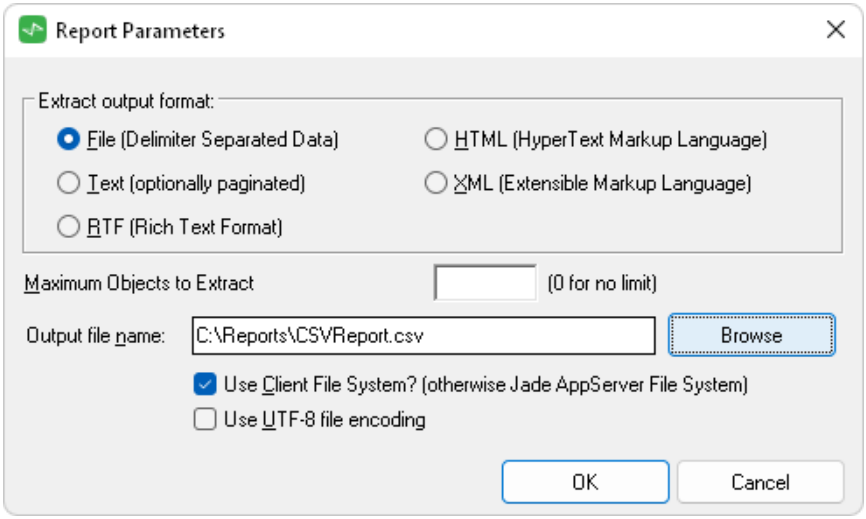


Exercise 6 – Extracting a Report

In this exercise, extract the **BankingView** report to a variety of formats.

1. Select the **Extract Data** command from the File menu.

   The following dialog is then displayed.



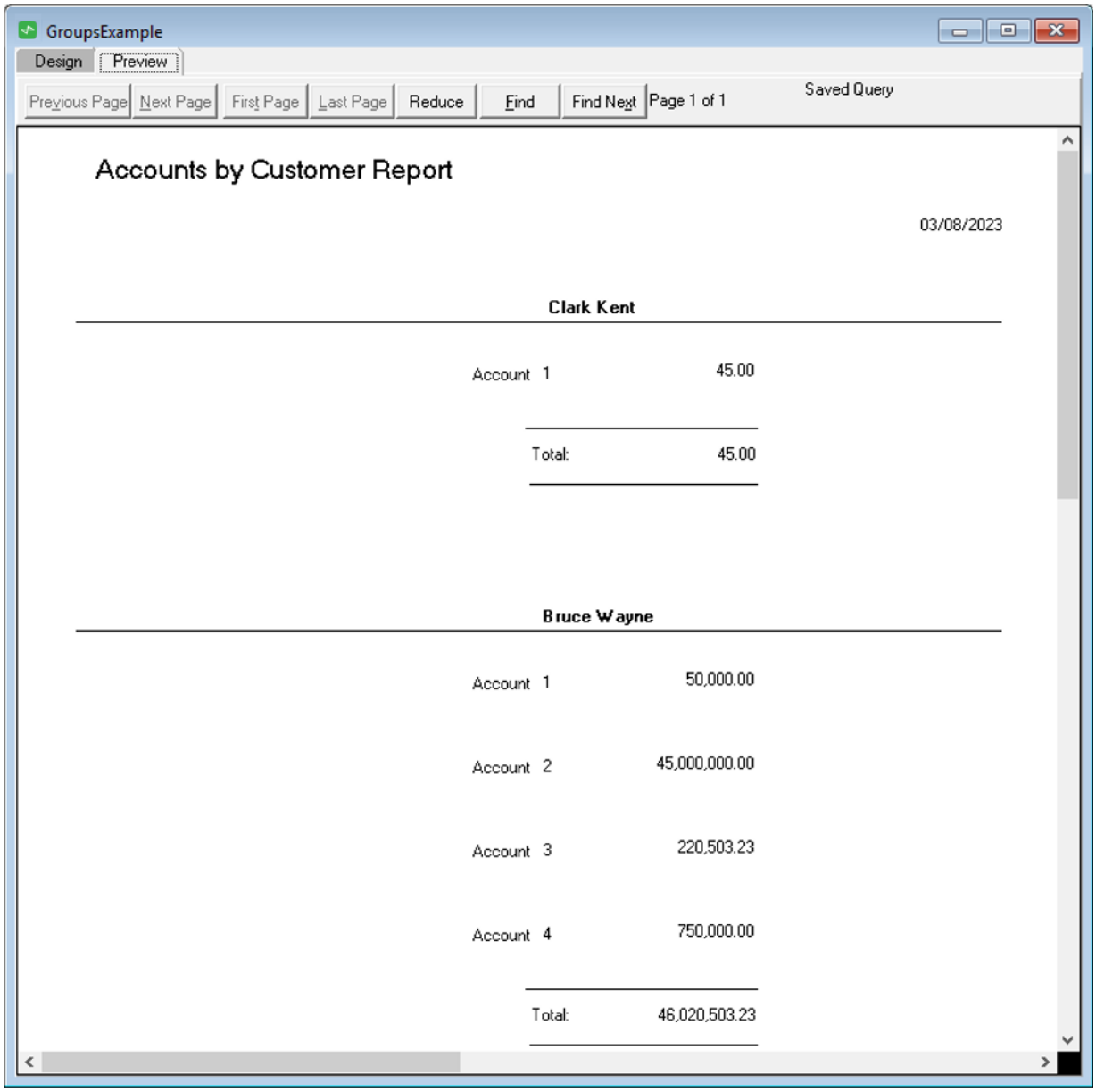
2. Select the **File (Delimiter Separated Data)** option button.
3. Click **Browse** at the right of the **Output file name** text box, to locate a convenient folder; for example, **C:\Reports**.
4. Ignore the **Maximum Objects to Extract** text box, and **Use Client File System** and **Use UTF-8 file encoding** check boxes.
5. Click **OK**, to extract the report as a CSV file.
6. Repeat steps 1 through 5 of this instruction for each of the available output formats.
7. View the generated files.

You will see that some look similar to the preview (for example, HTML output when viewed in a browser) and some different (for example, CSV output).

Report Writer Groups

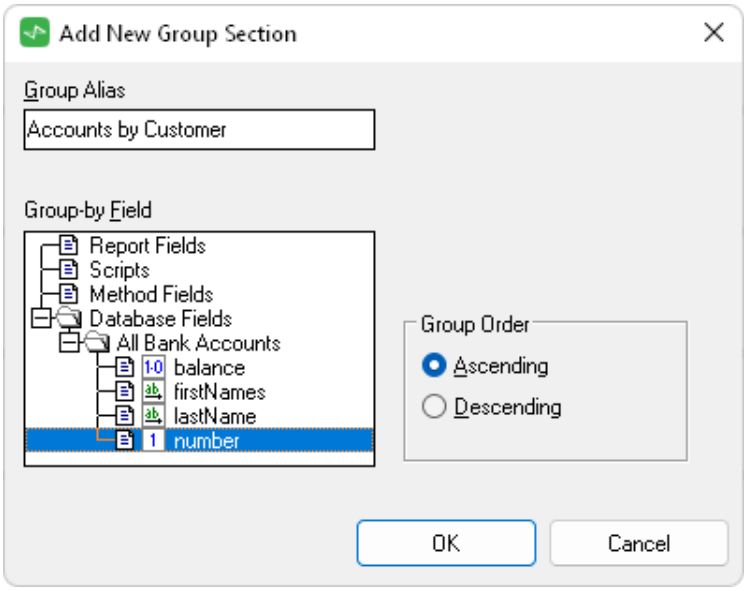
Report groups enable you to group data by subcategory and optionally produce summaries of the data within each group.

This can be useful for reporting on one-to-many relationships, grouping all data of the *many* property by a unique identifier of the *one*. For example, you can group all bank accounts by their customer's ID number, to display the bank accounts of each customer in the bank.

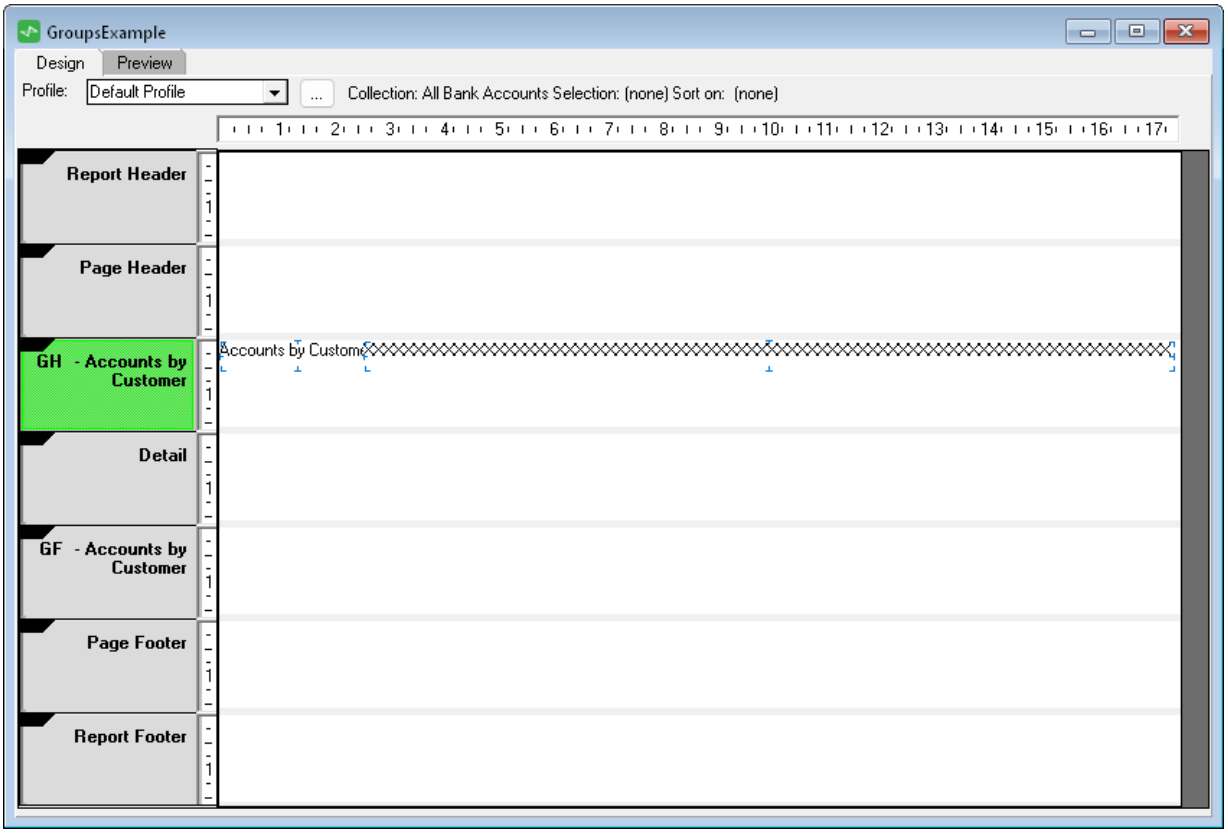


To create a group, select the **Group** command in the Insert menu.

The Add New Group Section dialog is then displayed.



You can name the group with an alias and select any of the features selected in the reporting view.

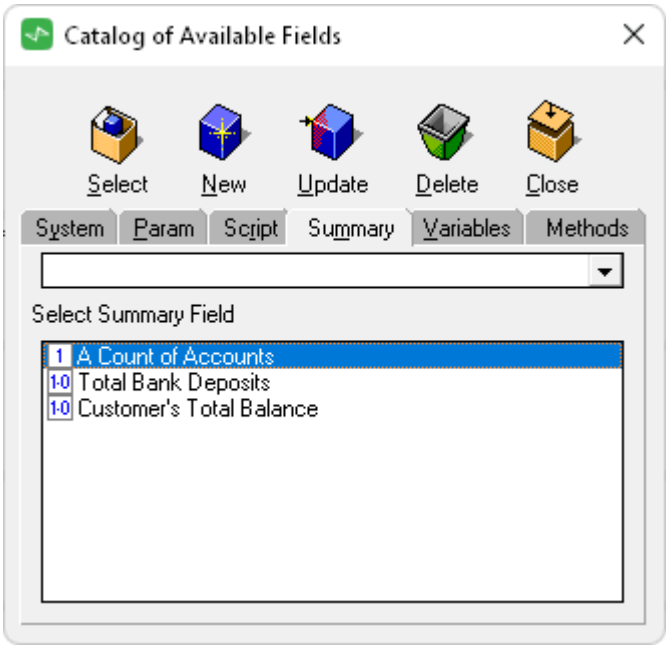


The report will surround the **Detail** section with new sections for the group - a group header above it and a group footer below it. By default, the header contains the alias of the group and the value of the selected feature for each group entry. You can modify this to meet your requirements; for example, if you want to put the name of the customer instead of the ID number, or to change the font. Although the group footer is empty, by default, it is however is a good place to put data summaries; for example, the total balance of the bank accounts of each customer.

In the detail section, you should put the data that you want shown in each grouping; for example, the balance of each account.

The Summary Sheet

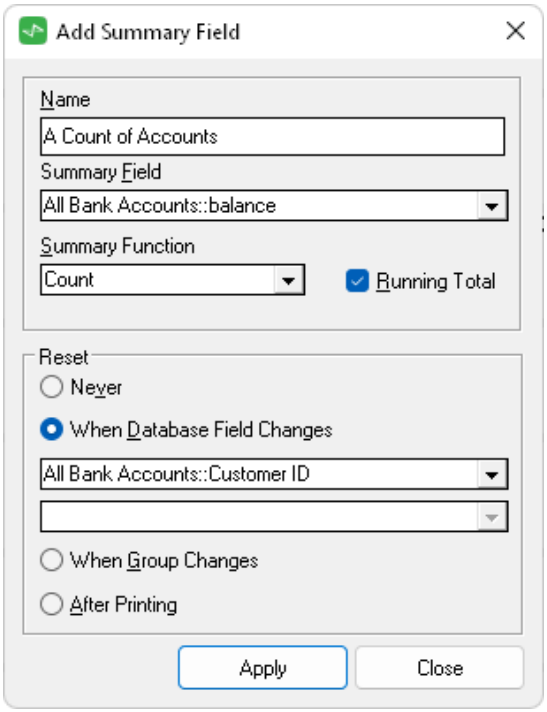
The **Summary** sheet of the Catalog of Available Fields dialog is used for producing totals and averages of sets of data. These summaries can be inserted into the report like any other element and can provide useful information such as the total balance of the accounts of a customer or a running total of how many accounts each customer has.



To create a new summary, use any of the following actions from the Catalog of Available Fields dialog.

- Click the **New** button.
- Press the Alt+N shortcut keys.

The Add Summary Field dialog is then displayed, from which you can create the new summary.



The value of the **Summary Field** combo box is the feature of the collection objects to be summarized. Although this can be any feature, the entities available in the **Summary Functions** combo box depend on the type of the value selected in the **Summary Field** combo box.

The **Summary Function** is the manner in which the **Summary Field** is to be summarized, and can be one of the following values.

Function	Description
Count	Total number of instances within the collection of the selected field.
Distinct Count	Number of unique values amongst the instances within the collection of the selected field.
Non-null Count	Number of instances that are not null within the collection of the selected field.
Max	Highest value found amongst the instances within the collection of the selected field. This can be a numerical maximum or an alphabetical maximum, depending on the type of the field.
Min	Lowest value found amongst the instances within the collection of the selected field. This can be a numerical minimum or an alphabetical minimum, depending on the type of the field.
Sum	Total of all the values of the target field within the collection. This can be applied only to numerical fields.
Average	Mean value of all the values of the target field within the collection. This can be applied only to numerical fields.
Non-Zero Average	Mean value of all the values of the target field within the collection, with zero-values skipped. This can be applied only to numerical fields.

The **Running Total** check box is used if the summary needs to be updated continuously between each item of the collection. As this incurs a performance cost, you should not check this if you are presenting the summary only at the end of a group or at the end of the report.

The Reset group box in the lower half of the dialog is used to select when the summary is reset. For a summary of the whole report, select **Never** or **After Printing**. These values behave identically except when performing multiple print runs within the same Report Writer Designer session, in which case the **After Printing** option button resets the total between print runs.

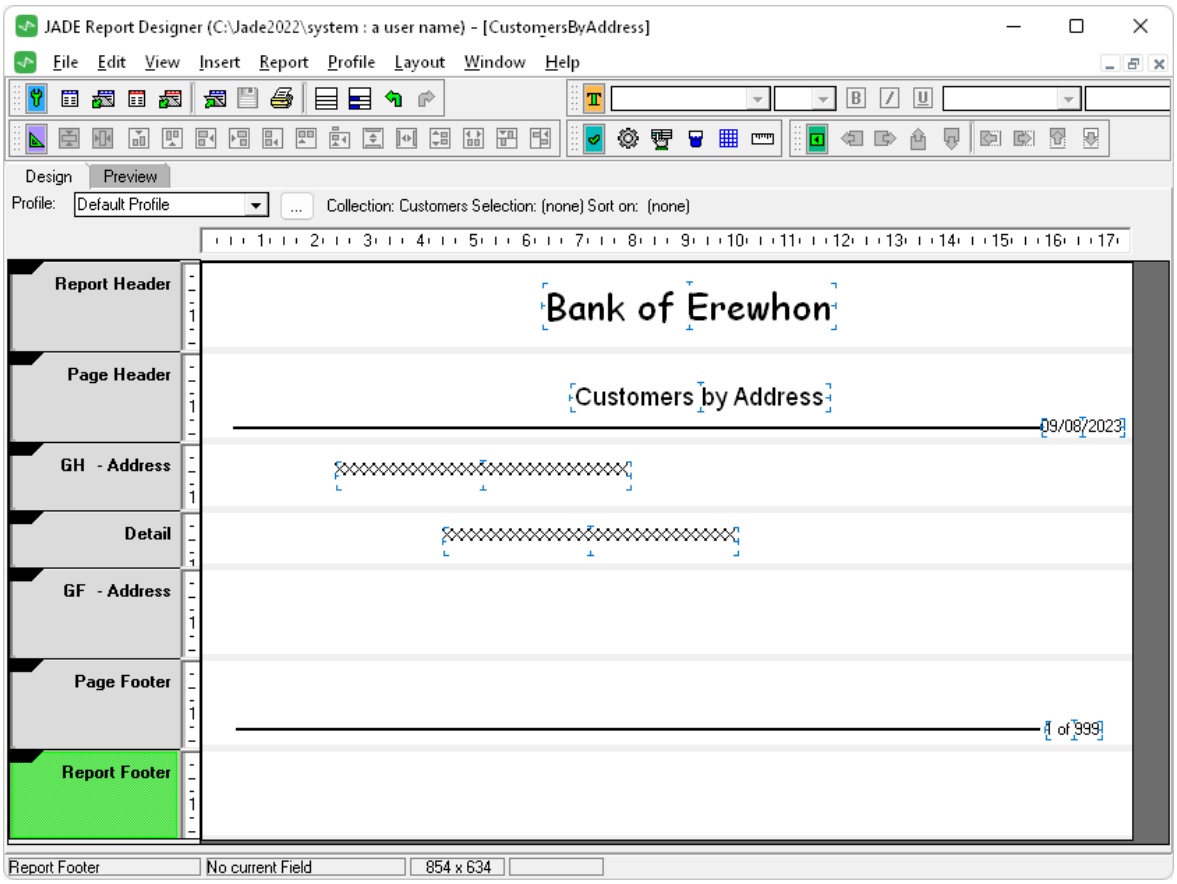
To create a summary for each group when using the **Groups** feature, select the When **Group Changes** option button, which resets the summary between each group of the collection.

You can also specify the summary to reset when a specific database field changes, by selecting the **When Database Field Changes** option button.

Exercise 7 – Grouping Data

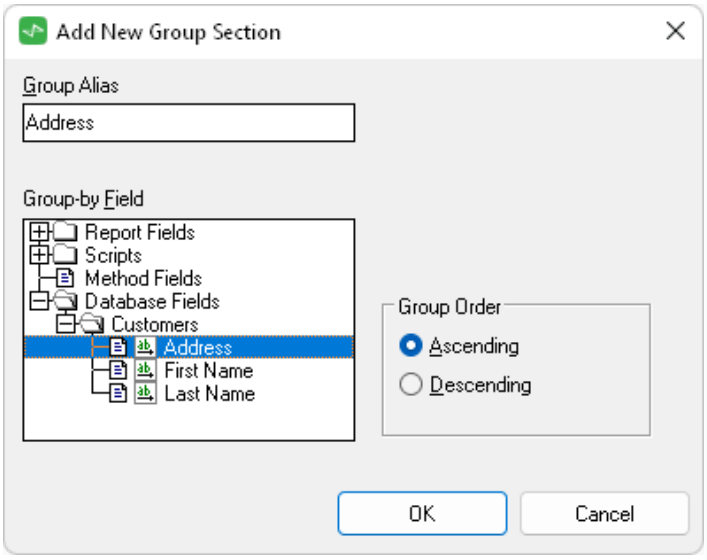
In this exercise, design a new report of all **Customers**, grouped by **Address**.

1. Open the Report Configuration application by running the **JadeScript** class **startConfiguration** method.
2. Select the Open command from the **View** menu.
3. Open the **BankingView** configuration.
4. In the **Customer** class, add the **address** feature and change its alias to **Address**.
5. Close the Report Configuration window and then open the Report Designer window (by running the **JadeScript** class **startDesigner** method).
6. Create a new report called **CustomersByAddress**, with **Customers** selected.

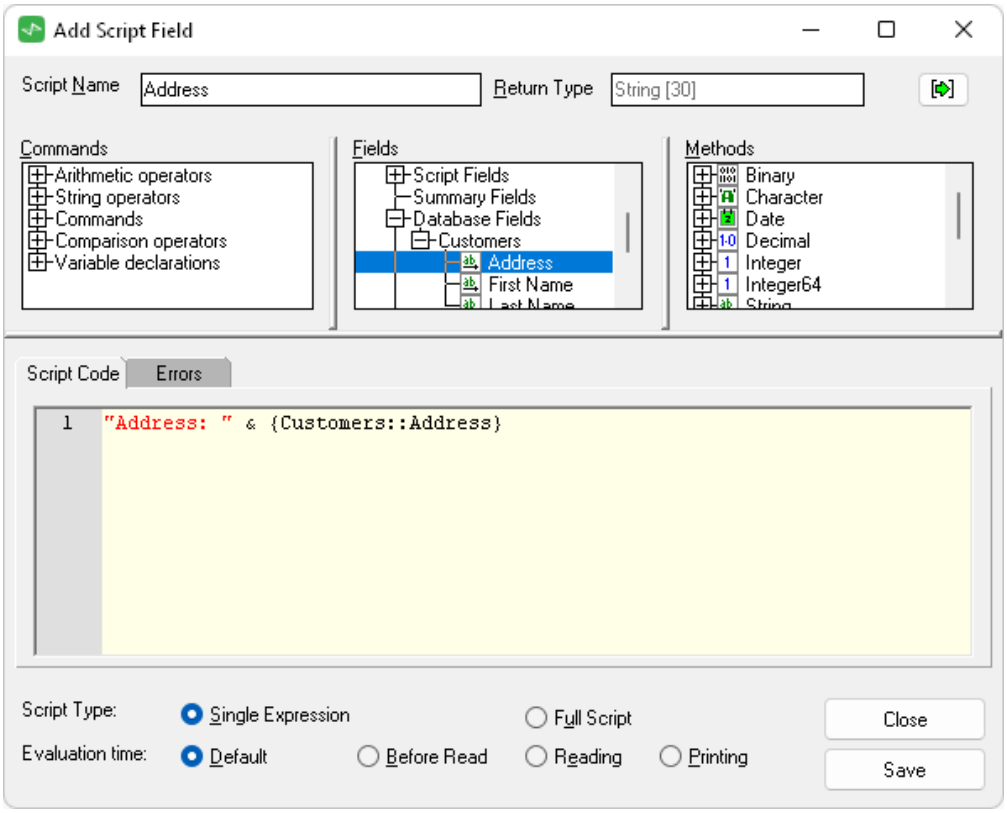


7. Design the report, as follows.

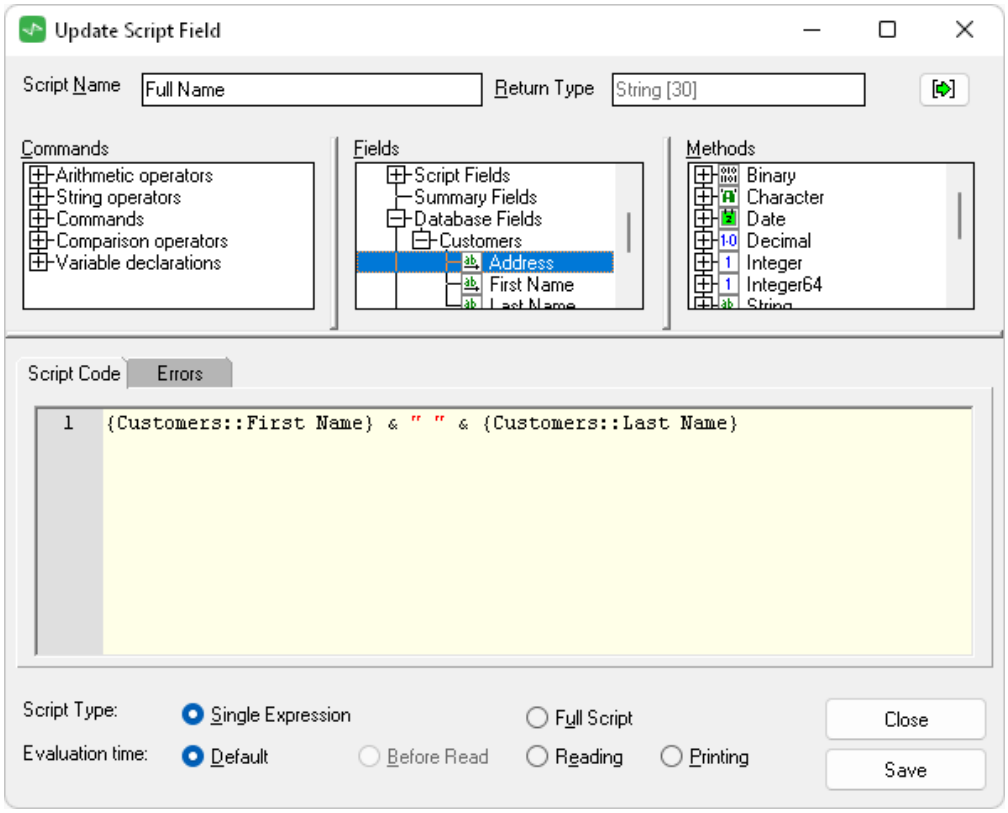
   a. The **GH - Address** and **GF - Address** are the header and footer for the **Address** grouping. To add the **Address** grouping, select the **Group** command from the Insert menu and then fill it out as shown in the following dialog.



   b. By default, the **Group Header** contains a **String** literal of the name of the field (**Address**, in our case) and the selected **Database Field** value. These should be replaced by a **Single Expression Script Field**, to format the data more appropriately.



c. The **Detail** section should contain the following **Single Expression Script Field** value.



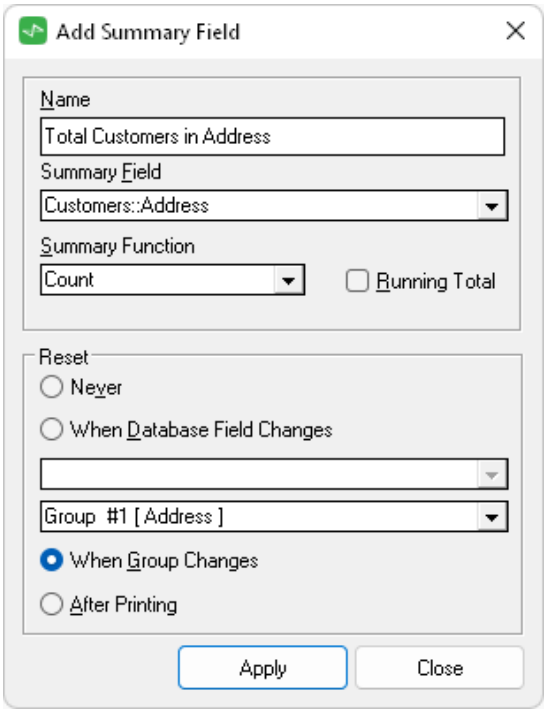
8. To view the report, select the **Preview** tab of the Jade Report Designer window.

Exercise 8 – Summarizing Data

In this exercise, use the **Summary** feature to count the number of customers at each address.

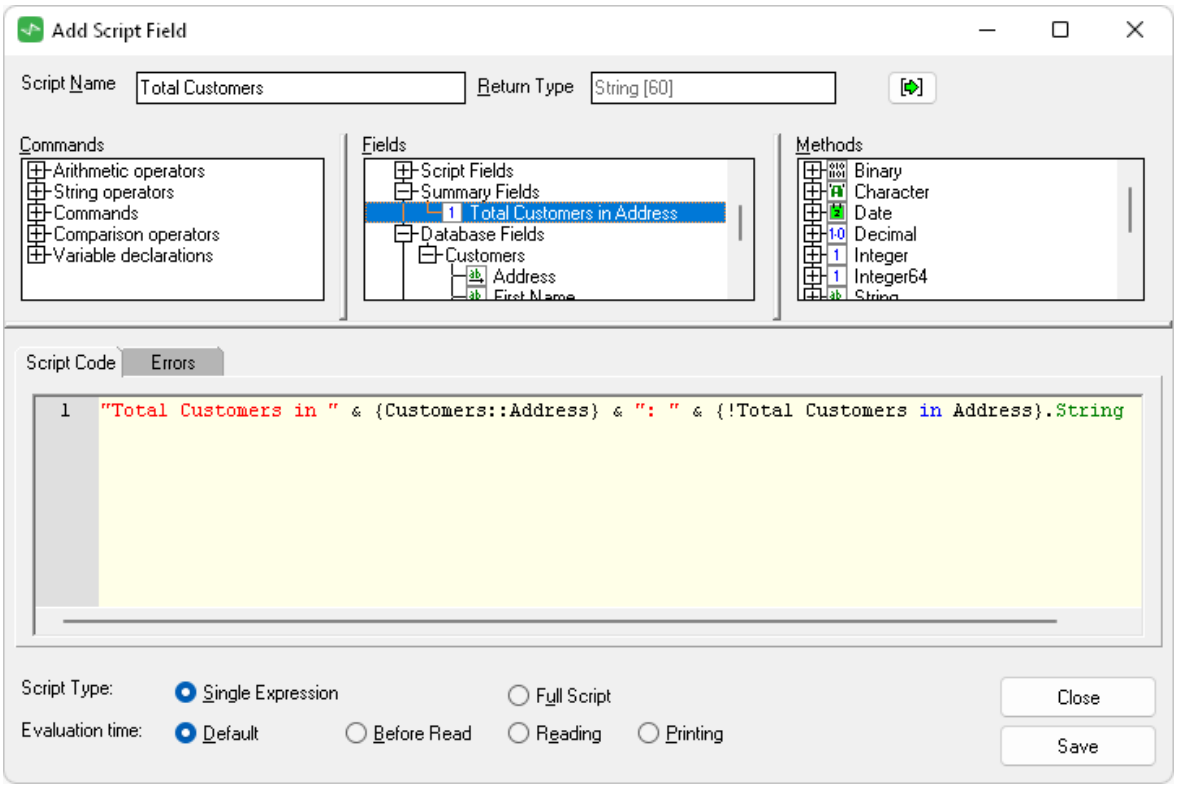
1. Select the **Summary** tab on the Catalog of Available Fields dialog, to display the **Summary** sheet.
2. Add a new summary by clicking the **New** button when the **Summary** sheet is displayed.

3. Fill out the form, as follows.



Note We could add this **Summary** field directly to the report. However, it can also be used within a **Single Expression Script Field** as a convenient way to include a label.

4. Create a new script, as follows.

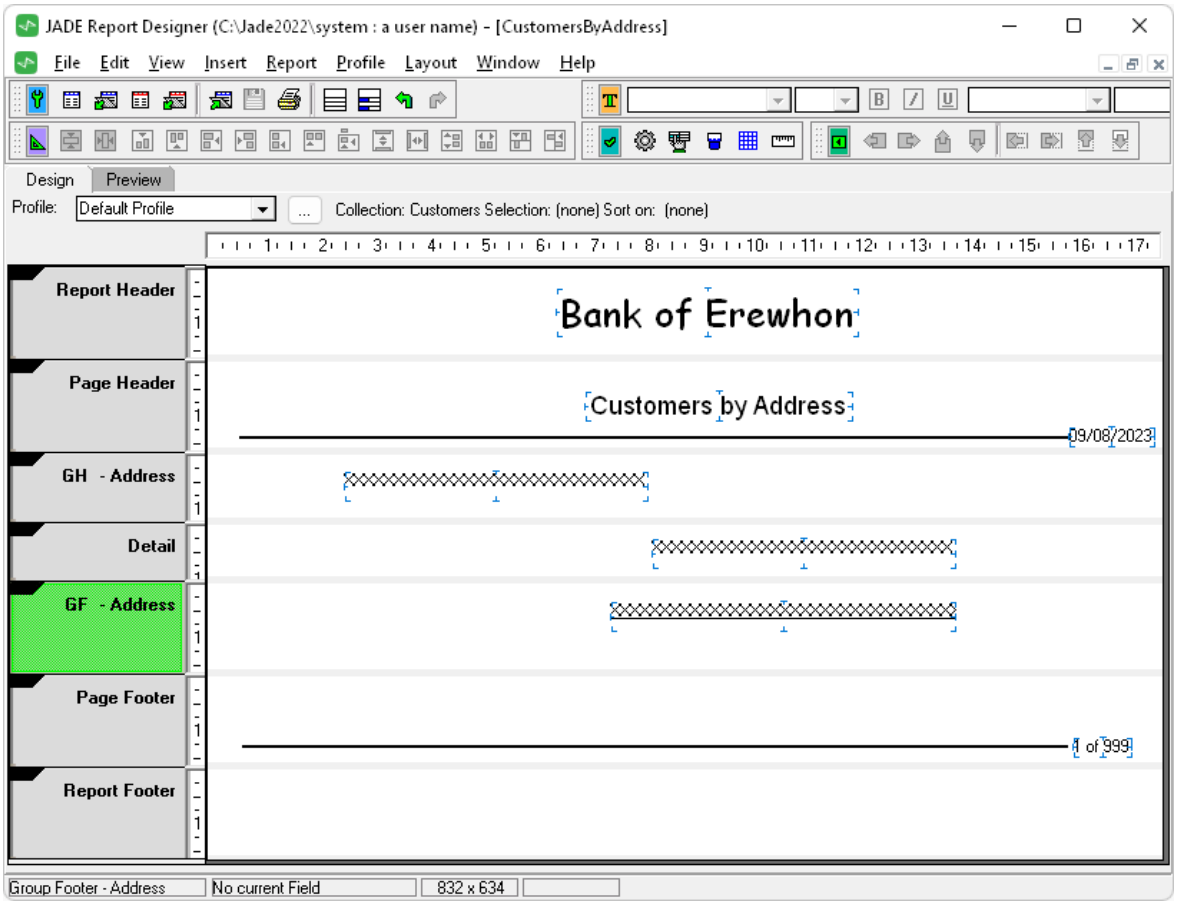


Note The value in the **Return Type** text box has been changed from **String[30]** to **String[60]**, as the results of this script may be longer than 30 characters.

To modify the return type, click the button at the right of the **Return Type** text box.

The **Summary** field has also been cast to a String type, by appending **.String** at the end.

5. Add this new script to the report in **GF - Address** and set it to underlined.



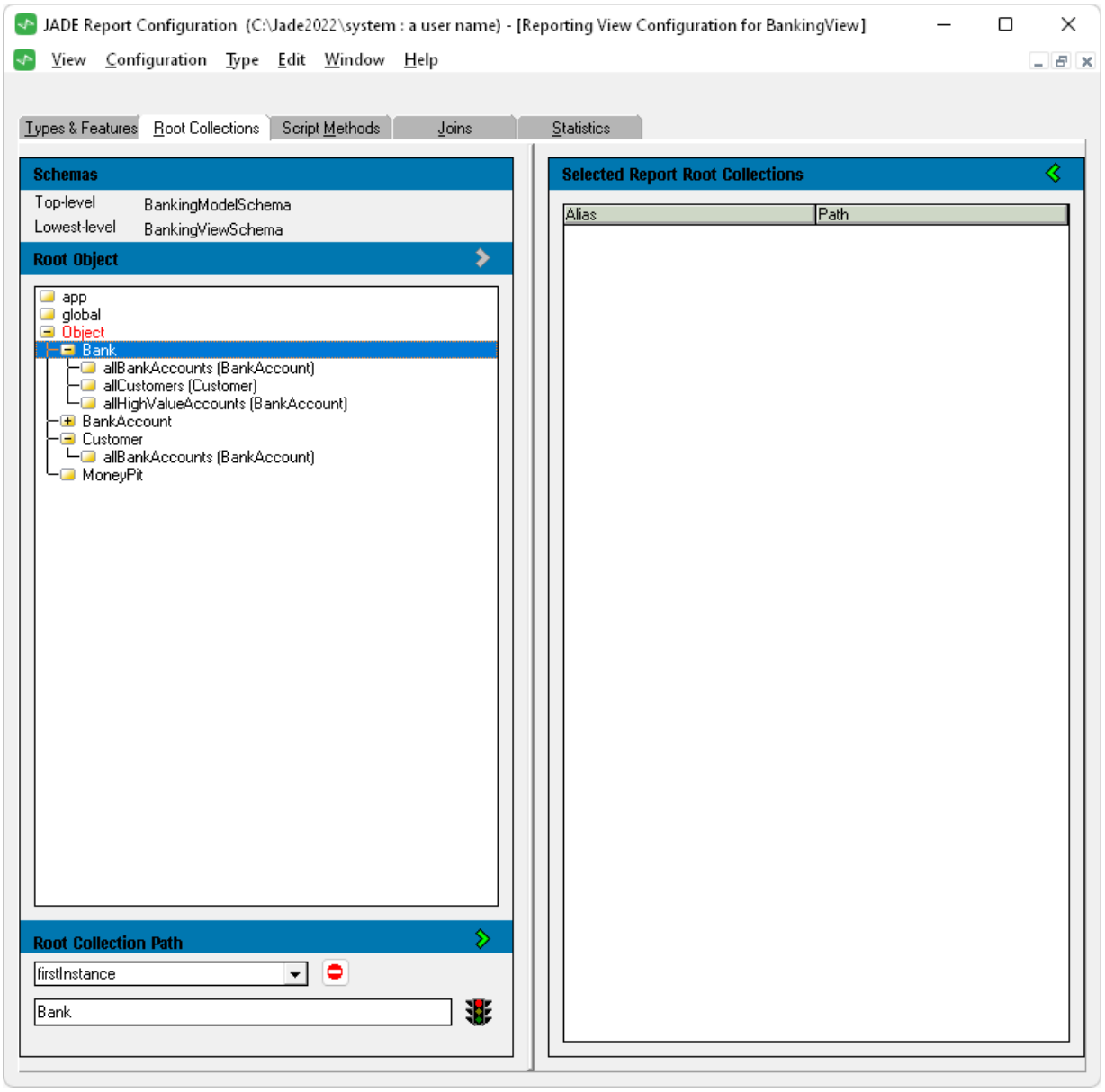
6. Select the **Preview** tab of the Jade Report Designer window, to view the report.

Root Collections

Reports are based on root collections, which provide the primary source of data for your reports. These root collections must be existing collections containing objects that have been selected in the **Types & Features** pane of the Report Configuration window.

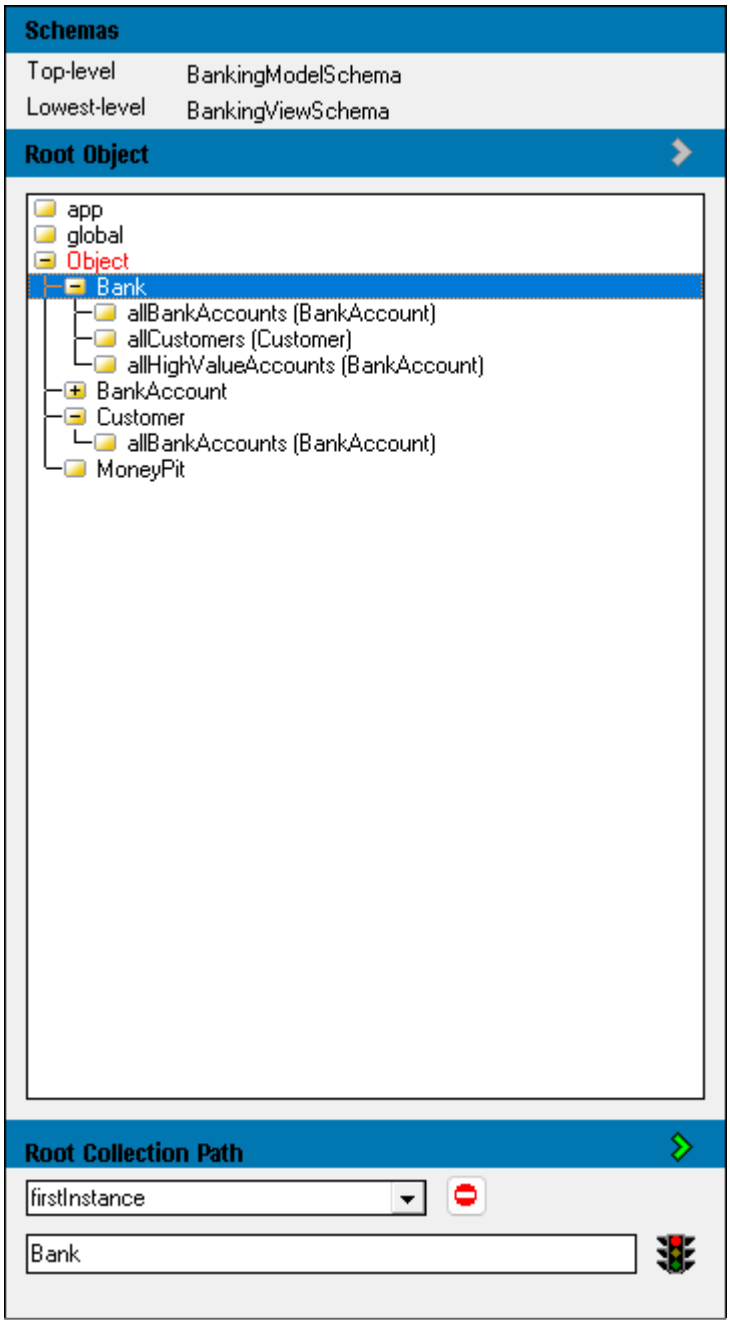
To set the root collections for your reporting view, use the **Root Collections** sheet of the Report Configuration window.

The **Root Collections** sheet contains a representation of all collections in the schemas between the top-level schema and lowest-level schema, inclusive.



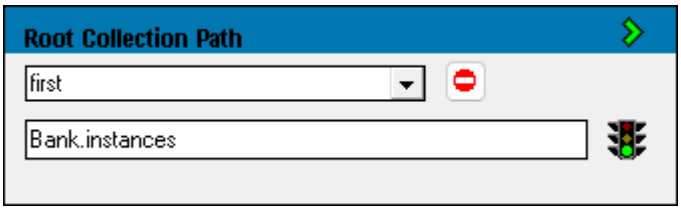
Note Classes themselves are represented by *class-name.instances* (for example, **Client.instances**, and therefore can be selected as a root collection. However, it is more usual to use a collection that has been defined explicitly in the schema.

When selecting a class or collection, the **Root Collection Path** pane will show which reference is currently highlighted, with a traffic light symbol showing whether it is valid as a root collection.

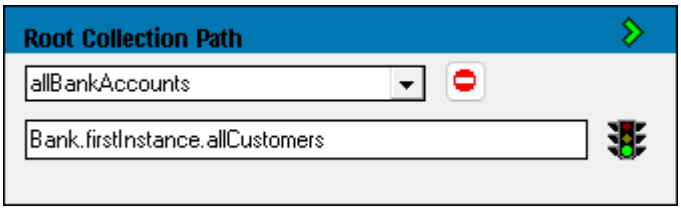


As the **firstInstance** property of the **Bank** class is a singular **Bank** object rather than a collection, this path is invalid, and the traffic light shows red.

As **Bank.instances** is a collection of all the **Bank** objects, the path is valid, and the traffic light shows green. However, this is likely not a useful collection to report on.



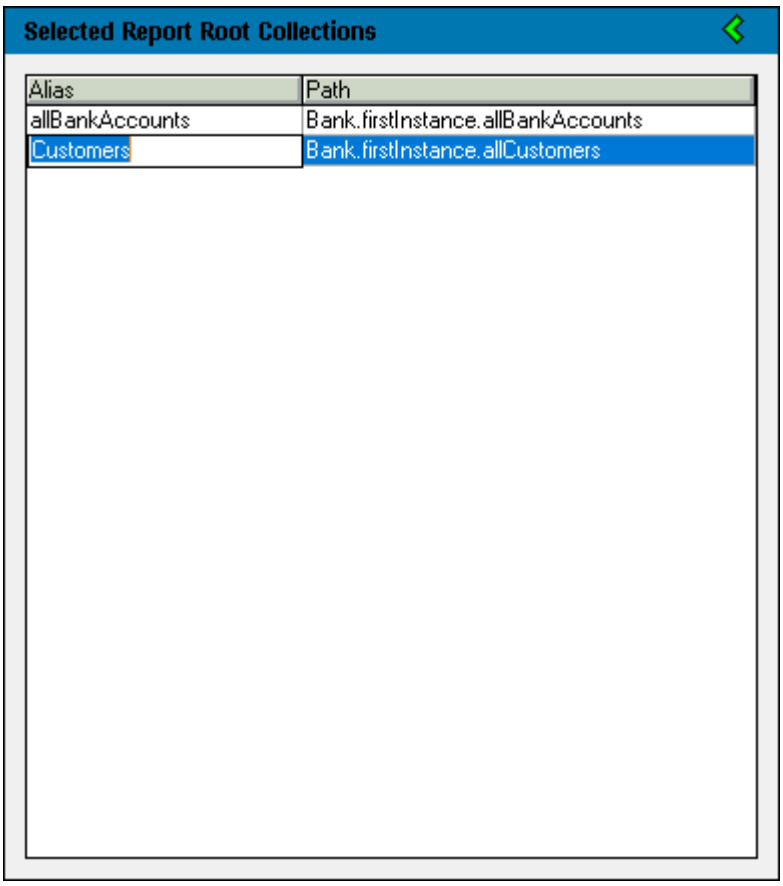
As **Bank.firstInstance.allCustomers** is a collection containing **Customer** objects, the path is valid, and the traffic light shows green.



When you have selected a valid collection to add as a Root Collection, click the green arrow button at the right of the header, to add it to the **Selected Report Root Collections** pane.

Alternatively, you can add it by double-clicking it in the **Root Object** pane.

As with types and features, you can set an alias if you do not want to use the variable name of the collection.



Setting Security Options

When designing reports in the Report Writer, it is often necessary to consider which users should have access to which reports and put security measures in place to enforce access rules. By default, all users have full access to all reports.

To allow for the definition of access rules, the **JadeReportWriterSecurity** class of **RootSchema** provides the following methods.

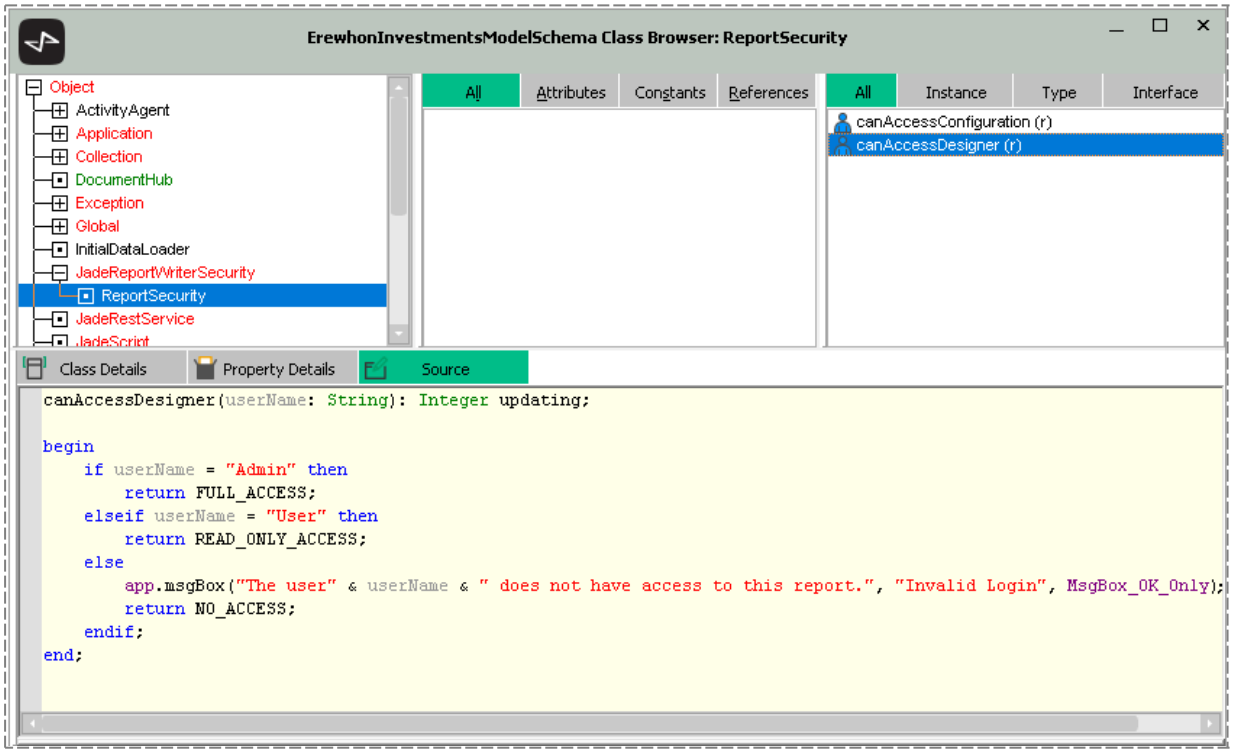
Method	Description
canAccessConfiguration	Returns the type of access that the specified user has to the configuration application.
canAccessDesigner	Returns the type of access that the specified user has to the designer application.
canAccessFolder	Returns the type of access that the specified user has to the specified folder in the configuration application.
canAccessReport	Returns the type of access that the specified user has to the specified report when reports are listed for selection.
canAccessView	Returns the type of access that the specified user has to the specified view when reports are listed for selection or extraction.
canAccessViewClass	Controls visibility of view classes in the designer application.
canAccessViewFeature	Controls visibility of view features in the designer application.
canDeleteReport	Controls which reports the user can delete.
canMaintainFolders	Returns the type of access that the specified user has to folders in the configuration application.
canMaintainSystemOptions	Returns the type of access that the specified user has to system options in the configuration application.
canMaintainViews	Returns the type of access that the specified user has to views in the configuration application.
folderDeleted	Called when a folder is deleted, with the specified folder path in the same format as that of the **newFolderAdded** method. Enables synchronization of user security details when a folder is deleted.
folderPathChanged	Called when a folder path is changed, with the specified folder paths in the same formats as that of the **newFolderAdded** method. Enables synchronization of user security details when a folder name is changed or the folder is moved to another parent folder.
isViewFeatureAccessSet	Specifies whether the user can access the view.
newFolderAdded	Enables the user who created a new report to access that report when security is set.
newReportAdded	Enables the user who created a new report to access that report when security is set.
newViewAdded	Enables the user who created a new view to access that view when security is set.
reportDeleted	Called when a report is deleted. Enables synchronization of user security details when a report is deleted.
reportNameChanged	Called when a report name is changed. Enables synchronization of user security details when a report name is changed.

Method	Description
viewDeleted	Called when a view is deleted. Enables synchronization of user security details when a view is deleted.
viewNameChanged	Called when a view name is changed.

These methods will return one of the following constants, as defined on the **JadeReportWriterSecurity** class.

Class Constant	Value	Description
FULL_ACCESS	2	Allows full access to the report for definition and use
NO_ACCESS	0	No access is allowed to the report
READ_ONLY_ACCESS	1	The report can be accessed and run but the definitions cannot be changed

To change the access rules from the default (that is, all users can access all reports), the methods can be reimplemented on a user-defined subclass of **JadeReportWriterSecurity**.



In this example, whenever the Report Writer Designer window is opened, it checks the user name that was provided. If it is **"Admin"**, the user has full access to reports. A user name of **"User"** returns read only access. An error message box is displayed and the Report Writer Designer application will not open for any other user name.

Note This is a simplified example, normally there would be a greater level of security defined in the method than simply checking for an expected user name.

Exercise 9 – Specifying Designer Security Settings

In this exercise, add a password to the Report Writer Designer application, to prevent unauthorized access.

1. Open the **BankingViewSchema** in the Class Browser.
2. Press F4, to display the Find Type dialog and then search for **JadeReportWriterSecurity**, opening it in the current browser.
3. Add a **JadeReportWriterSecurity** subclass called **BankingReportSecurity**.
4. Add a method called **canAccessDesigner** to the **BankingReportSecurity** class and click **Yes** on the message box warning that you are reimplementing a superclass method.
5. Code the method as follows.

```
canAccessDesigner(userName: String): Integer updating;

vars
    form : Logon;
begin
    create form transient;
    form.showModal();
    if form.txtPassword.text = "secret" then
        return FULL_ACCESS;
    else
        app.msgBox("Incorrect Password.", "Login Failed", MsgBox_OK_Only);
        return NO_ACCESS;
    endif;
epilog
    delete form;
end;
```

6. Modify the **JadeScript** class **startDesigner** method in **BankingViewSchema** as follows.

```
startDesigner();

vars
    rw : JadeReportWriterManager;
begin
    create rw transient;
    //rw.startReportWriterDesigner("User", null);
    rw.startReportWriterDesigner("User", BankingReportSecurity);
epilog
    delete rw;
end;
```

7. Run the **JadeScript** class method. If you enter **secret** as the password, the Report Writer Designer application starts as normal; otherwise you are denied entry.

Exercise 10 – Specifying Configuration Security Settings

In this exercise, add a user name requirement to the Report Writer Configuration application.

1. Add a method to **BankingReportSecurity** called **canAccessConfiguration** and then click **Yes** in the message box warning that you are reimplementing a superclass method.

2. Code the method as follows.

```
canAccessConfiguration(userName: String): Integer updating;

begin
    if userName = "Admin" then
        return FULL_ACCESS;
    elseif userName = "User" then
        return READ_ONLY_ACCESS;
    else
        return NO_ACCESS;
    endif;
end;
```

3. Modify the **JadeScript** class **startDesigner** method in **BankingViewSchema**, as follows.

```
startConfiguration();

vars
    rw : JadeReportWriterManager;
begin
    create rw transient;
    //rw.startReportWriterConfiguration("User", null);
    rw.startReportWriterConfiguration("Admin", BankingReportSecurity);
epilog
    delete rw;
end;
```

4. Run the **JadeScript** class method, which should allow you full access.

Try changing the user name from **Admin** to **User** for read-only access, or any other user name for denial of access.

REST Services

The Jade Platform provides the ability to provide RESTful web services from a Jade database, as well as a limited capacity for acting as a REST client. The **JadeRestService** class handles the processing of REST requests, and the **JadeHTTPConnection** class can send **POST** and **GET** requests to a REST web server.

Representational State Transfer (REST) is a type of web service that receives and responds to HTTP requests in JSON (JavaScript Object Notation) or XML (eXtensible Markup Language). REST web services are a lightweight alternative to other web services such as SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language).

One aspect of REST web services that causes them to be more lightweight than SOAP-based web services is their property of statelessness. Statelessness means that a web server handling REST requests does not store any information about the client on the server. The client is responsible for maintaining any required state information and communicating its state to the server as part of any request that requires it. This means that, from the server's perspective, all requests are completely isolated from each other and as such, the cost of each client is lessened, improving scalability potential.

REST requests embed all required query parameters within the URL, meaning that the URL will contain not only the location of the web provider but also the type of request and any parameters needed for that request. This style of URL is known as a REST-style URL. For example, consider the following URL.

```
http://localhost/RestfulJade/jadehttp.dll/Customer/3.xml?RestService
```

This URL contains both where to find the web service (that is, the location of the resource) expressed in the **http://localhost/RestfulJade/jadehttp.dll/** part and also the request itself, expressed in the **Customer/3.xml?RestService** part. This request part of the REST-style URL expresses that the method name is **Customer**, the parameter to pass is **3**, the returned message should be in **XML** format, and the name of the Jade web services application is **RestService**.

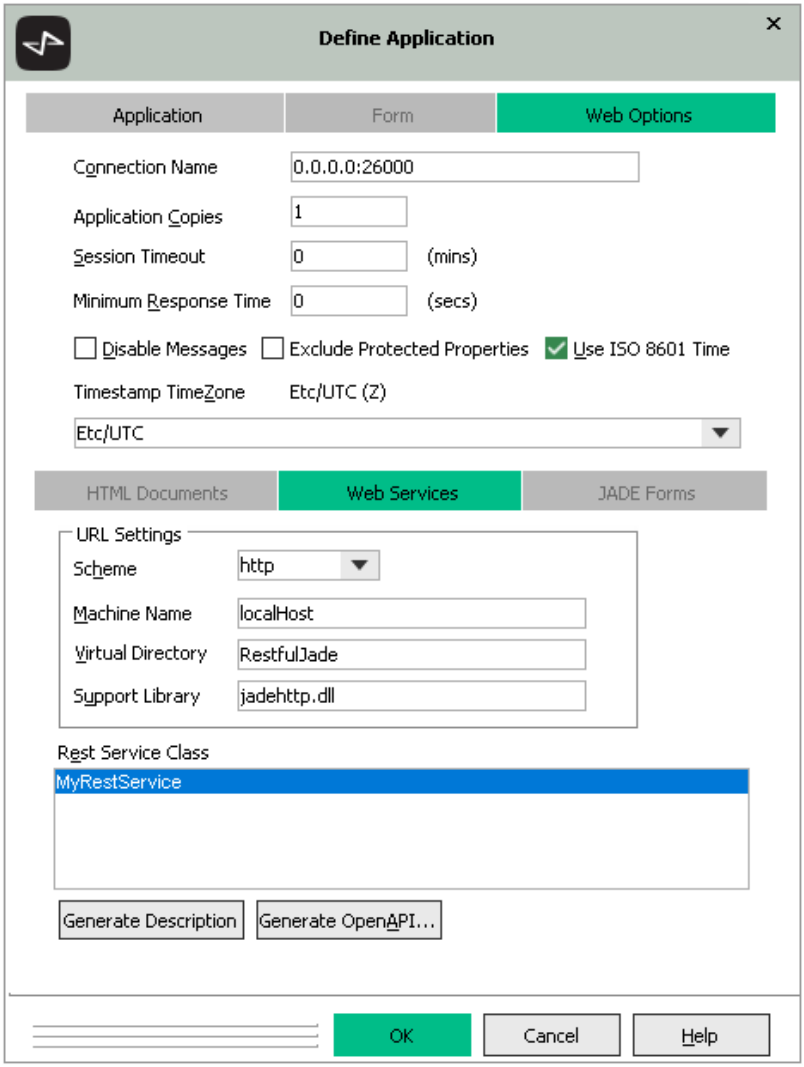
Internet Information Services (IIS)

Internet Information Services (IIS) is Microsoft's web server software for hosting web services. IIS is most commonly run on a dedicated machine running Windows Server but can also be run on a home computer running Windows 7 or later for smaller-scale operations.

When a machine running IIS receives a web request, it checks its list of applications for one that matches the request, and delegates the request to the web service provider specified in the application. For Jade databases acting as a web service provider, this is done by specifying the **jadehttp.dll** file, found within the bin directory of the Jade Platform installation folder. This DLL file then sends the request through the port specified in the **jadehttp.ini** file. Whichever Jade application is running and listening for requests on the matching port handles the request and returns its reply through IIS and back to the client.

REST Services Applications and JadeRestService Class

The Jade Platform's implementation of REST web services relies on having an application of application type **Rest Services** (or **Rest Services, Non-Gui**) with a reference to a subclass of the **JadeRestService** class specified in the **Rest Service Class** list box on the **Web Options** sheet of the Define Application dialog.



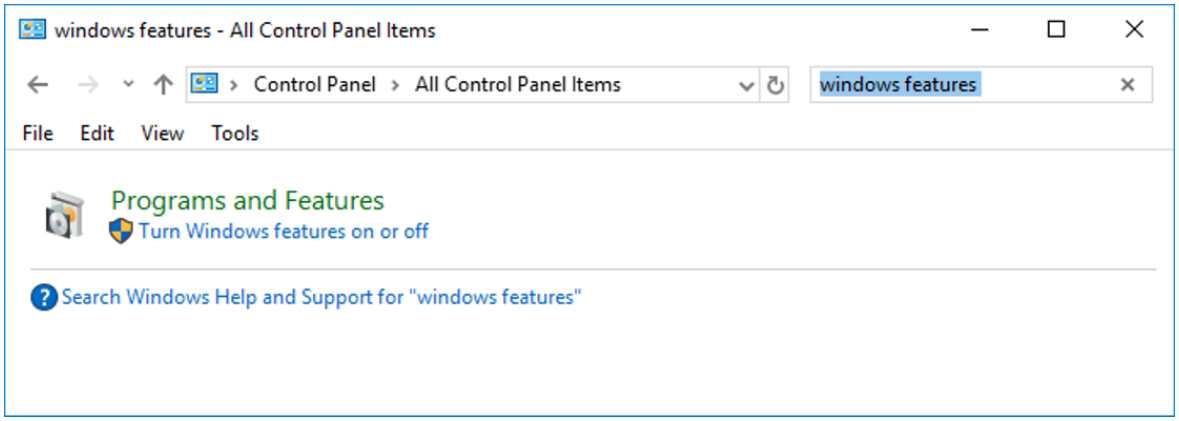
This application is the entry point into the Jade database and calls the **processRequest** method of the specified **JadeRestService** class subclass whenever a request is received on the port specified in the **Connection Name** text box.

The **processRequest** method of the **JadeRestService** class is responsible for determining which method should be called to fulfil the request, calling that method, and then calling the **reply** method of the **JadeRestService** class to send the returned value of the called method back to the client. To determine which method to call, it concatenates the type of request (that is, **GET**, **POST**, **PUT**, or **DELETE**) with the name provided in the URL. For example, a **GET** request with **Customer** as the provided name calls the **getCustomer** method.

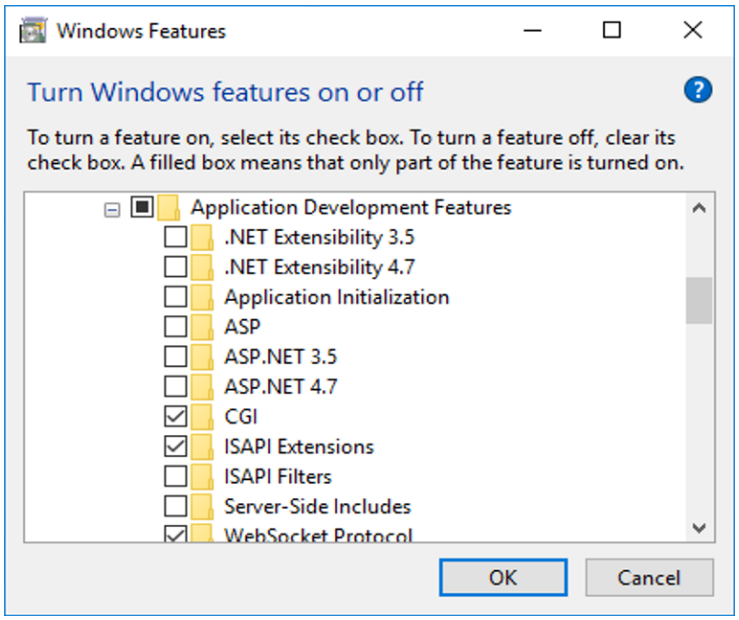
Exercise 1 – Setting Up IIS

In this exercise, set up IIS for use with REST web services.

1. Open the Windows Control Panel and search for **windows features**.

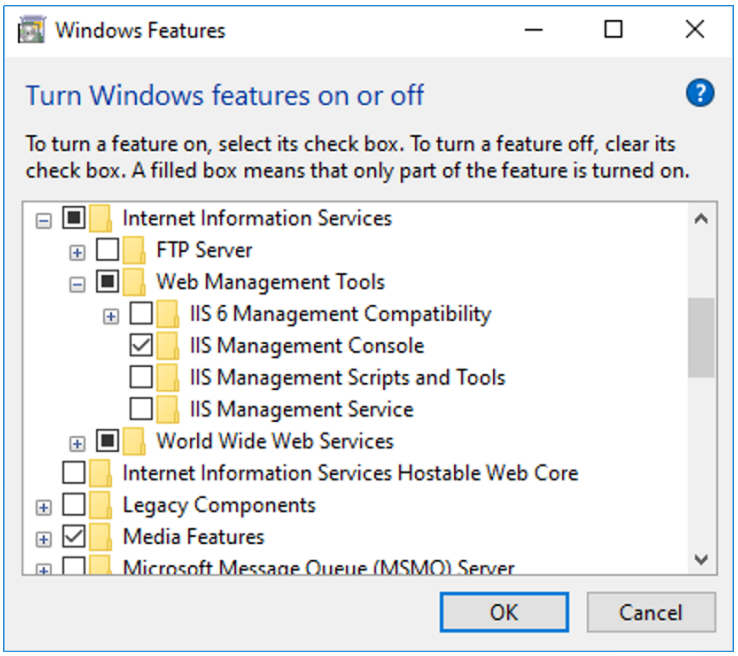


2. Select **Turn Windows features on or off**. The Windows Features window is then displayed.

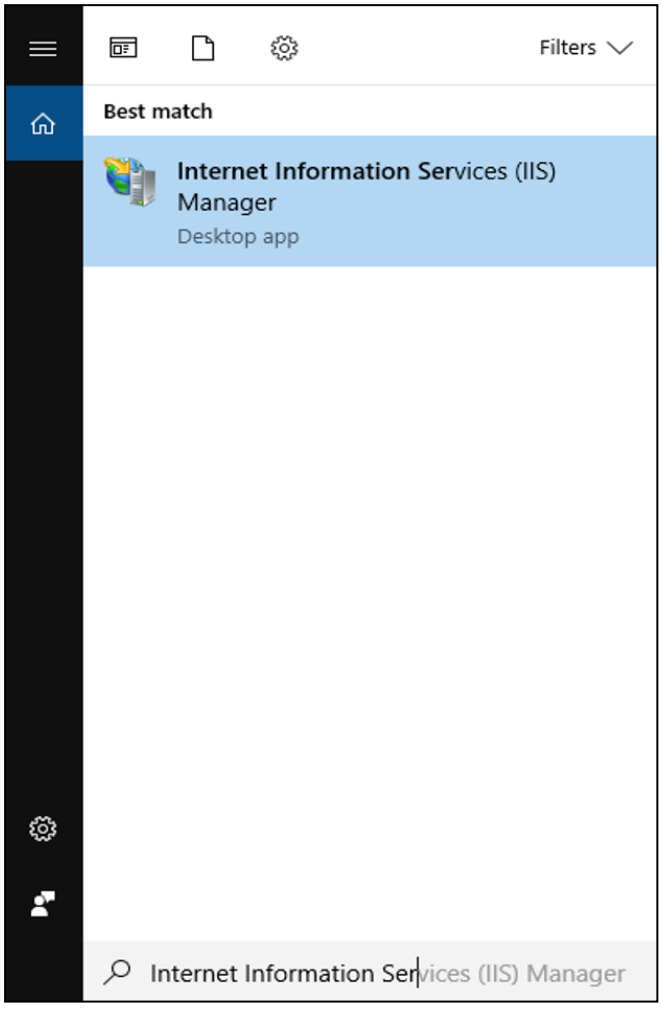


3. From this window, expand **Internet Information Services**, then **World Wide Web Services**, then **Application Development Features**, and check the following check boxes.
   - CGI
   - ISAPI Extensions

4. In addition, from this window, expand **Web Management Tools**, which is under **Internet Information Services**, and then check the **IIS Management Console** check box.



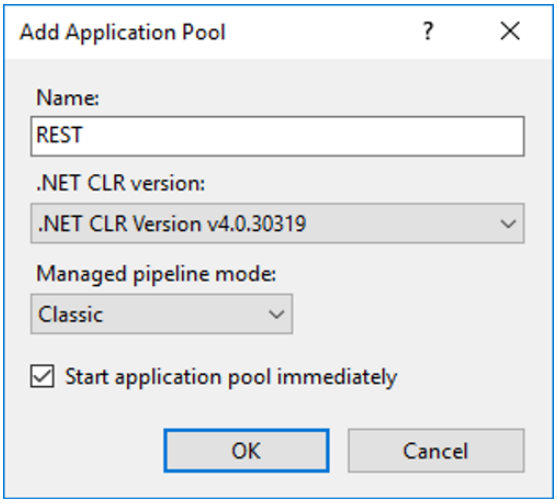
5. Open the Internet Information Services (IIS) Manager.



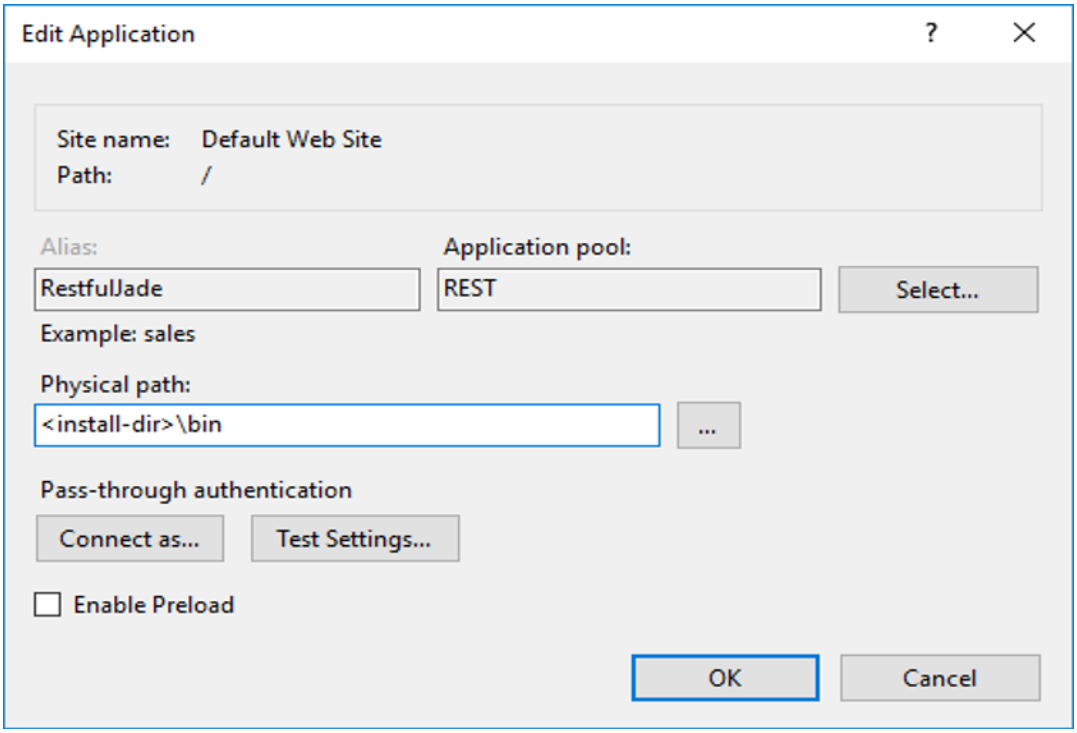
Tip The easiest way to access it is to search for it.

6. From the **Connections** panel on the left of the IIS Manager, select **Application Pools**.
7. From the **Actions** panel on the right, select **Add Application Pool**.

8. Call the application pool **REST** and set **Managed pipeline mode:** to **Classic**.

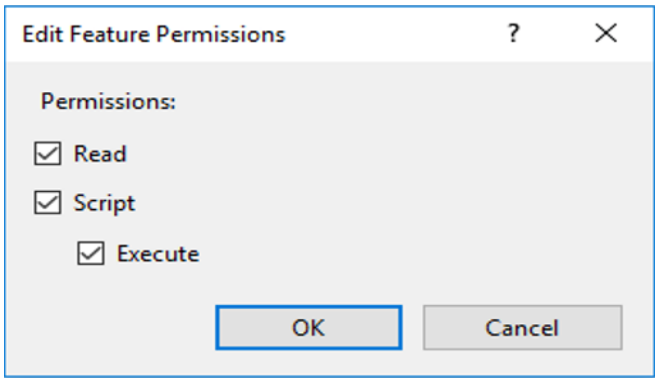


9. From the **Connections** panel, right-click **Default Web Site** and then select **Add Application**.
10. Call the application **RestfulJade** and set the physical path to the **bin** directory of your Jade Platform installation.

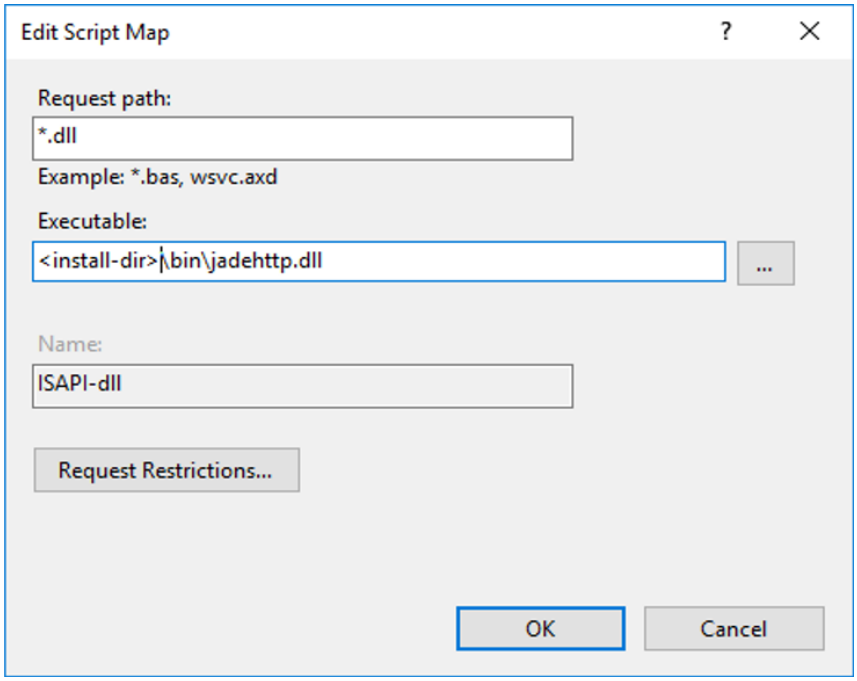


11. Select the newly created **RestfulJade** application from the **Connections** panel and then open **Handler Mappings** from the center panel.

12. Select **CGI-exe** from the middle panel and then select **Edit Feature Permissions** from the **Actions** panel in the right. Check all check boxes.



Select **ISAPI-dll** from the middle panel and then select **Edit** from the **Actions** panel on the right. Set the executable to the **jadehttp.dll** file in the bin directory of your Jade Platform installation.

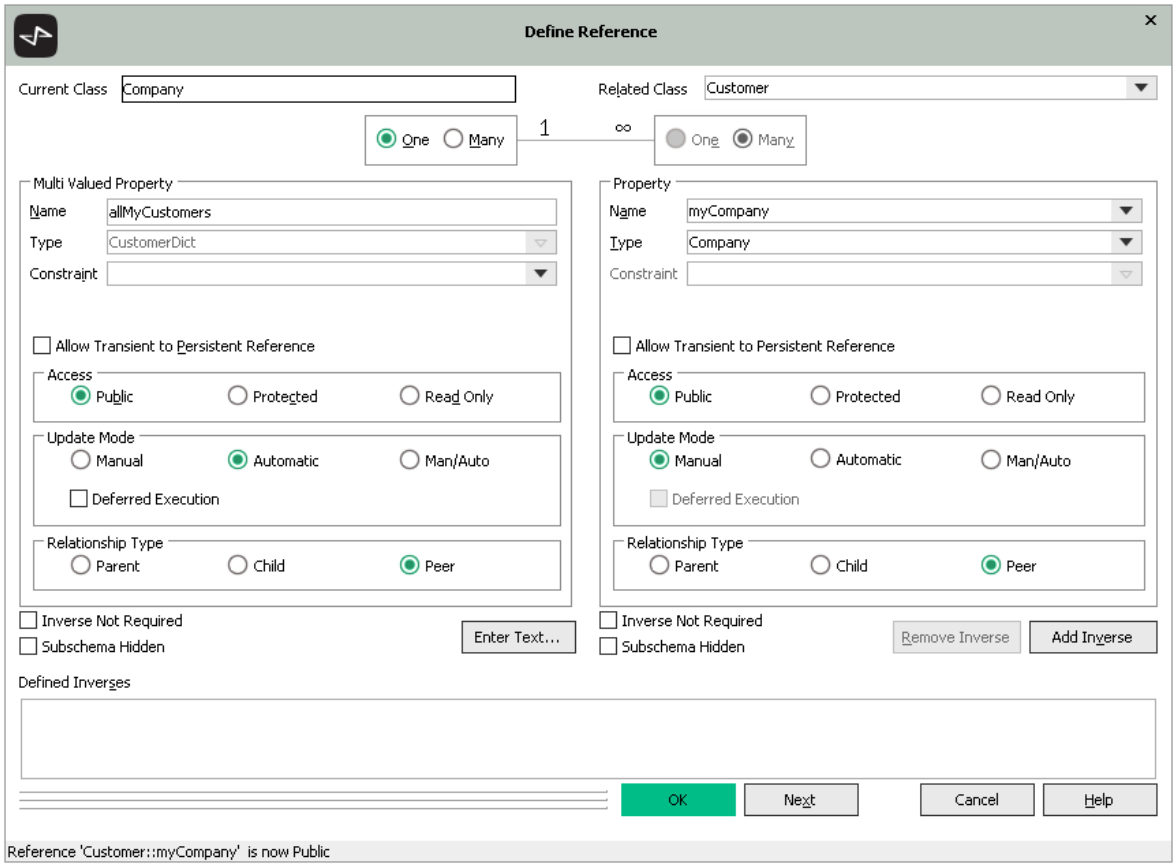


Exercise 2 – Creating a REST Provider

In this exercise, create a Jade application that provides REST web services.

1. Open Jade if it is not already open, and then create a schema called **RestfulSchema**.
2. In **RestfulSchema**, create a class called **Company** and another class called **Customer**.
3. To the **Customer** class, add a public attribute called **id** of type **Integer** and a public attribute called **name** of type **String**.
4. Search for **MemberKeyDictionary** (F4) and then add a subclass to it called **CustomerDict** that has membership **Customer** and key **id**.

5. To the **Company** class, add an inverse reference with **Customer**, as follows.



- Inverse property called **myCompany**
- Both access types **Public**
- Automatic **allMyCustomers** to manual **myCompany**

6. Create a **JadeScript** class **init** method, coded as follows, and then run it.

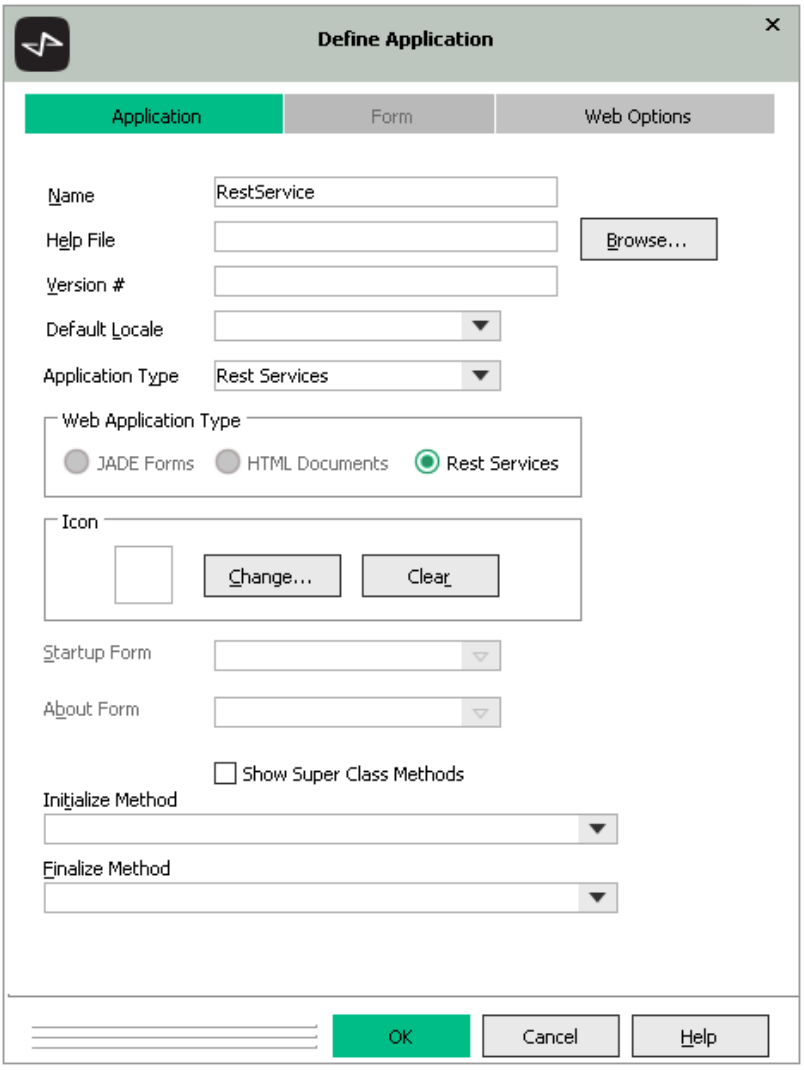
```
init();

vars
    company : Company;
    cust    : Customer;
begin
    beginTransaction;
    create company;

    create cust;
    cust.id := 1;
    cust.name := "Albert Brian";
    cust.myCompany := company;

    create cust;
    cust.id := 2;
    cust.name := "Carla Donaldson";
    cust.myCompany := company;

    create cust;
    cust.id := 3;
    cust.name := "Edward Fields";
    cust.myCompany := company;
    commitTransaction;
end;
```

7. In the Class Browser, search for **JadeRestService** (F4) and then add to it a subclass called **MyRestService**.

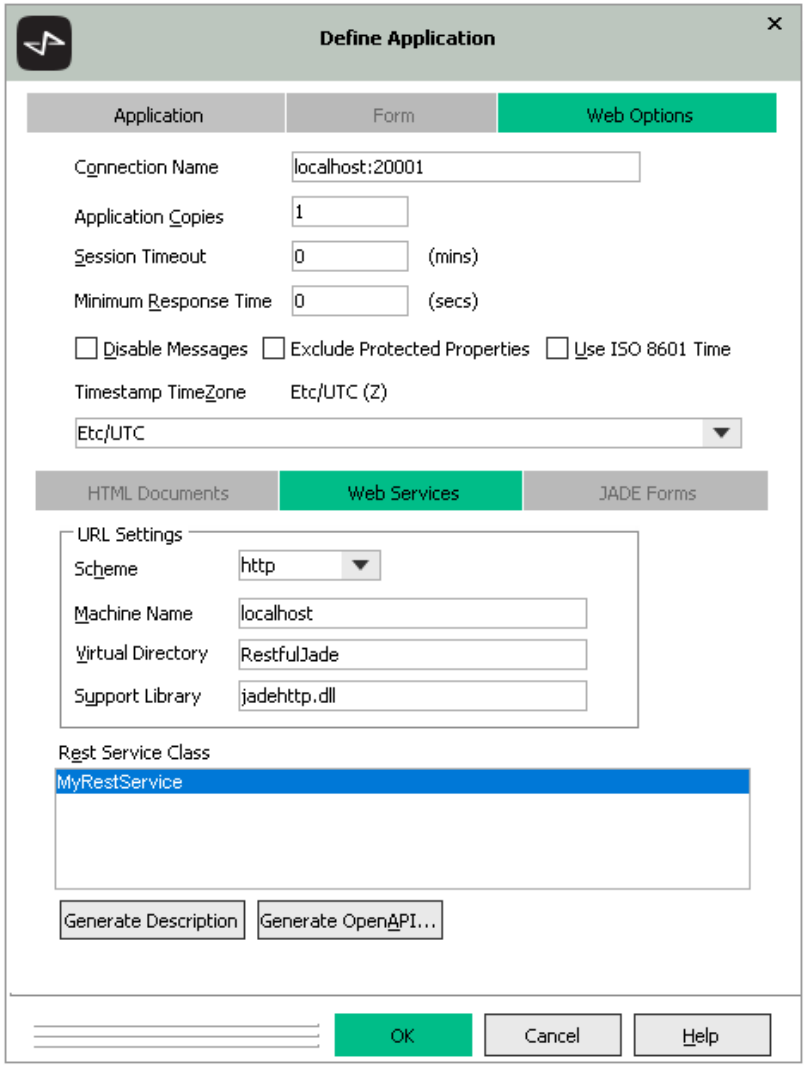
8. To the **MyRestService** class, add a method called **getCustomer**, coded as follows.

```
getCustomer(id : Integer) : Customer;

vars
    theCompany      : Company;
    theCustomer     : Customer;
    proxyCustomer   : Customer;

begin
    // Company.firstInstance is not scalable, but will serve for this exercise
    theCompany      := Company.firstInstance;
    theCustomer     := theCompany.allMyCustomers[id];

    // We need to send a proxy to avoid object access conflicts.
    proxyCustomer   := theCustomer.copySelf(true);
    return proxyCustomer;
end;
```

9. Open the Application Browser (Ctrl+L) and add an application called **RestService**, with the application type **Rest Services**.

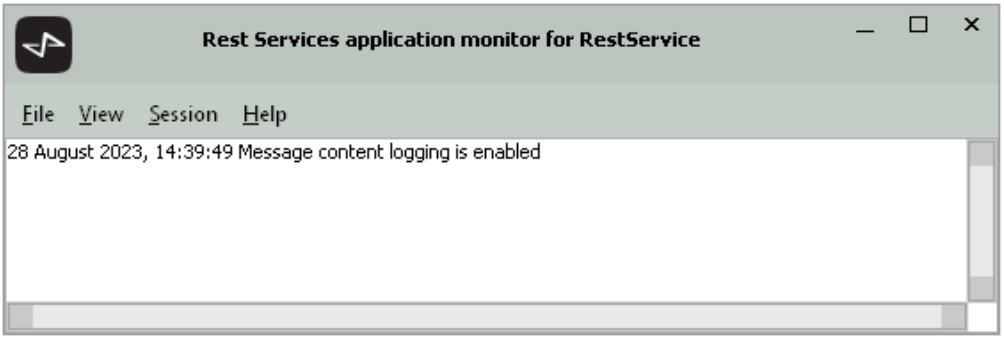


Select the **Web Options** sheet and specify the following.



- For the connection name, enter **localhost:20001**
- For the machine name, enter **localhost**
- For the virtual directory, enter the name of the IIS application created in Exercise 1 of this module; that is, **RestfulJade**
- All other settings can be left as the default values

10. Start the **RestService** application.



11. Open a web browser and then navigate to the following URL.

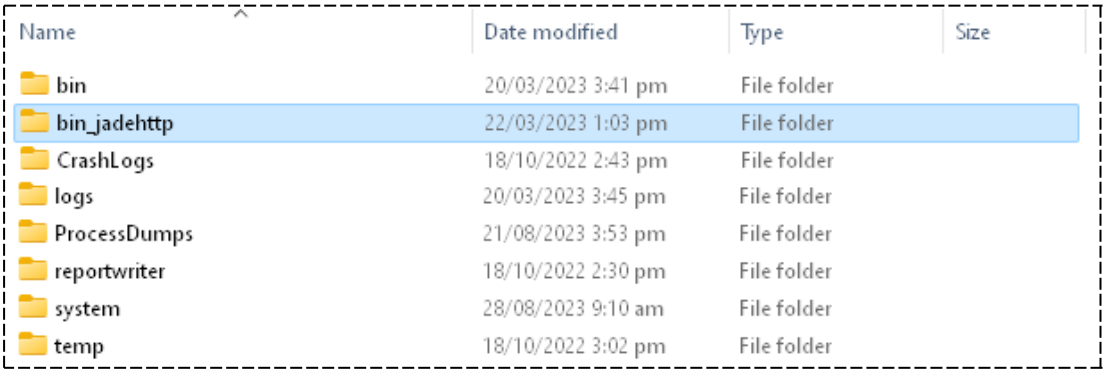
    http://localhost/RestfulJade/jadehttp.dll/Customer/3/?RestService

    Note that the URL has the following parts.

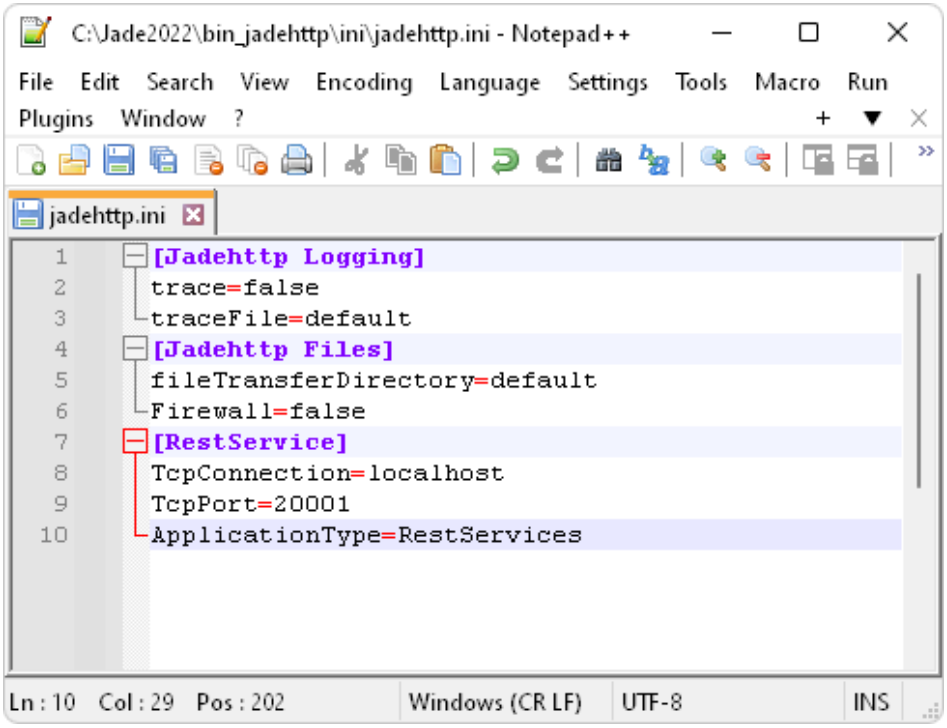
    - **http://localhost** means use the current computer as the web server
    - **/RestfulJade** means use the handler mappings specified in the **RestfulJade** IIS application; that is, the **jadehttp.dll** specified in the **ISAPI-dll** mapping
    - **/Customer/3** means call the **getCustomer** method with the parameter **3**
    - **/?RestService** means use the **RestService** Jade application

12. The following error is displayed.

Don't panic. As the **jadehttp.ini** file is not needed except when using Jade as an HTTP server, it is generated only when attempting to use Jade as an HTTP server for the first time. Navigating in File Explorer to your Jade Platform installation folder, you will see the following folder has been created.



13. Open this folder, then edit the **ini** folder within it, adding the following [RestService] section to the **jadehttp.ini** file.



Note The **TcpPort** parameter value can be any unused port, but it must match the port number specified in the **Connection Name** text box on the **Web Options** sheet of the Define Application dialog for the Jade Rest Services application.

14. Reopen the http://localhost/RestfulJade/jadehttp.dll/Customer/3/?RestService URL. It now shows a JSON string representation of the **Customer** object, as follows.

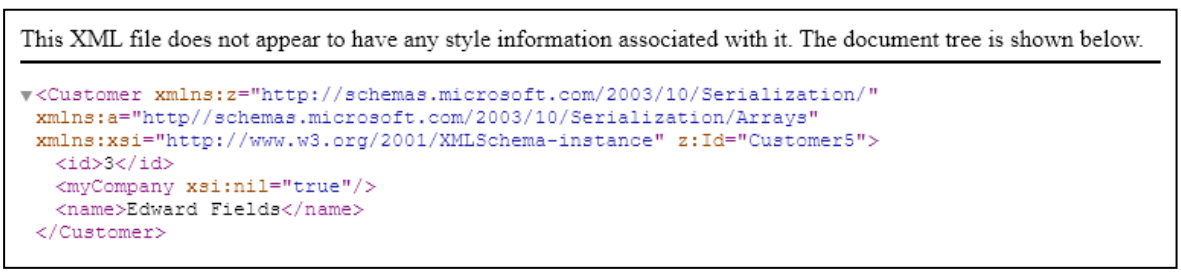
```
{"id":3,"myCompany":null,"name":"Edward Fields"}
```

Note The default representation of objects is JSON. This could also be set explicitly; for example, http://localhost/RestfulJade/jadehttp.dll/Customer/3.json?RestService.

15. Modify the URL in the browser to the following, and then refresh the page.

```
http://localhost/RestfulJade/jadehttp.dll/Customer/3.xml?RestService
```

This will show the same customer, but this time in XML format rather than in JSON.



Note The **myCompany** reference is null, as the **Customer** object passed to the browser was a clone, and references of clones are not automatically cloned (although it can be set explicitly, if required).

Making REST Requests from Jade

In addition to functioning as a provider of REST web services, Jade can also act as a client. The Jade Platform provides the **JadeHTTPConnection** class, which can send HTTP requests including those of the form required for REST web service providers.

Although the **JadeHTTPConnection** class provides many methods, the one we will use is the **getHttpPage** method, which returns a JSON or XML string containing the REST response and has the following parameters.

Parameter	Description
pVerb	The HTTP verb as a String; for example, **GET** or **POST**.
pServerName	The URL of the web server. It should contain the full REST-style URL.
pUrlAddress	Not needed for REST requests, so it can be null.
pUrlAddress	Not needed for REST requests, so it can be null.
pContentType	Not needed for REST requests, so it can be null.

For example, consider the following **JadeScript** class method, which returns the same result as navigating to http://localhost/RestfulJade/jadehttp.dll/Customer/3.xml?RestService in a browser.

```
restfulExample();

vars
    httpConnection  : JadeHTTPConnection;
    restStyleURL    : String;
begin
    create httpConnection transient;
    restStyleURL := "http//localhost/RestfulJade/jadehttp.dll/Customer/3.xml?RestService";
    write httpConnection.getHttpPage("GET", restStyleURL, null, null, null);

epilog
    delete httpConnection;
end;
```

HTTP Request Types in Jade REST Requests

So far, we have only performed **GET** requests through REST web services. Jade REST services are also compatible with the **GET**, **POST**, **PUT**, and **DELETE HTTP** request verbs, as follows.

Verb	Description
GET	Retrieves a representation of a resource from a web server. This representation can be in JSON or XML format. **GET** requests should never modify any of the resources of the web server.
POST	Creates a new resource on the web server. Note that **POST** requests are not idempotent by default; that is, multiple identical **POST** requests create multiple identical resources.
PUT	Updates an existing resource on the web server. Unlike **POST** requests, **PUT** requests are idempotent. As such, multiple PUT identical requests cause only a single change to the web server's resources.
DELETE	Deletes a resource on the web server. Although usually idempotent, repeated identical **DELETE** requests may return error 404, as the target resource no longer exists after the first deletion.

Note An *idempotent* request is one where making the request multiple times results in the same behavior as making the request a single time only.

Exercise 3 – Creating a REST Client

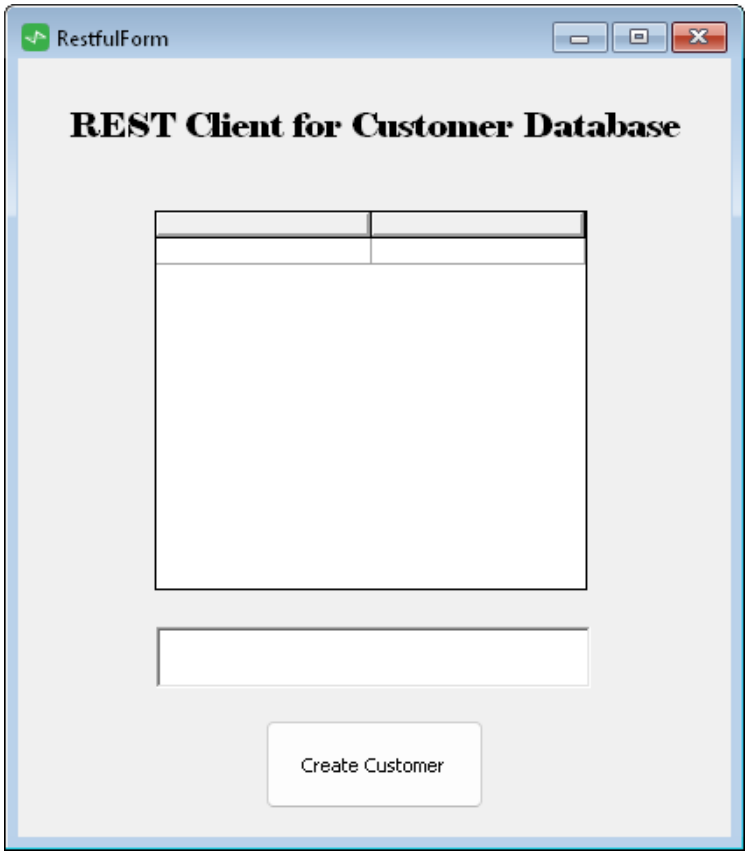
In this exercise, create a Jade application that acts a client to the web server created in the previous exercise and uses REST requests to read data from the **RestfulSchema** schema.

1. Create a schema named **RestfulClient**.

   Note As the REST client will not directly reference **RestfulSchema** but uses REST requests to obtain relevant data, it does not need to be in the same database (or even on the same physical machine). However, since we are assuming you do not have multiple Jade databases available, a Jade database can act as a client and a server at the same time for the purposes of this exercise.

2. From the **RestfulClient** schema, open the Jade Painter.

   Create a form called **RestfulForm** and add the following controls.



   - A label called **lblTitle** with:
     - Caption: **REST Client for Customer Database**
     - Alignment: **7 - Center - MIDDLE**
     - FontName: **Elephant**
     - FontSize: **14**

- A table called **tblCustomers** with:
  - Columns: **2**
  - fixedColumns: **0**
- A text box called **tbCustomerName**
- A button called **btnCreateCustomer** with:
  - Caption: **Create Customer**

3. The first thing we will do is display the customers from the **RestfulClient** schema in the table. Add a class called **RestQuerier** to the **RestfulClient** schema.
4. Add the following class constants to the **RestQuerier** class.
   - **RestURL**, which is a **String** with definition **"http://localhost/RestfulJade/jadehttp.dll/"**
   - **RestApp**, a **String** with definition **"?RestService"**

Note As this is a **String** definition, the double quotes must be included.

5. Add a method called **getCustomers**, coded as follows, to the **RestQuerier** class.

```
getCustomers(xmlDoc : JadeXMLDocument io) typeMethod;

constants
    RestMethodName  = "AllCustomers";
    ResultType      = ".XML";
vars
    connection  : JadeHTTPConnection;
    url : String;
begin
    create connection transient;
    url := RestURL & RestMethodName & ResultType & RestApp;
    xmlDoc.parseString(connection.getHttpPage("GET", url, null, null, null));
epilog
    delete connection;
end;
```

Note As the **RestQuerier** class has only methods and no properties, all methods are *type* methods, which means they can be called without creating a **RestQuerier** object.

The **xmlDoc** parameter is passed as an **io** parameter, so it is modified and returned to the caller (who is responsible for deleting the transient **JadeXMLDocument** when finished with it).

For more details about XML documents in Jade, see the XML additional module of the Jade Platform Developer's course.

6. On the **RestfulForm** class, modify the Form Events **load** method, as follows.

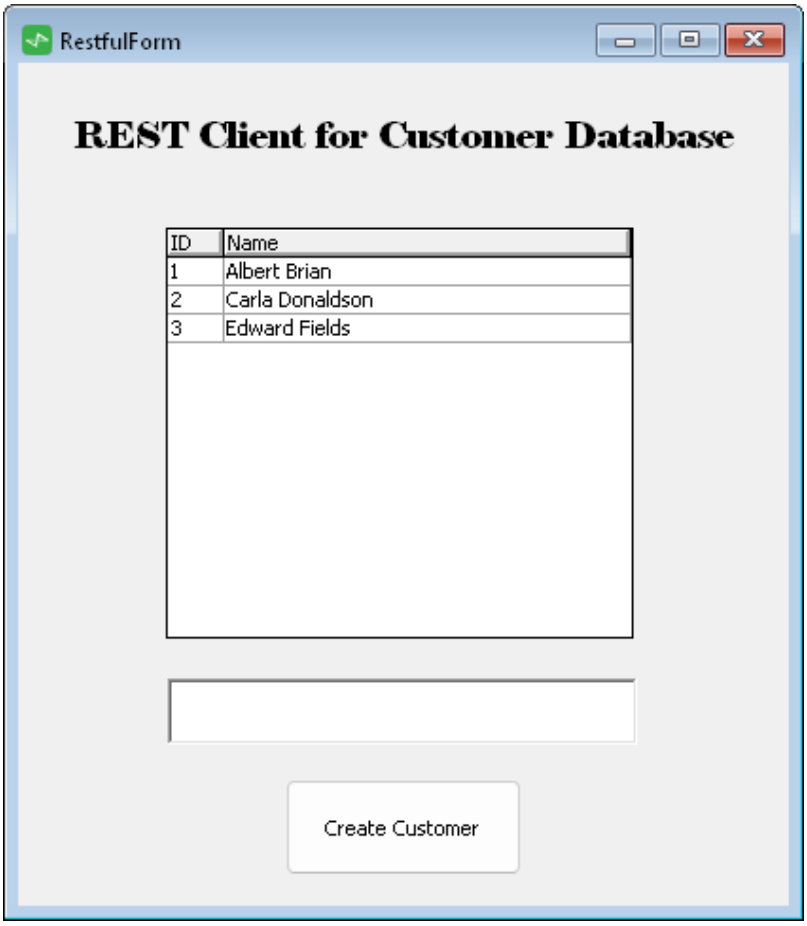
```
load() updating;

vars
    xmlDoc      : JadeXMLDocument;
    elements    : JadeXMLElementArray;
    element     : JadeXMLElement;
    id          : String;
    name        : String;
begin
    // Sets up the table with correct headers
    self.tblCustomers.rows := 1;
    self.tblCustomers.accessCell(1, 1).text := "ID";
    self.tblCustomers.accessCell(1, 2).text := "Name";
    self.tblCustomers.accessColumn(1).widthPercent := 15;

    create xmlDoc transient;
    // This ClassName@methodName syntax invokes a type method.
    // Note that there is no RestQuerier object instantiated.
    RestQuerier@getCustomers(xmlDoc);

    create elements transient;
    xmlDoc.findElementsByTagName("Customer", elements);

    foreach element in elements do
        id   := element.getElementByTagName("id").textData;
        name := element.getElementByTagName("name").textData;
        tblCustomers.addItem(id & Tab & name );
    endforeach;

epilog
    delete xmlDoc;
    delete elements;
end;
```

7. In the **RestfulSchema** schema, add a new method to **MyRestService**, coded as follows.

```
getAllCustomers() : CustomerDict;

vars
    proxy : CustomerDict;
    cust  : Customer;

begin
    create proxy transient;
    foreach cust in Company.firstInstance.allMyCustomers do
        proxy.add(cust.copySelf(true));
    endforeach;
    return proxy;
end;
```

8. Run the default **RestfulClient** application.

You should see that the table has been populated with the customers from the **RestfulSchema**.



Tip Ensure that the **RestfulSchema** application is running whenever using the **RestfulClient** application.

Exercise 4 – Making a POST Request over REST

In this exercise, enable the form created in the previous exercise to send **POST** requests to the REST web service so that you can add customers to the company defined in **RestfulSchema**.

In the **RestQuerier** class of the **RestfulClient** schema:

1. Add a *type* method called **postCustomer**, coded as follows.

```
postCustomer(name : String) typeMethod;

constants
    RestMethodName = "Customer";
vars
    connection : JadeHTTPConnection;
    url : String;
begin
    create connection transient;
    url := RestURL & RestMethodName & "/" & name & RestApp;
    connection.getHttpPage("POST", url, null, null, null);
epilog
    delete connection;
end;
```

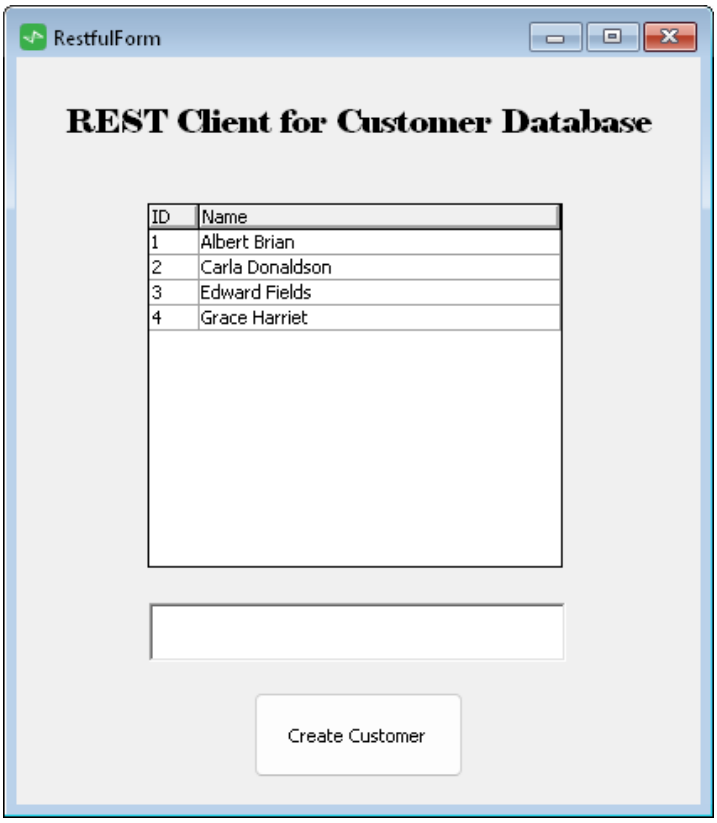
2. Modify the **click** method of the **btnCreateCustomer** control in the **RestfulForm** class, as follows.

```
btnCreateCustomer_click(btn: Button input) updating;

vars

begin
    RestQuerier@postCustomer(tbCustomerName.text);
    tbCustomerName.text := "";
    self.load();
end;
```

3. In the **RestfulSchema** schema, add a new method to **MyRestService**, coded as follows.

```
postCustomer(name : String);

vars
    newCustomer : Customer;
begin
    beginTransaction;
    create newCustomer persistent;
    newCustomer.id := Customer.lastInstance.id + 1;
    newCustomer.name := name;
    newCustomer.myCompany := Company.firstInstance;
    commitTransaction;
end;
```

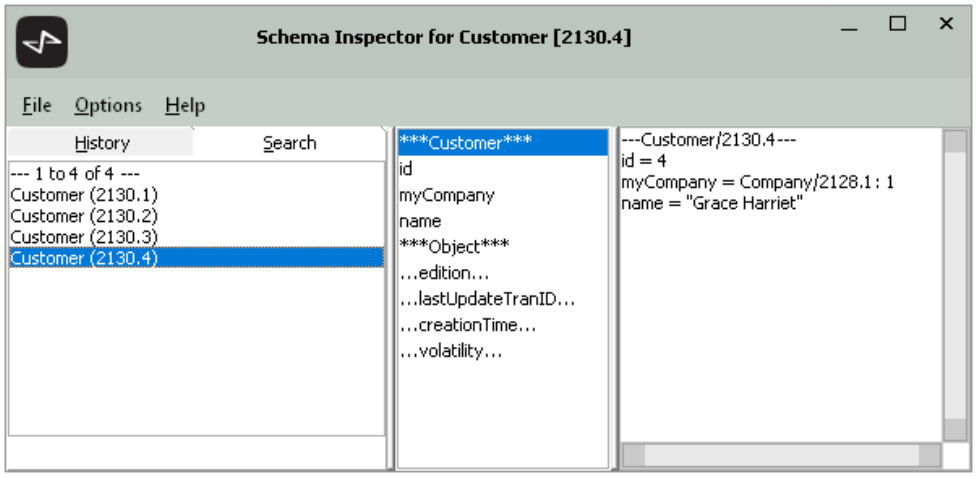
4. Open the **RestfulClient** application, enter **Grace Harriet** in the text box, and then click the **Create Customer** button.



Grace Harriet will be added to the **RestfulSchema** database so that she will appear in the list of customers when the form is automatically reloaded.

5. Open **RestfulSchema** in the Class Browser and then select the **Customer** class. Press Ctrl+I, to open the Schema Collection Inspector.

6. Select the fourth **Customer** in the list.



You can see that **Grace Harriet** has been added to the database.

Security

Jade database systems contain valuable data and intellectual property, and can perform business-critical operations. As such, it is important to consider how that data and intellectual property is kept safe, and how those operations are kept running with high-availability and integrity.

When considering how best to implement and enforce security in the Jade Platform, you should start by asking the following three questions.

1. What specific assets do I need to protect?
2. What threats are possible that could compromise those assets?
3. What tools and best practices can I use to protect those assets from those threats?

For example, consider the Jade database system for a business that gives small business loans. What assets does this Jade system need to protect?

Asset	Example Threat	How will I protect it?
Availability of core system	Tampering could bring down a Remote Access Program (RAP) or application server, preventing users from accessing the system	Control who has access to the system
Integrity of core system	An unauthorized user could approve loans he or she shouldn't be able to; for example, his or her own	Authenticate users to ensure they are who they say they are
Source code	A competitor could steal source code and copy the proprietary algorithm for determining whether to accept a loan	Encrypt the source code
User data privacy	A malicious actor could steal the customers' contact details and spam them	Encrypt data at rest and in transit
Data integrity	If I can make the system "forget" my loan, I could create an infinite money glitch	Control interfaces to the data to restrict what changes can be made

Security Threats

In this module, we will cover the common types of security threats, which can be remembered with the **STRIDE** acronym, as follows.

- **S**poofing
- **T**ampering
- **R**epudiation
- **I**nformation disclosure
- **D**enial of service
- **E**levation of privilege

Spoofing

Spoofing is when a malicious actor impersonates another identity to gain access to which he or she is not authorized.

Note A *malicious actor*, also known as a threat actor, is a person or entity who is responsible for a negative security impact. This actor can be external or internal.

Example

A loan applicant logging in as a loan manager to approve his or her own loan.

Tampering

Tampering is when a malicious actor modifies data in a way that causes harm.

Example

An insider connects to the Jade database server with an application server running the **JadeLogicalCertifier** application and deploys a malicious "fix" that corrupts data.

Repudiation

Repudiation is when the authenticity or origin of a service or action cannot be proven.

Example

A developer logs onto the production environment and loads a schema, but it cannot be determined which developer it was because all developers share the same admin account.

Information Disclosure

Information disclosure is when private or confidential information is leaked to anyone who does not have authority to view that information.

Example

A hacker gains access to the unencrypted data (.**dat**) files of the database and opens them in WinHex to get a list of emails and plain-text passwords.

Denial of Service

Denial of service is when a malicious actor overloads a service with so many requests that it cannot keep up, preventing legitimate users from accessing the service.

Example

A botnet makes thousands of mal-formed web requests per second to the REST service, overloading the web servers and causing customers to be unable to access it.

Elevation of Privilege

Elevation of privilege is when an application gains elevated access because of a bug or exploit.

Example

A Jade application compiles and runs a transient method that includes user input without proper sanitization. The user code does an injection attack to gain access to the Schema Inspector, which causes Information Disclosure.

Note In security, *sanitization* is the process of stripping or replacing special characters from user input, to avoid injection attacks.

Discussion Questions

If you are going through this course module at your own pace, think about and write down your answers to the following questions. If you are going through the course instructor-led, your instructor will pose these questions for a class discussion.

Consider the following situations and determine which of the STRIDE security threats have occurred in each of these situations.

- While John is working on fixing a bug in a method called **filterLoansByCreditRating**, he notices a rather odd snippet of code.

```
// Ensure compliance
foreach loan in allLoans where loan.theBorrower.loans.size > 5 do
    // Only when all loans are last
    if loan.theBorrower.loans.last = loan then
        loan.theBorrower.loans.remove(loan); // needed for ISO requirement
    endif;
endforeach;
```

  This code doesn't make any sense to John, and it seems inappropriate in the context of filtering loans by credit rating. When he tried to find out who added the code and why, nobody in his team knew anything about it.

- After an unexpected reorganization is initiated at 2:38am, Jane is looking through the access logs to see who was online at that time. To her great surprise, the logs show that it was her! But she wasn't online at that time; she was fast asleep.

- Leigh is browsing Reddit late one Friday afternoon. Er, I mean, he's hard at work. Yup. But anyway, on Reddit he sees a link to a list of all his company's customers and their outstanding loan amounts.

The Three 'A's of Access

The three '**A**'s is a set of key security concepts that govern the best practices for controlling access to sensitive data. They are:

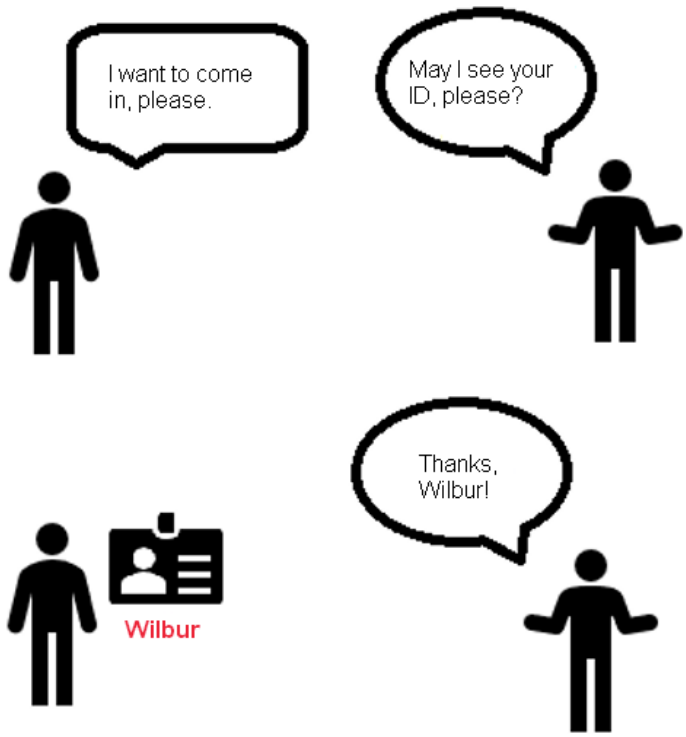
1. **Authentication** – determining the identity of the user.
2. **Authorization** – determining whether the user is allowed to access the data.
3. **Accounting (or Auditing)** – tracking and logging what data a user accessed and any changes he or she made to the data.

You will often hear the terms *authentication* and *authorization* used interchangeably.

There is a subtle difference between them, and we need to do both when deciding whether to allow a request to be executed on the server. The first thing we need to do is *authenticate* the security principle.

Authentication

When we perform an authentication, we check the identity of the security principle; that is, we are verifying that they "are who they say they are".



Authorization

After we know who it is who is attempting access, we then need to *authorize* them. This is when we check a set of rules to determine whether the (now-identified) security principle is allowed to access the resource.

Accounting

After the user's identity has been authenticated and he or she has been authorized to access the data, we still have one more step: Accounting. This involves tracking and logging all data the user accessed and any actions the user takes.

When accounting is done correctly, all changes to data should be able to be traced back to the source of that change, and there should be a record of what data each user has accessed.

This is important for a few reasons. When planning an accounting process, it is helpful to first consider the goals of the accounting, as well as how to meet these goals, as shown in the example in the following table.

Goal	To meet this goal, the…
Detect suspicious activity that may indicate an attack	Accounting process should not only monitor activity but also apply some rules to determine when that activity is suspicious and to generate a report. For example, a user who is authorized for a restricted set of activities attempts to perform every activity for which he or she is not authorized. This can indicate that the user is attempting to find a vulnerability and should automatically be reported to an administrator
Determine the impact of a security breach if it *does* occur	Process should keep a record of all data with which the user interacted (whether modified or only viewed). These records should not be stored in the database itself, as otherwise a malicious actor can delete them.
Provide evidence of criminal proceedings against a malicious actor	Detail and reliability of the records must be sufficient.

Applying the Three 'A's

This section covers applying the three 'A's to desktop applications and Jade REST services.

Desktop Applications

You can use the **getAndValidateUser** and **isUserValid** methods of your schema's **Global** class to authenticate and authorize the users of your Jade desktop applications.

Whenever a Jade application is started, the following authentication process occurs.

1. The **getAndValidateUser** method is called first.

   ```
   getAndValidateUser(usercode: String output;
                      password: String output): Boolean;
   ```

   This method should typically be **clientExecution**, and is intended to obtain credentials from the user.

   The default implementation sets **usercode** to your workstation name suffixed with your operating system process ID, the **password** to null, and returns **true**. You can re-implement the method to replace this behavior with an implementation that obtains credentials from the user. Set the **usercode** and **password** output parameters based on this, then when you return from the method, these credentials are used to authenticate that user in the **isUserValid** method. You can also optionally perform some preliminary authentication in this method; for example, if you wanted to give the user multiple attempts before verifying on the server.

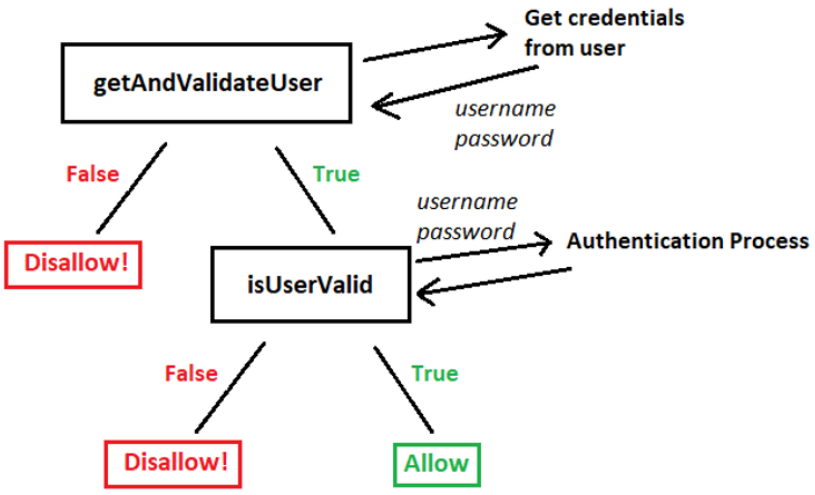
2. The **isUserValid** method is then called.

   ```
   isUserValid(usercode: String;
               password: String): Boolean;
   ```

   This method should typically be **serverExecution**, and is intended for authenticating the credentials.

   The default implementation simply returns **true**, but you can re-implement the method to replace this behavior with your authentication process. When you do, the **usercode** and **password** set in the **getAndValidateUser** method are passed as parameters.

3. If either method in step 1 or 2 returns **false**, the user is disallowed and the application will not start.



While you could implement the authentication process in Jade code, we want to avoid storing the user credentials in the Jade database. If user credentials are stored in the Jade database, it becomes your responsibility to ensure that they are encrypted with a strong and trustworthy encryption algorithm. It is easier and safer to delegate this responsibility to a trusted, re-usable solution from a reliable third-party vendor such as Microsoft Active Directory.

One way to do this is with the Application Programming Interface (API) provided by **CardSchema**, available free from the Jade Developer Center web site (https://www.jadeplatform.com/developer-centre/extensions).

CardSchema provides the **CnExternalMethods** class, which includes a **cnUserCheck** method. This method takes a username and password as input parameters, and an output parameter for the result. This method calls a Microsoft library that validates the username and password against the domain log ins, then sets the result to zero for successful validation or a non-zero error code for unsuccessful validation. Using this technique, we can authenticate the user without having to store user credentials in the Jade database.

Note Microsoft Windows login credentials will authenticate only the user (that is, verify who he or she is); the credentials will not be sufficient to authorize the user (that is, verify that the user is allowed to log on to the system).

REST Web Service Security

You can secure your Jade REST services by requiring consumers to include JSON Web Tokens to authenticate their requests and associating required claims with your REST service methods to enforce authorization rules.

When dealing with the web, there are a lot of acronyms and terms with specific definitions. You can use the following as a reference for the terms we will use in this section.

Term	Meaning
REST (Representational State Transfer)	An architectural style for web services where the server is stateless – the server forgets the client as soon as a request is fulfilled.
REST Service	An application that responds to REST requests over the web.
Security Principle	A person or program that requires access to a secured service.
Resource	The data or operation of the REST service that the security principle is attempting to access.
Consumer	The client of a REST service. Makes requests to the service and gets responses back.
JSON (JavaScript Object Notation)	A standard format for describing objects.
JWT (JSON Web Token)	A set of claims about a security principle plus a signature that proves that the token came from a trusted source.

When a Jade REST service has not been secured, anyone who knows the correct Uniform Resource Locators (URLs) can access anything in your REST API. The traditional way of restricting access is by having a security principle log in, providing some secret or password to authenticate them, then authorizing them for the appropriate actions based on their identity.

As REST services are stateless, there is an additional complication.

The server does not store any session information and therefore cannot determine whether the client is logged in. In lieu of this, we can provide the client with a security token after authentication. He or she can then include this token in future requests, removing the need to re-authenticate.

What is a JSON Web Token?

A JSON Web Token is a string made up of the following segments.

- Header, with meta-information about the token itself; for example:

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- Payload, with a set of claims about the identity of the bearer; for example:

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

- Signature, which proves that the token came from a trusted source; for example:

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Each segment is then base64-encoded and delimited by a period, resulting in the following token.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ
SI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P
Ok6yJV_adQssw5c
```

In this token we have five claims. Two are in the header:

1. **"alg": "HS256"** – the token was signed using the HS256 signing algorithm.
2. **"typ": "jwt"** – the token is a JSON Web Token.

The header claims are meta-information; that is, claims about the token itself.

There are also three claims in the payload:

1. **"sub": "1234567890"** – the subclaim describes a unique identifier for the subject of the token; that is, who the token is about.
2. **"name": "John Doe"** – the name of the subject of the token.
3. **"iat": "1516239022"** – the time at which the token was issued in the JSON **NumericDate** format, which is the number of seconds since midnight on the first of January 1970.

These payload claims are information about the subject of the token; that is, the security principle to be authenticated. There are many more claims you can put in the payload, such as the issuer of the token, the time at which the token should expire, and the audience of the token (by whom the token is intended to be validated). You can also make up your own claims to describe the security principle however you like, which can be useful when applying authorization rules.

Finally, the signature is generated by base64-encoding the header and payload, then encrypting the result with a secret, in this case using the HS256 algorithm and the secret **your-256-bit-secret**. When the server validates the signature, it can perform the same process on the header and payload, and if it was signed with a different secret or if anything in the header/payload has been modified, the signatures will not match and the token will be rejected.

Symmetrical vs Asymmetrical Tokens

The symmetry of a token refers to the linked concepts of signing algorithm choice and whether it is the same entity that generates as validates the token.

In a symmetrically signed token, the token is generated and issued by the server itself and it will use a single secret that can encrypt and validate the token. This is the example we saw above. These tokens will use an HS encryption algorithm (HS256, HS384, HS512).

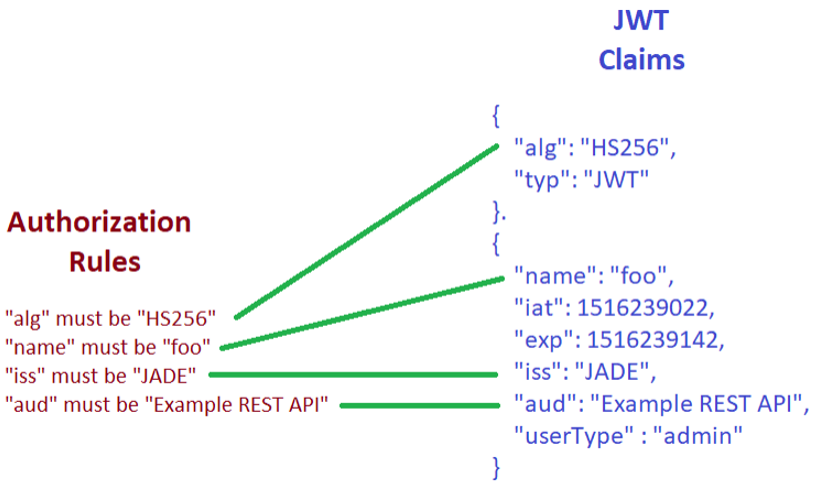
In an asymmetrically signed token, the token is generated by a trusted third party and will be signed with a public/private key pair. This will allow the server to use the public key to verify the token while only the trusted third party has the private key used to sign it. These tokens usually use an RS encryption algorithm (RS256, RS384, RS512) but there are other algorithms that are less common; for example, Elliptical-curve encryption.

Generating a JSON Web Token from Jade

You can generate symmetrically signed JSON Web Tokens from Jade using the **JadeJsonWebToken** class.

Enforcing Authorization Rules with JSON Web Tokens

As mentioned previously, the purpose of a JSON Web Token is to allow the user to authenticate once only and obtain a token that can then be used to authorize multiple REST requests. This works by having a set of required claims associated with the REST resource and then comparing them with the claims included in a JSON Web Token that has been included in a REST request.



In this example, all of the authorization rules, also known as required claims, are present in the provided token.

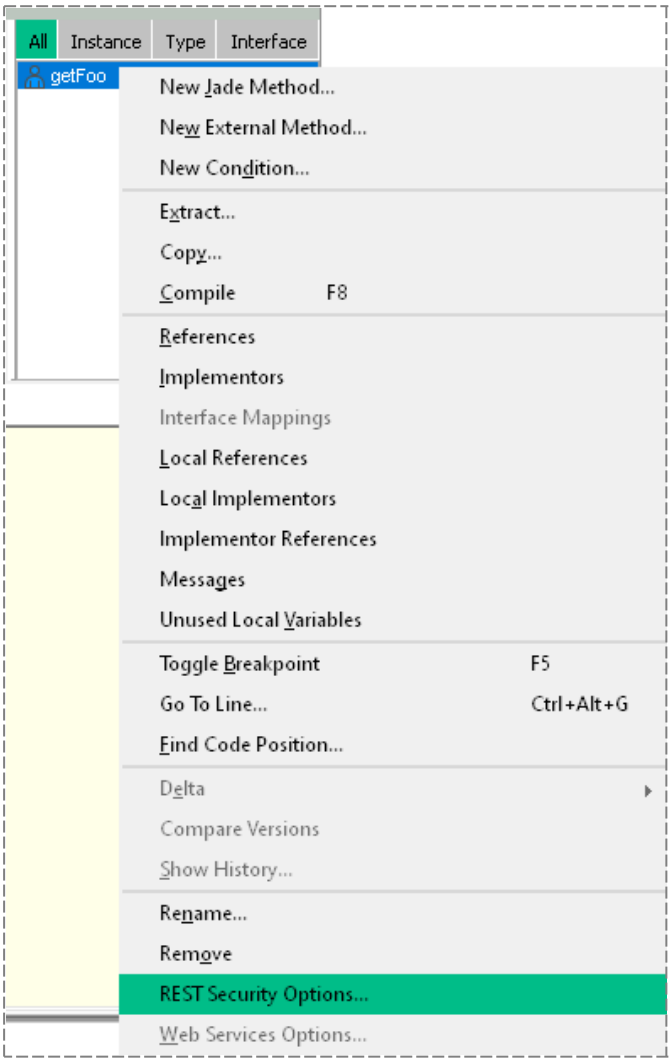
If the token's signature is valid and the token hasn't expired, the request will be allowed. Notice that the token also has a claim **"userType"**, which isn't required by the rules. That's allowed, as the token may have superfluous claims, as long as all of the required claims are present.

From Jade 2025 R2, you can configure JSON Web Token (JWT) authentication at the class level, applying the specified JWT claims to *all* REST methods defined on that class and its subclasses (excluding imported classes). Both class-level and method-level JWT authentication is inherited by subclasses, and can be reimplemented at the subclass level. For details, see "REST Service Security", in Chapter 2 of the *Object Manager Guide*.

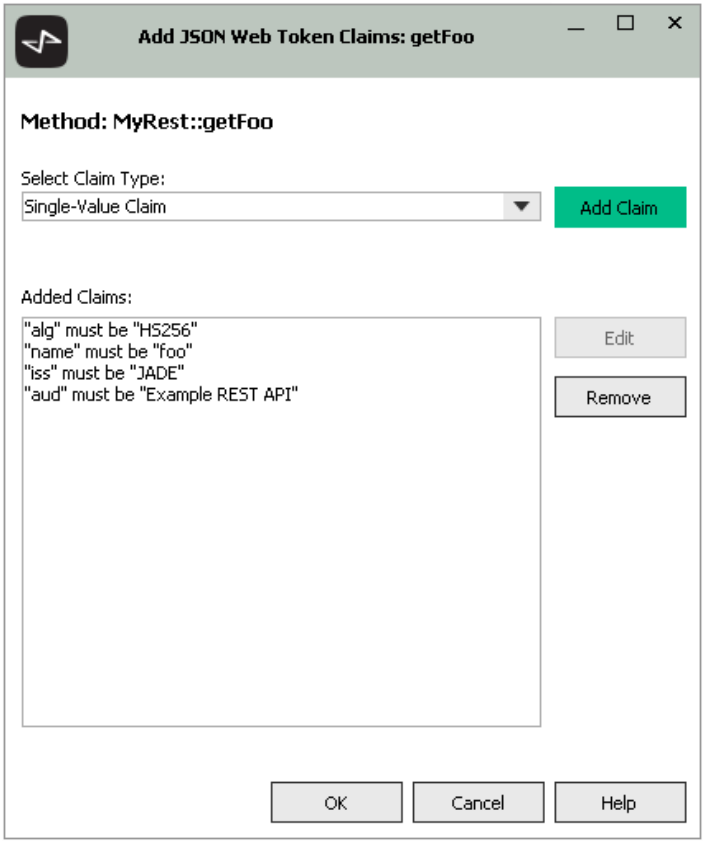
Note Method-level JWT authentication always takes precedence over class-level authentication, even when no JWT claims are defined at method level.

To associate a set of authorization rules with a REST resource method in Jade, we use the Add JSON Web Token Claims dialog.

- Select the **REST Security Options** command of the method's context menu, shown in the following image.



The Add JSON Web Token Claims dialog, shown in the following image, is then displayed.



If any claims are associated with a method, all incoming requests must be authorized. If a request fails authorization, an HTTP response code 403 (*Forbidden*) will be returned.

To pass authorization, the following conditions must be met.

- A JSON Web Token must be present in the header of the request.
- The token's signature must be valid. How this occurs will depend on whether it is a symmetrically or asymmetrically signed token.
- All required claims must be present in the token.
- Default claims must be valid. For example, the expiry (**"exp"**) claim must be a time in the future.

Mitigation of Potential Vulnerabilities

When using JSON Web Tokens, there are two main security vulnerabilities to know about, and both are mitigated for you by Jade.

- "alg": "none" vulnerability

  One of the options for the signing algorithm is **"none"**, where no signature is provided. The intent of this option is for testing purposes only, as any malicious actor could just send in such a token with whatever arbitrary claims he or she chooses. In Jade, the **"none"** algorithm is not supported and all tokens with **"alg": "none"** will be rejected. In fact, Jade uses a Deny by Default approach and only HS256, HS384, HS512, and RS256 algorithms are allowed. As such, you do not need to do anything to mitigate this vulnerability, as Jade does it for you.

- RSA vs HMAC vulnerability

  If a malicious actor sends a token he or she has created and set the **"alg"** claim to HS256 but actually pass in your auth provider's RS256 public key (which anyone can get, as it's public), some security libraries may treat the RS256 key as an HS256 key and erroneously accept it. By implementation, Jade is not vulnerable to this attack, so setting your **"alg"** as a required claim is entirely optional.

Exception Handling and Deny by Default

When considering how to enforce authorization rules, it is not sufficient to consider only code paths that resolve as expected. It is also important to consider what will happen if an unexpected exception occurs.

The core principle for all authorization code is Deny by Default; that is, if the authorization code is unable to complete successfully, the default behavior should be to deny access rather than to allow it.

To illustrate this idea, we will consider some authorization methods, which will each call the following validation method.

```
validateUser(password : String) : Boolean;

begin
    return password = self.credentialStore.password;
end;
```

The important thing to notice about this method is that if the **credentialStore** reference is null, we will get a 1090 exception.

Consider this first method, in which we perform a simple authorization that should be allowed.

```
authenticationExample_ALLOWED();

vars
    validator : Validator;
    credential : Credential;

begin
    on Exception do handleException(exception); // returns "Resume Next"

    create credential transient;
    credential.password := "foobar";
    create validator transient;
    // We set the credential store.
    validator.credentialStore := credential;

    // The validateUser method will return true.
    if not validator.validateUser("foobar") then
        write "The user was forbidden.";
        return;
    endif;

    // So we will get this result.
    write "The user was authenticated";

epilog
    delete validator;
    delete credential;
end;
```

Everything is good so far, but let's see what happens if the passwords do not match.

Note As with all of the examples and exercises in this module, we will be looking at simple situations to illustrate one point at a time. You will notice that it is not doing anything when the user is authenticated; we are merely writing to the console log. In addition, we are faking the credentials by just using a transient object. The idea is to focus on the key concept of each example so that you can apply it to a wide variety of situations.

```
authenticationExample_FORBIDDEN();

vars
    validator : Validator;
    credential : Credential;

begin
    on Exception do handleException(exception); // returns "Resume Next"

    create credential transient;
    credential.password := "foobar";

    create validator transient;
    // We set the credential store.
    validator.credentialStore := credential;

    // The provided password is not the same as the stored credential.
    if not validator.validateUser("not foobar") then
        // So we are forbidden.
        write "The user was forbidden.";
        return;
    endif;

    // We don't get to here.
    write "The user was authenticated";

epilog
    delete validator;
    delete credential;
end;
```

As we would expect, the **validateUser** method will return **false** and the user is forbidden.

For simple cases, this implementation will therefore work but it fails the Deny by Default rule. Let's see what happens if we have an exception.

```
authenticationExample_EXCEPTION();

vars
    validator : Validator;

begin
    on Exception do handleException(exception); // returns "Resume Next"

    // This time we haven't set any credential to the validator so we will get a 1090
    create validator transient;

    // This method will generate an exception and we will resume next, bypassing the whole if instruction.
    if not validator.validateUser("foobar") then
        write "The user was forbidden.";
        return;
    endif;

    // So we will end up here and erroneously authorize the user.
    write "The user was authenticated";

epilog
    delete validator;
end;
```

Here, when the **validateUser** method gets an exception, the exception handler's resume next bypasses the **if** instruction and we end up in the default code path. Since we did not Deny by Default, the user is erroneously authorized.

Exercise 1 – Applying Deny by Default

In this exercise, rewrite the **authenticationExample** method to deny access when an exception occurs.

1. Using the Schema Loader, load the **ExceptionSchema.scm** and **ExceptionSchema.ddx** files.
2. Add a new **JadeScript** class method called **authenticate** and code it as follows.

```
authenticate();

vars
    validator : Validator;
    credential : Credential;
begin
    on Exception do handleException(exception);

    create credential transient;
    credential.password := "foobar";
    create validator transient;
    validator.credentialStore := credential;

    if validator.validateUser("foobar") then
        write "The user was authenticated";
        return;
    endif;

    write "The user was forbidden.";

epilog
    delete validator;
    delete credential;
end;
```

3. Execute the method. You should see that the user is authenticated.

4. Modify the **authenticate** method so that the password no longer matches.

```
authenticate();

vars
    validator : Validator;
    credential : Credential;
begin
    on Exception do handleException(exception);

    create credential transient;
    credential.password := "foobar";
    create validator transient;
    validator.credentialStore := credential;

    if validator.validateUser("Some wrong password") then
        write "The user was authenticated";
        return;
    endif;

    write "The user was forbidden.";

epilog
    delete validator;
    delete credential;
end;
```

5. Execute the method. You should see that the user is not authenticated. This is as expected, because the password does not match.

6. Now we will generate an exception to make sure we are not vulnerable to erroneous authorization when an exception occurs.

7. Navigate to the **Validator** class and modify the **validateUser** method.

```
validateUser(password : String) : Boolean;

begin
    write 1/0;
    return password = self.credentialStore.password;
end;
```

   The method will now generate an exception when it is run.

8. Execute the authenticate method (which should still have the invalid password). You should see that the user is still forbidden.

   By having the default behavior being to deny access and then executing the code relating to a successful authorization only if the validation passes, we have made our code resilient to unauthorized access when things go wrong.

Transient Methods and Code Injection

Transient methods are a powerful tool for dynamically creating Jade code at run time. However, it is critical to ensure that the transient method is going to perform as you expect it to before execution, as it can be vulnerable to code injection attacks.

Consider the following method that creates and executes a transient method. It is a simple example as the use of a transient method is superfluous, but it serves to illustrate an important idea.

```
generateStatement(cust : Customer) : String;

constants
    SourceTemplate : String =
'createStatement(): String;
vars
    statement : String;
begin
    statement := "This is a statement for {CustomerName}" & CrLf & CrLf & "Your balance is: {CustomerBalance}";
    return statement;
end;
';

vars
    transMeth : JadeMethod;
    sourceCode : String;
    errCode, errPos, errLen : Integer;

begin
    sourceCode := SourceTemplate.replace__("{CustomerName}", cust.name.trimBlanks(), false)
                                .replace__("{CustomerBalance}", cust.balance.String, false);

    transMeth := process.createTransientMethod("createStatement", Customer, currentSchema,
                                       sourceCode, false, String,
                                       errCode, errPos, errLen);

    return process.executeTransientMethod(transMeth, cust).String;

epilog
    delete transMeth;
end;
```

This method has a template for the source code of its transient method with two placeholders: **{CustomerName}** and **{CustomerBalance}**.

We are replacing these placeholders with values from a customer object, then creating a transient method with the source and executing it.

The **{CustomerBalance}** placeholder is set to the **balance** property of the customer. That's not an issue, because it is of type **Decimal** and decimals have a specific format that is not vulnerable to injection.

The **{CustomerName}**, however, is set to the name of the customer, which is a **String**. Strings can get up to all sorts of mischief.

If we call the method with a customer who has a name **John Smith**, the method will behave as expected and the transient method will generate a **String**.

```
This is a statement for John Smith

Your balance is: 12550.20
```

So far so good. But what if the customer has the name:

```
"; currentSchema.inspectModal(); //
```

A bit of a funny name, but if we just let users type whatever they want for their name, nothing stops them from choosing such a name. Let's look at the source code for the transient method if we have that name.

```
createStatement() : String;
vars
        statement : String;
begin
        statement := "This is a statement for "; currentSchema.inspectModal(); //" & CrLf & CrLf & "Your balance is: 12345.78";
        return statement;
end;
```

The initial quote will escape the string that is supposed to be assigned to the **statement** variable, then the semicolon finishes that statement. It then executes the **currentSchema.inspectModal();** command, which will allow the user to see all information about all classes and objects in the entire schema. Alternatively, this could be any malicious code. Finally, it ends with **//** to comment out the rest of the line of code.

Using this method, the malicious actor can execute whatever code they want to against the database.

Mitigation of Potential Vulnerabilities

When using JSON Web Tokens, there are two main security vulnerabilities to know about, and both are mitigated for you by Jade.

- "alg": "none" vulnerability

  One of the options for the signing algorithm is **"none"**, where no signature is provided. The intent of this option is for testing purposes only, as any malicious actor could just send in such a token with whatever arbitrary claims he or she chooses. In Jade, the **"none"** algorithm is not supported and all tokens with **"alg": "none"** will be rejected. In fact, Jade uses a Deny by Default approach and only HS256, HS384, HS512, and RS256 algorithms are allowed. As such, you do not need to do anything to mitigate this vulnerability, as Jade does it for you.

- RSA vs HMAC vulnerability

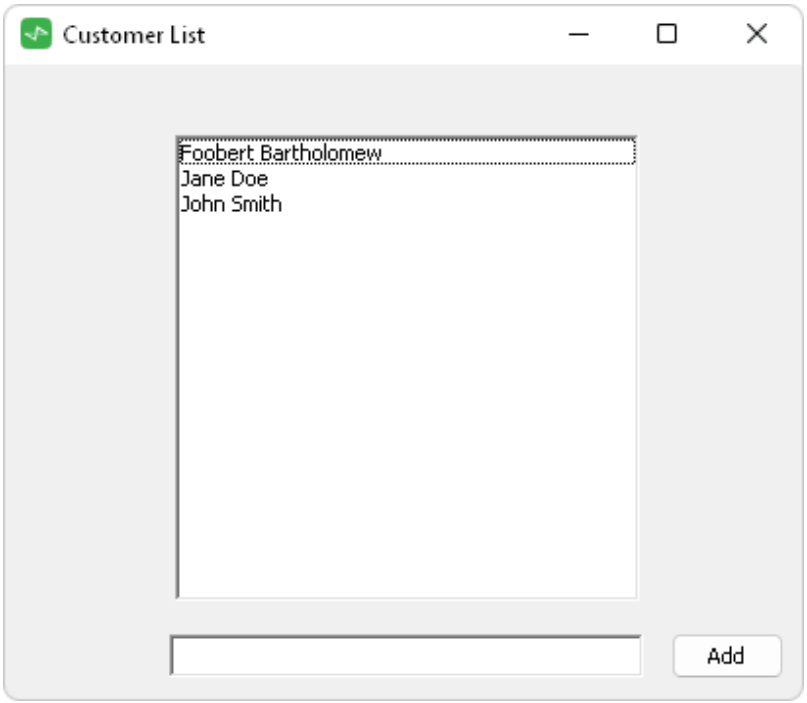
  If a malicious actor sends a token he or she has created and set the **"alg"** claim to HS256 but actually pass in your auth provider's RS256 public key (which anyone can get, as it's public), some security libraries may treat the RS256 key as an HS256 key and erroneously accept it. By implementation, Jade is not vulnerable to this attack, so setting your **"alg"** as a required claim is entirely optional.

Exercise 2 – Code Injection

In this exercise, use code injection to attack a poorly implemented system.

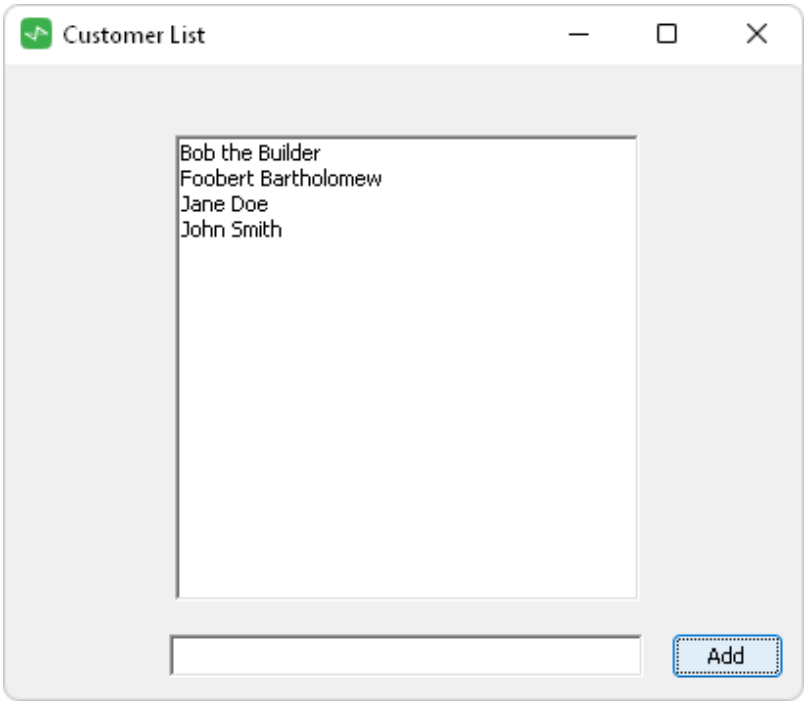
1. Close Jade if it is running, then restart it in multiuser mode; that is, start a database server and a fat client.
2. Use the Schema Loader to load the **TransientInjection.scm** and **TransientInjection.ddx** files.
3. From the **TransientInjection** schema, run the **TransientInjection** application.

The Customer List dialog, shown in the following image, is then displayed.



This simple application displays the names of some customers in a list box. In the text box at the bottom of the dialog, you can enter the name of a new customer, which is added to the list after you click the **Add** button.

4. Enter **Bob the Builder** into the text box and then click the **Add** button.



The list box in the Customer List dialog is populated in an unusual way. It uses a transient method to update itself whenever a new customer is added. I wonder what mischief can we get up to?

5. Try to do as much damage as possible to the database with your 30-character limit for the customer's name.

   Use the following goals to get you started:

   - Try to bring down the fat client.
   - Can you access or delete data from another schema?

Unit Testing

Unit testing is an important tool that is used to identify defects in a system during development. The key idea is to develop tests for a unit of code (which can be a fragment, a method, or a module) as the code is being written.

The Jade Platform unit testing framework enables unit tests to be executed automatically and for the results of the tests to be captured. By doing so, the test provides a contract of the minimum functionality that a unit of code must satisfy.

In addition to providing a contract of functionality, unit tests provide the following additional benefits.

- Enables refactoring without regression (that is, ensuring that the module still works correctly)
- Eliminates uncertainty in the units
- Enables a bottom-up testing style approach
- Documents the functionality provided by the unit and how to use it

JadeTestCase Class

Unit tests are added as subclasses of the **JadeTestCase** class in **RootSchema**.

The **JadeTestCase** class is an abstract class that provides common functionality for unit tests as well as providing the **JadeTestListenerIF** interface that allows callback methods to report on the progress of a set of test cases.

The following methods are defined in the **JadeTestCase** class. By using these methods in test cases, you can define the expected behavior of your tested module.

Method	Code Example
assert	`// Sets the message to be displayed upon test failure` `assert("the unit test failed");`

assertEquals	`// Compares the first parameter to the second,`
	`// and fails the test if they are not equal.`
	`assertEquals(2, (1+1));`

assertEqualsMsg	`// Compares the first parameter to the second, and fails the`
	`// test if they are not equal, displaying the given message.`
	`assertEqualsMsg("one plus one was not two", 2, (1+1));`

assertFalse	`// Asserts that a given boolean evaluates to false. If the`
	`// condition evaluates to true, the test fails.`
	`assertFalse(1 = 2);`

assertFalseMsg	`// Asserts that a given boolean evaluates to false. If the`
	`// condition evaluates to tryue, the test fails, dislaying`
	`// the given message.`
	`assertFalseMsg("One equals two was true", 1 = 2);`

assertNotNull	`// Asserts that an object exists. If this is not the case,`
	`// the test fails.`
	`assertNotNull(myObject);`

Method	Code Example

assertNotNullMsg	`// Asserts that an object exists. If this is not the case,`
	`// the test fails, and the given message is displayed.`
	`assertNotNullMsg("Object doesn't exist.", myObject);`

assertNull	`// Asserts that an object doesn't exist. If it does exist,`
	`// the test fails.`
	`assertNull(myObject);`

assertNullMsg	`// Asserts that an object doesn't exist. If it does exist,`
	`// the test fails.`
	`assertNullMsg("Expected object to not exist", myObject);`

assertTrue	`// Asserts that a condition is true. If the condition is`
	`// false, the test fails.`
	`assertTrue(1 + 1 = 2);`

assertTrueMsg	`// Asserts that a condition is true. If the condition is`
	`// false, the test fails, and the given message is displayed.`
	`assertTrueMsg("One plus one should equal two", 1 + 1 = 2);`

expectAbort	`// Registers that the test method is expected to abort`
	`// execution.`
	`expectAbort(true);`

expectedException	`// Registers an exception that the test should produce.`
	`// The test will fail if no exception is raised.`
	`expectedException(1090);  // Null object reference`

expectTerminate	`// Registers that the test method is expected to terminate.` `expectTerminate(true);`
info	`// Outputs the given message without failing the test` `info("The test is going fine so far.");`

Writing a Test Case

When writing a unit test, you must first create a subclass of the **JadeTestCase** class and add the unit tests of methods in that subclass. Unit tests have the **unitTest** method option, shown in the following example.

```
exampleTest() unitTest;

vars
    calc : Calculator;
begin
    calc := create Calculator() transient;
    assertEquals(2, calc.add(1, 1));

epilog
    delete calc;
end;
```

Note When using the **unitTest** method option, the method cannot have any parameters or a return type.

In addition to the **unitTest** method option, you can use the following method options to establish pre- and post-conditions of unit tests.

Method Option	The method will be run…
unitTestBefore	Before every method of the class that uses the **unitTest** method option. It is used to enforce pre-conditions that may be impacted during test execution. The **unitTestBefore** method option should be defined in one method only in each class.
unitTestAfter	After every method of the class that uses the **unitTest** method option. It is used to enforce post-conditions that may be impacted during test execution.
unitTestBeforeClass	Once before the first **unitTest** method of a class is run. It is used to enforce pre-conditions that are required specifically for the tests of a class and that are unlikely to be impacted during test execution.
unitTestAfterClass	Once after the last **unitTest** method of a class is run. It is used to enforce post-conditions that are required at the end of the tests of a class but do not need to be maintained between tests within the class.
unitTestBeforeAll	Once before the first **unitTest** method of the first class is run. It is used to enforce pre-conditions that are relevant to all test classes and that are unlikely to be impacted during test execution. The method in which this option is specified must be defined directly in the **JadeTestCase** class.
unitTestAfterAll	Once after the last **unitTest** method of the last class is run. It is used to enforce post-conditions that are required at the end of the entire set of tests across all test classes and that do not need to be maintained between tests or test classes. The method in which this option is specified must be defined directly in the **JadeTestCase** class.

Note Specify each of the **unitTestBefore**, **unitTestAfter**, **unitTestBeforeClass**, and **unitTestAfterClass** method options in this table in one method only in each class.

The **unitTestBefore** and **unitTestAfter** method options are defined only on the base **JadeTestCase** class, and there can be one instance only of these method options in a schema.

Running a Test Case

You can run unit test methods from any of the following.

- The Class Browser, by selecting:
  - A unit test class or method and then pressing F9.
  - The **Unit Test** command from the Jade menu.

  If you selected a single test method, the test runner runs only that test. If you selected a test class, the test runner runs only the tests of that class.
- The Schema Browser, by selecting:
  - A schema that contains at least one unit test class and then pressing F9.
  - The **Unit Test** command from the Jade menu.

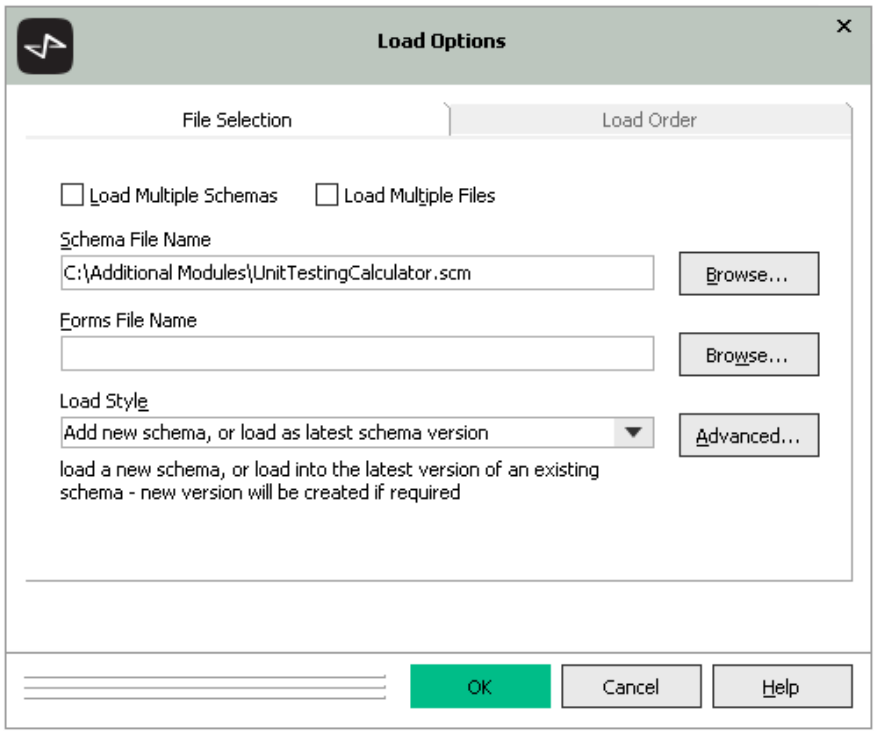
  The Unit Test Runner form is then opened, displaying all of the tests of that schema. The tests are not run until you select one or more tests and then click the **Run** button.
- Calling the **JadeTestRunner** class **runTests** method from your code, passing the collection of test classes in to that method as a parameter.

Note As it is not common practice to run Jade tests from code, you should do so only if you specifically need more control over how the results of the tests are presented.

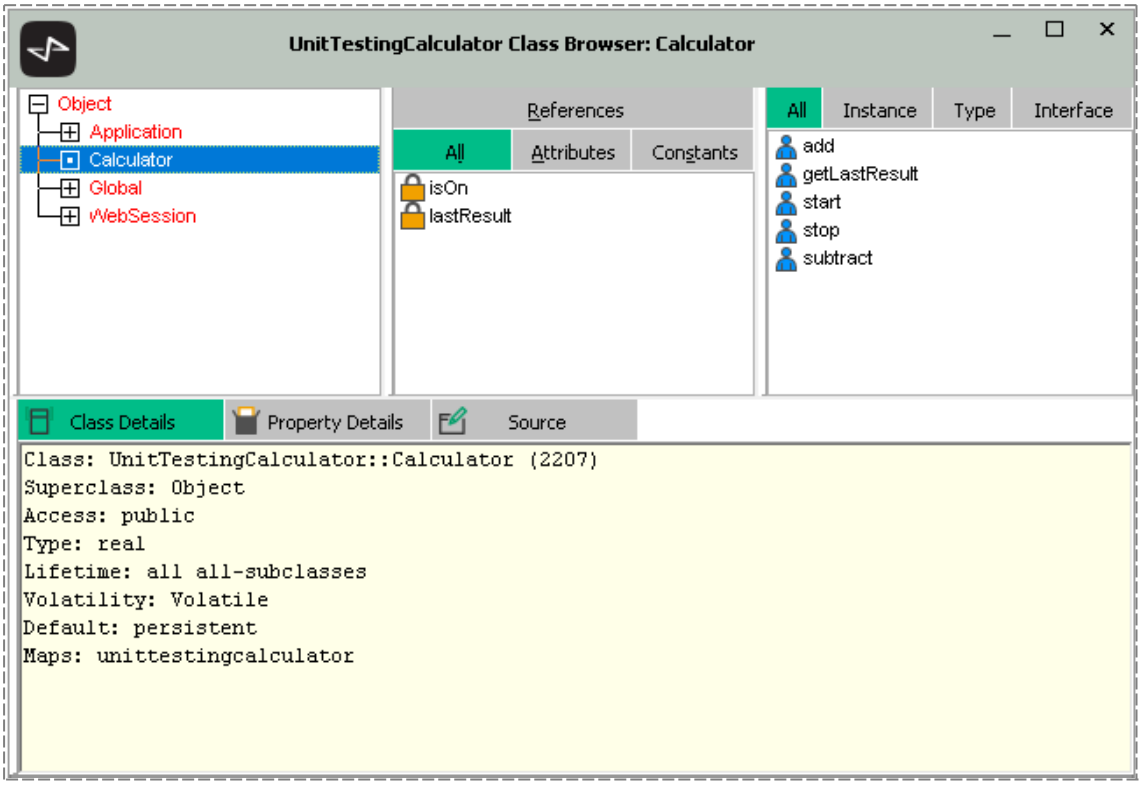
Exercise 1 – Loading the UnitTestingCalculator Schema

In this exercise, load the **UnitTestingCalculator** schema and then locate the **JadeTestCase** class.

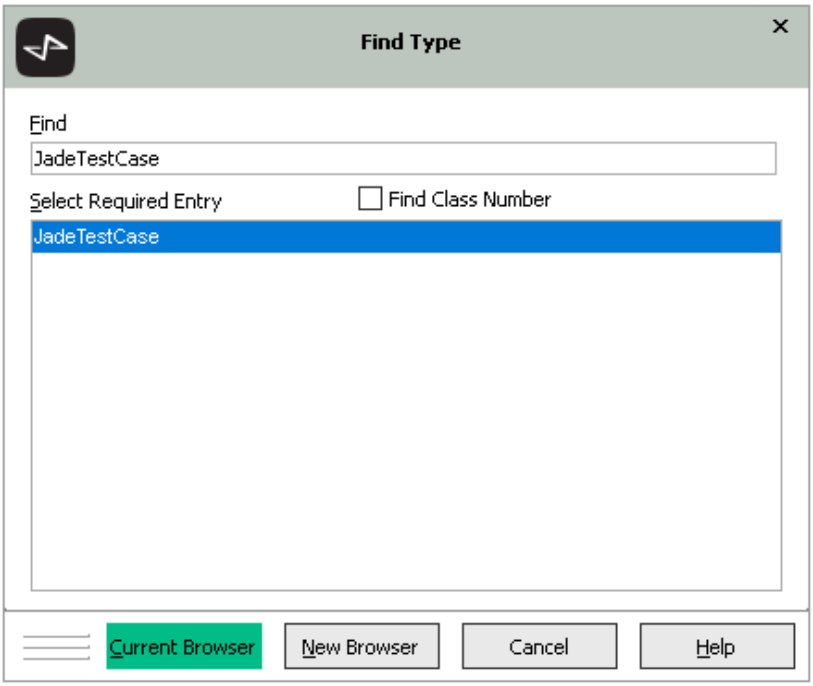
1. Right-click on **RootSchema** in the Schema Browser and then select the **Load** command.
2. Click the **Browse** button at the right of the **Schema File Name** text box and then locate and select the **UnitTestingCalculator.scm** file from the Zip file you downloaded from the Jade web site.



3. Open the **UnitTestingCalculator** schema in the Class Browser and then select the **Calculator** class so that the properties and methods shown in the following image are displayed.



4. Press F4 to display the Find Type dialog and then search for **JadeTestCase**.



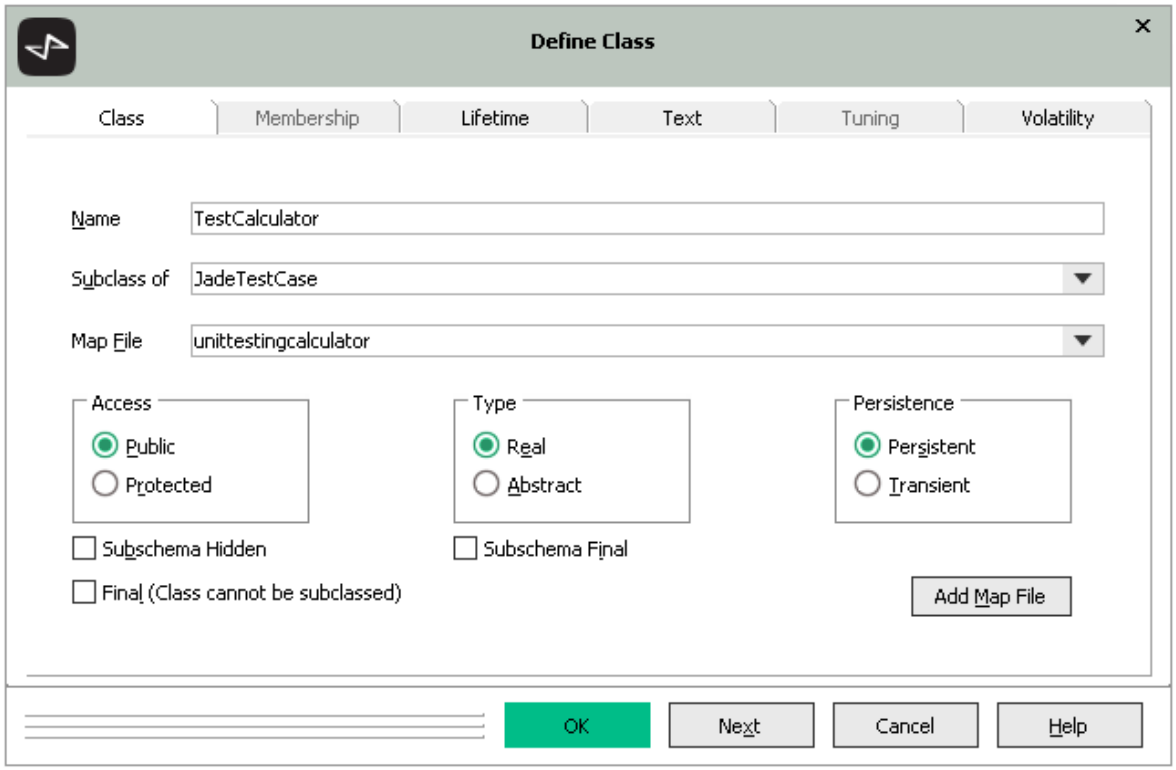
5. Click the **Current Browser** button, to display the **JadeTestCase** class in the Class Browser.

Exercise 2 – Writing a Test Case

In this exercise, write a unit test for the **Calculator** class **add** method.

1. Right-click on the **JadeTestCase** class and then select the **Add** command.

   In the Define Class dialog, specify **TestCalculator** as the class name and then click the **OK** button.



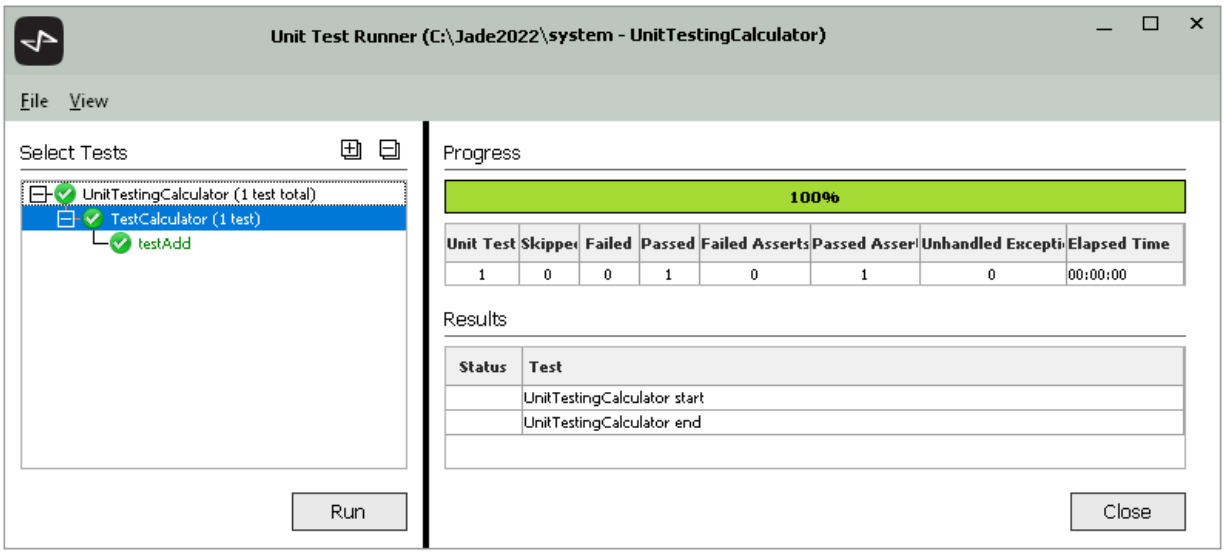
2. Add a reference called **calc** of type **Calculator** to the class.
3. Add a new method called **testAdd** and then check the **Updating** and **Unit Test** check boxes.

4. Code the method, as follows.

```
testAdd() unitTest, updating;

vars
    actual      : String;
    expected    : String;
begin
    self.calc   := create Calculator() transient;
    self.calc.start();

    actual      := self.calc.add(1, 1);
    expected    := "2";

    assertEquals(expected, actual);

epilog
    self.calc.stop;
    delete self.calc;
end;
```

5. Run the method, by pressing F9.

The Unit Test Runner form displays the results of the unit test, which should have passed.



Exercise 3 – Test Case Failure

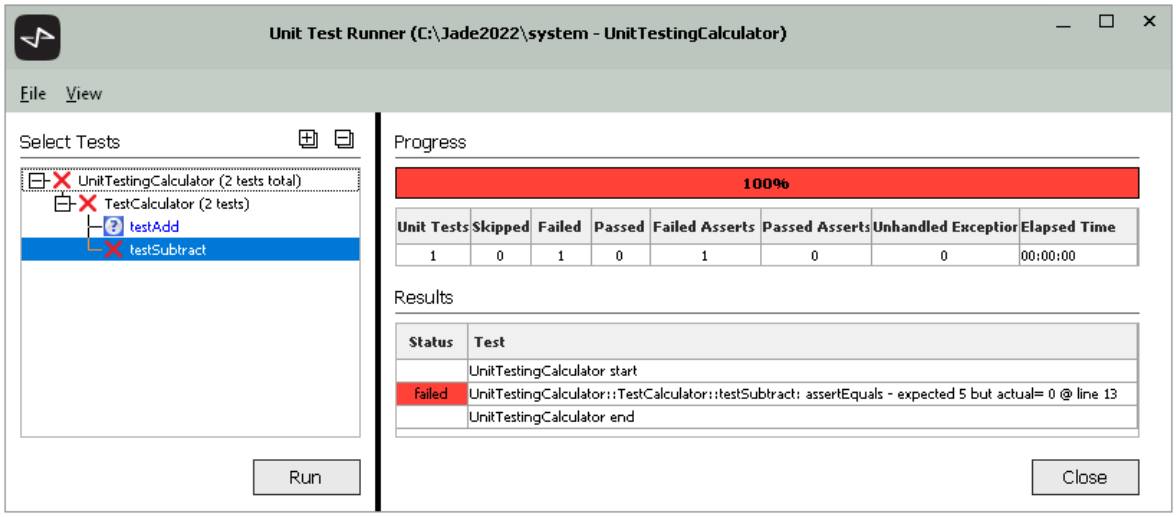
In this exercise, write a test case for the test case for the faulty **subtract** method of the **Calculator** class.

1. Add a **testSubtract** method to your **TestCalculator** class.
2. Code the method, as follows.

```
vars
    actual      : String;
    expected    : String;
begin
    self.calc   := create Calculator() transient;
    self.calc.start();

    actual      := self.calc.subtract(8, 8);
    expected    := "5";

    assertEquals(expected, actual);

epilog
    self.calc.stop;
    delete self.calc;
end;
```

3. Run the method, by pressing F9.

   The Unit Test Runner form then displays the results of the unit test, which should have failed.

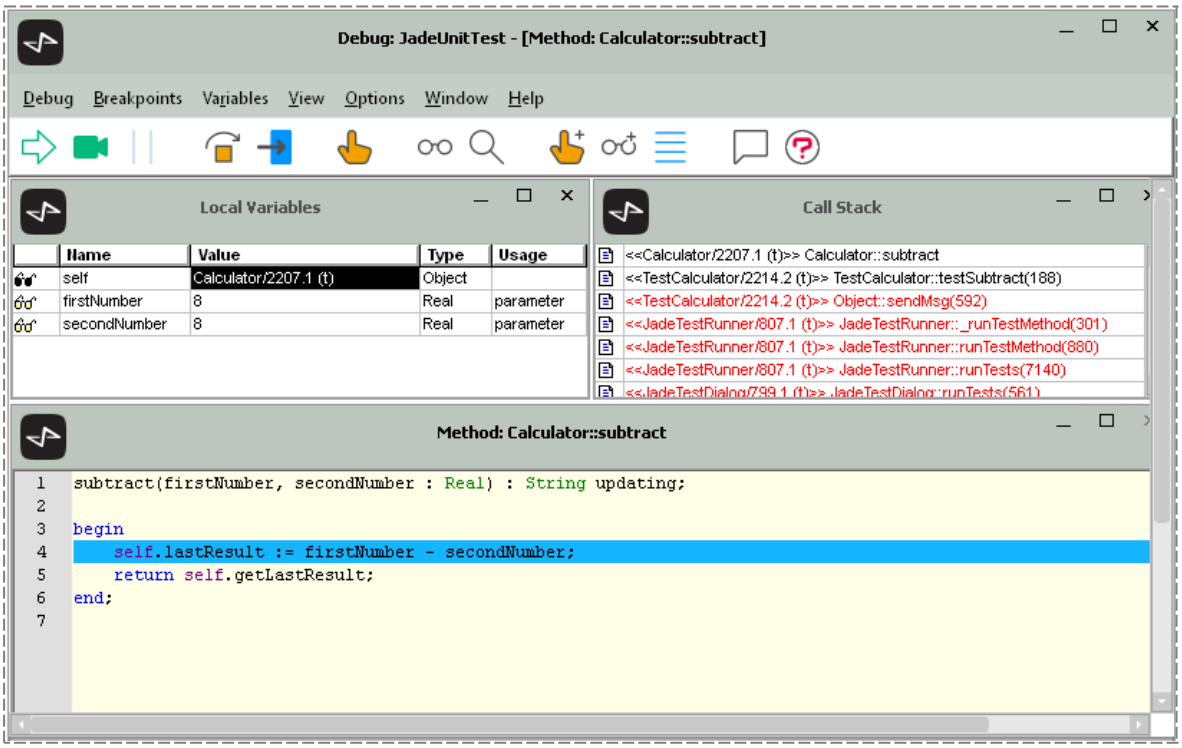


4. Looking at the **Results** section, we see that the unit test expected a result of **5** but the **testSubtract** method returned zero (**0**).

5. Use the F5 key to add a breakpoint to the **testSubtract** method, as follows.

```
testSubtract() unitTest, updating;

vars
    actual      : String;
    expected    : String;
begin
    self.calc   := create Calculator() transient;
    self.calc.start();

    actual      := self.calc.subtract(8, 8);
    expected    := "5";

    assertEquals(expected, actual);

epilog
    self.calc.stop;
    delete self.calc;
end;
```

6. Run the test again, but this time using Shift+F9 to run it in debug mode.



7. Click the **Continue execution** button at the far left of the toolbar (or press F9) to skip to the breakpoint, and then step into the code using the F7 shortcut key until you have found the fault.

Exercise 4 – unitTestBefore and unitTestAfter Method Options

You may have noticed that there is a significant amount of redundant code in the **testAdd** and **testSubtract** methods. In this exercise, refactor this code into **unitTestBefore** and **unitTestAfter** methods that will automatically run for each unit test.

1. Add a **setUp** method to your **TestCalculator** class and code it as follows.

```
setUp() updating, unitTestBefore;

begin
    self.calc   := create Calculator() transient;
    self.calc.start();
end;
```

2. Add a **tearDown** method to your **TestCalculator** class and code it as follows.

```
teardown() updating, unitTestAfter;

begin
    self.calc.stop();
    delete self.calc;
end;
```

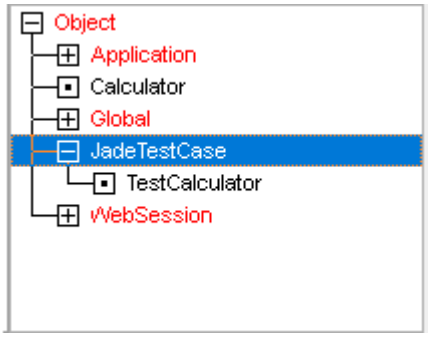
Note The **unitTestBefore** option in the method signature specifies that the method will automatically run *before* each unit test. The **unitTestAfter** method option specifies that the method will automatically run *after* each unit test.

3. Change the **testAdd** and **testSubtract** code to remove the redundant code that has now been factored out to your new **setUp** and **tearDown** methods, as shown in the following method examples.

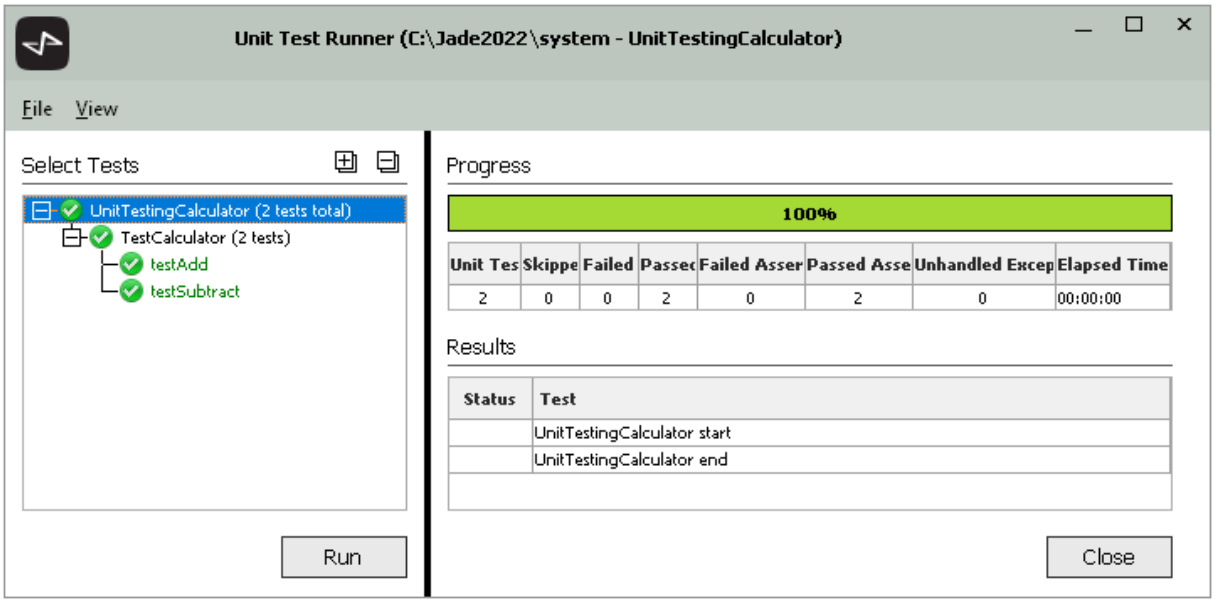
```
testSubtract() updating, unitTest;

vars
    actual      : String;
    expected    : String;

begin
    actual      := self.calc.subtract(8, 3);
    expected    := "5";

    assertEquals(expected, actual);

end;
```

Note The **testSubtract** code is also correct so that the unit test runs successfully.

4. Run all of the unit tests by selecting the **JadeTestCase** class in the Class Browser and then pressing F9.



The Unit Test Runner form is then displayed.



Tip You can achieve a similar result by selecting the class in the Class Browser and then pressing F9.

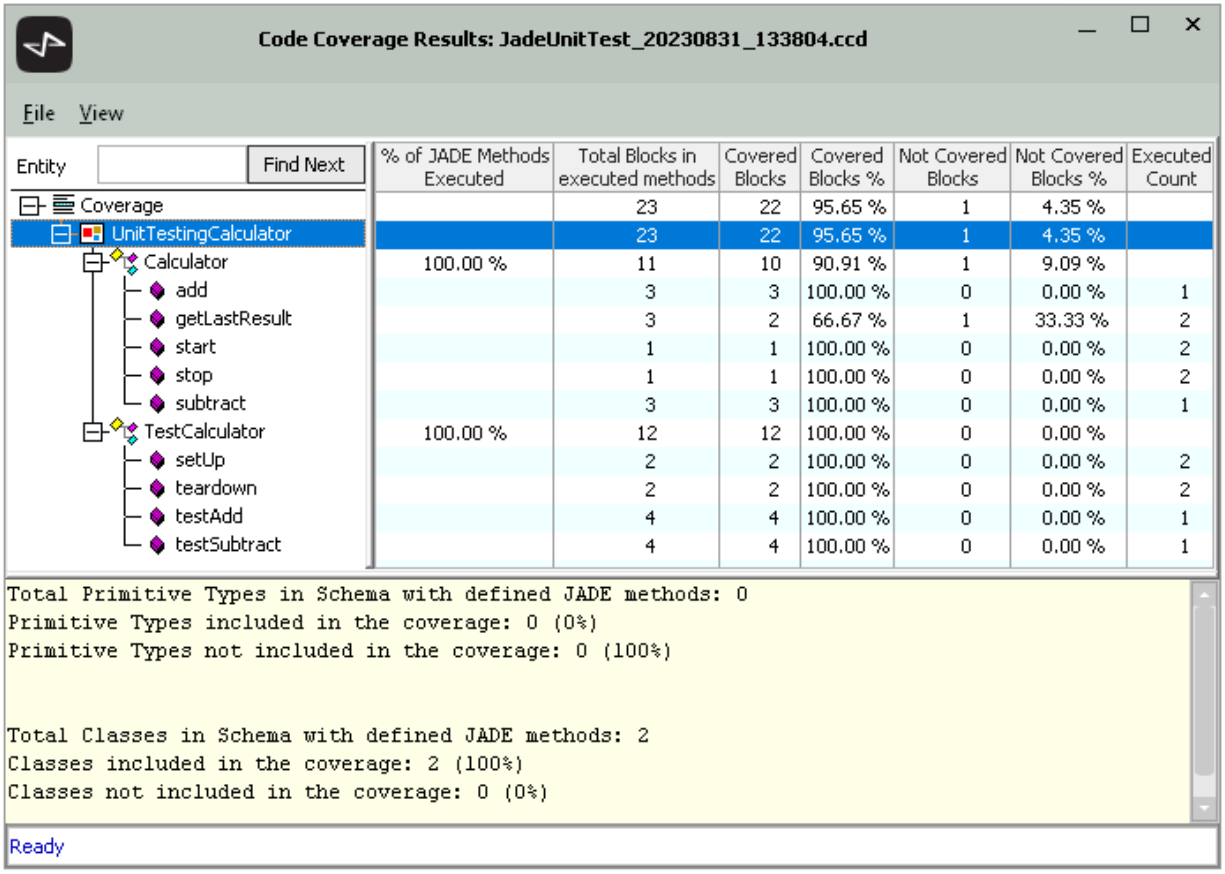
Code Coverage

Code coverage is a measure used in software engineering to describe the degree to which the system's source code has been executed. It is a useful measure to assure the quality of a set of tests, as opposed to directly reflecting the quality of the system under test.

To monitor code coverage during unit tests, select the **Code Coverage** command in File menu of the Unit Test Runner form.

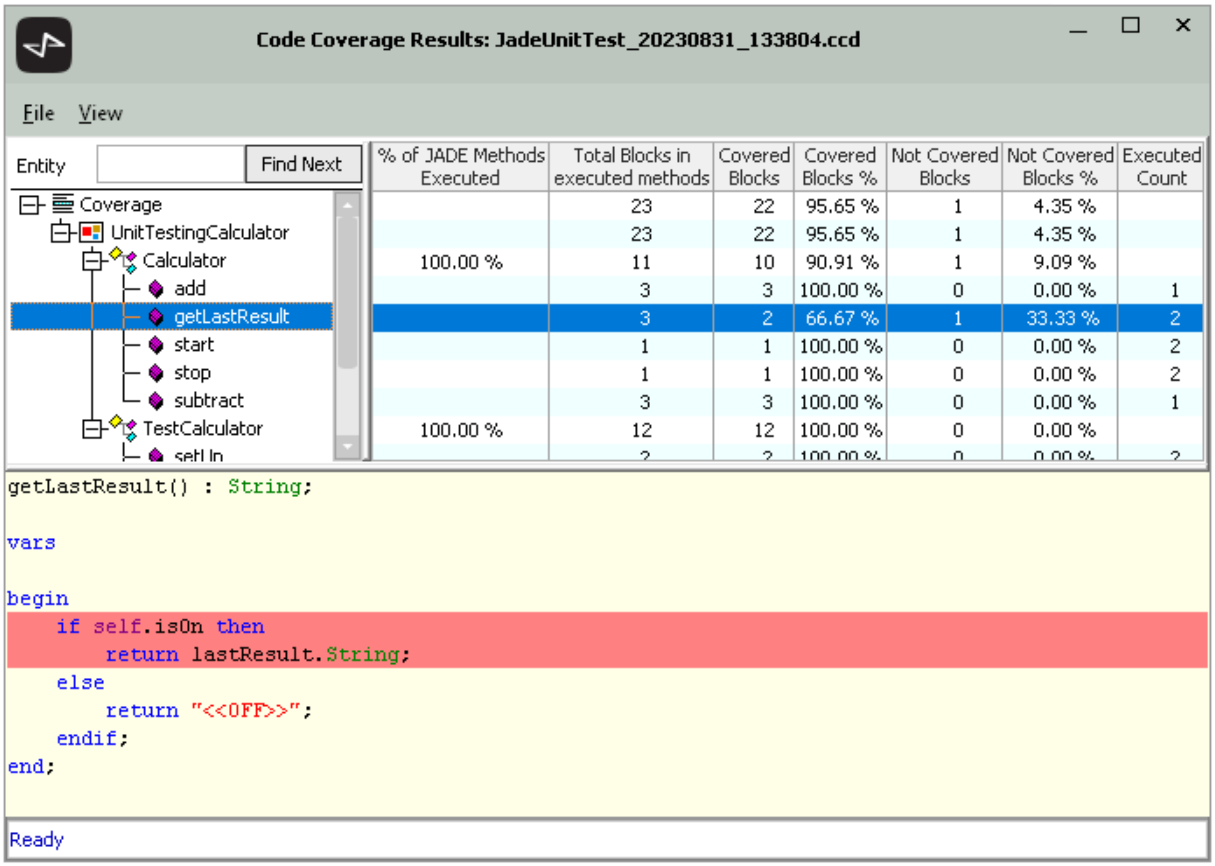
After running the unit tests for which you want to record code coverage, select the **View Code Coverage** command in File menu of the Unit Test Runner form.

The code coverage results are then displayed on a new form; that is, the Code Coverage Results Browser.



The Code Coverage Results Browser displays the number of Jade methods that were executed, and within those methods, the percentage of code blocks that were executed.

When selecting a method, the Code Coverage Results Browser displays the method code with the executed code highlighted in red.



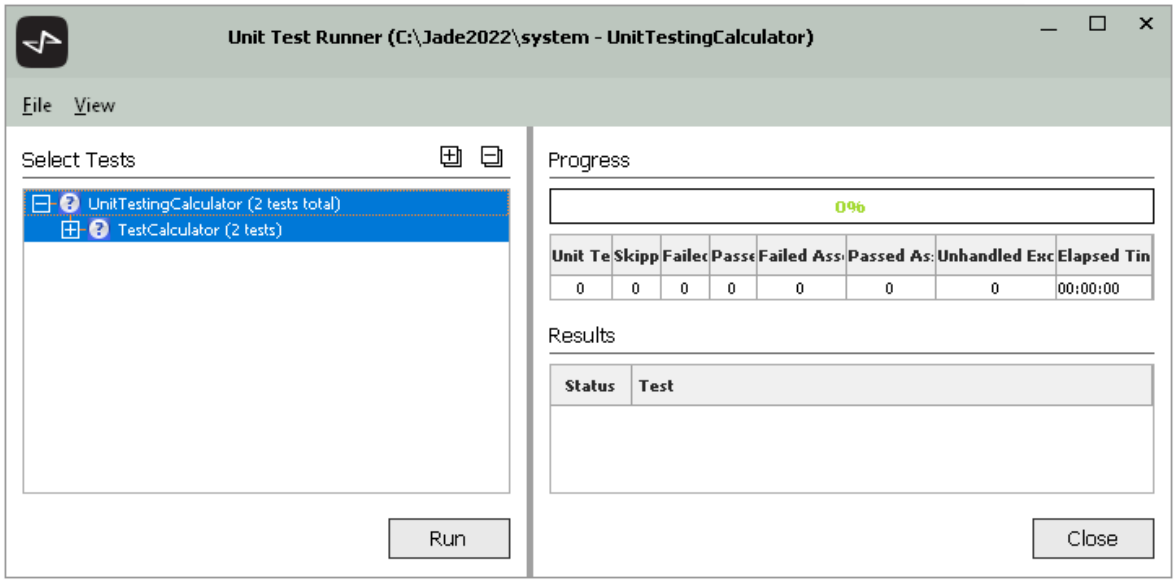
By using the code coverage information, you can identify areas of code that have not been tested by a specific tests suite. This enables you to focus new tests on code that has not yet been executed by any test.

Alternatively, if the code coverage is very high, it can provide evidence of how thoroughly the test suite exercises the code, which is one measure of the quality of a set of tests.

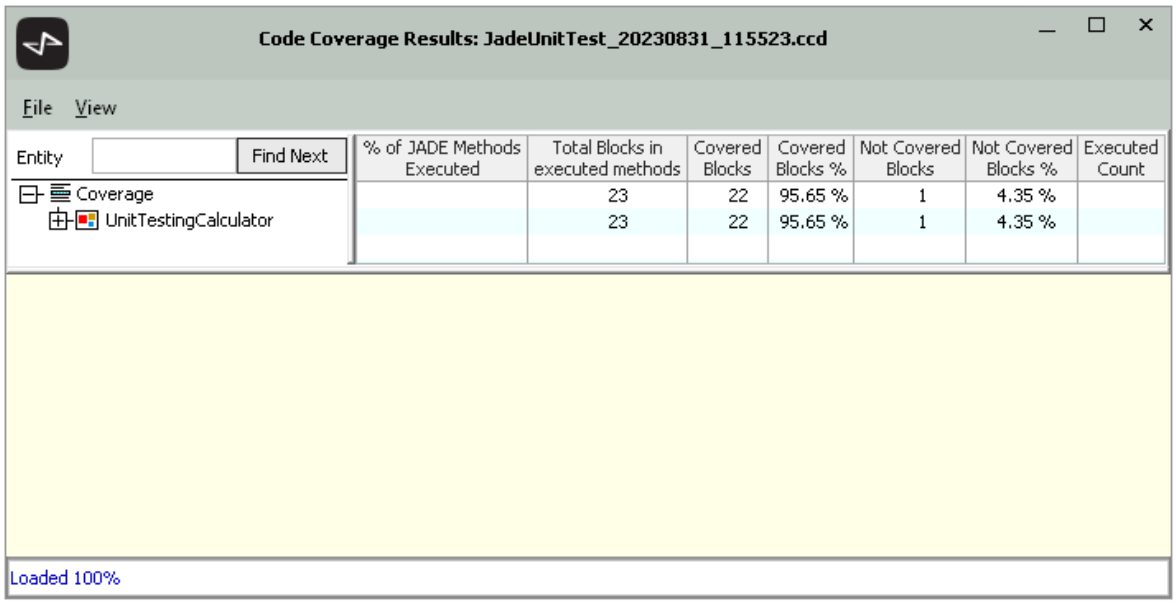
Exercise 5 – Viewing Code Coverage Results

In this exercise, use the code coverage functionality to find untested code.

1. Select the **UnitTestingCalculator** class from the Schema Browser and then press F9.

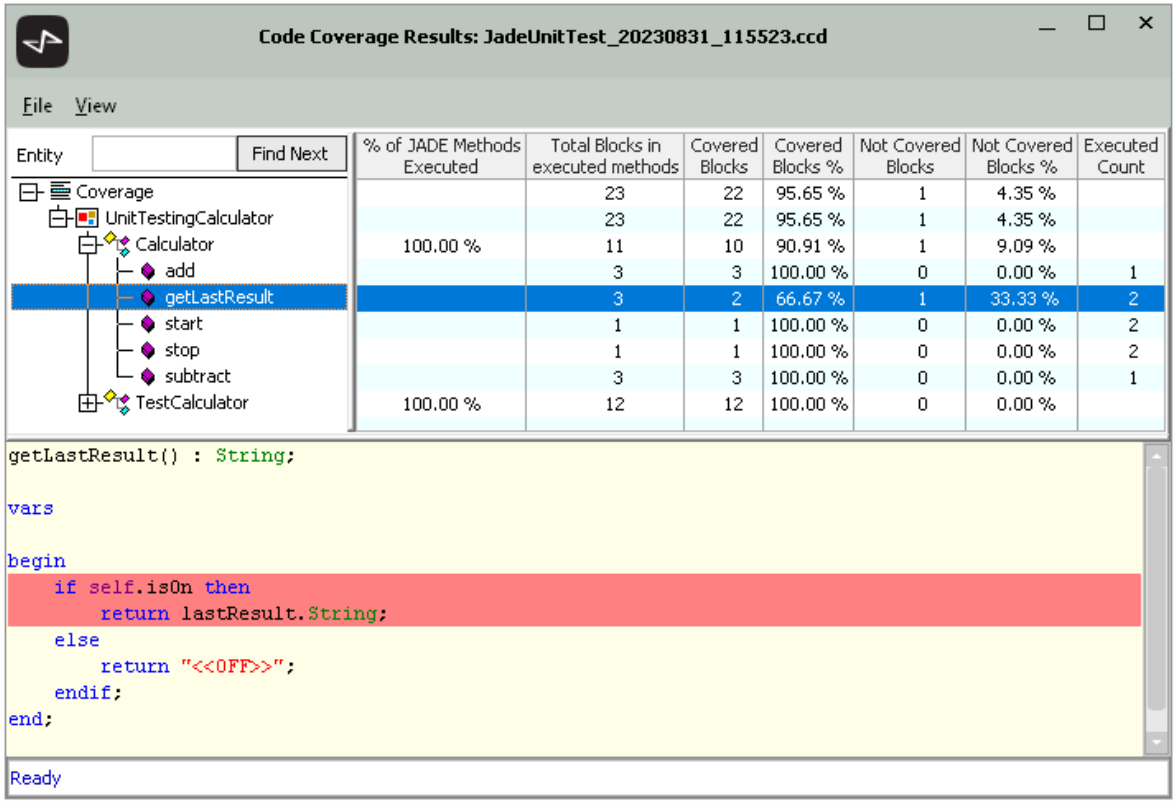


2. Select the **Code Coverage** command from the File menu.
3. Click the **Run** button, to run the tests.
4. When the tests have completed, select the **View Code Coverage** command from the File menu.



5. You will see that 95.65 percent of blocks have been covered.

Expand the **Calculator** class in the **UnitTestingCalculator** schema and then locate the untested code block.



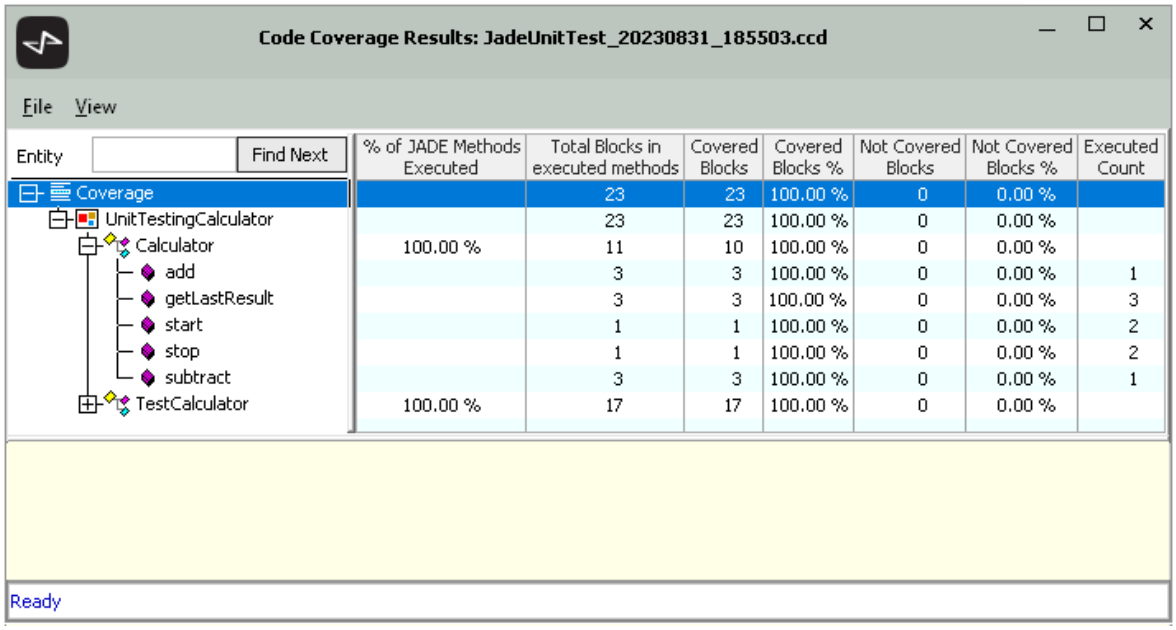
6. Write a new unit test to cover the situation in which you try to use the calculator while it is off.

Note While 100 percent code coverage can be a good goal for smaller projects, it can often be impractical in larger systems. It is only one of many measures of code quality.

Exercise 6 – Saving Code Coverage Results

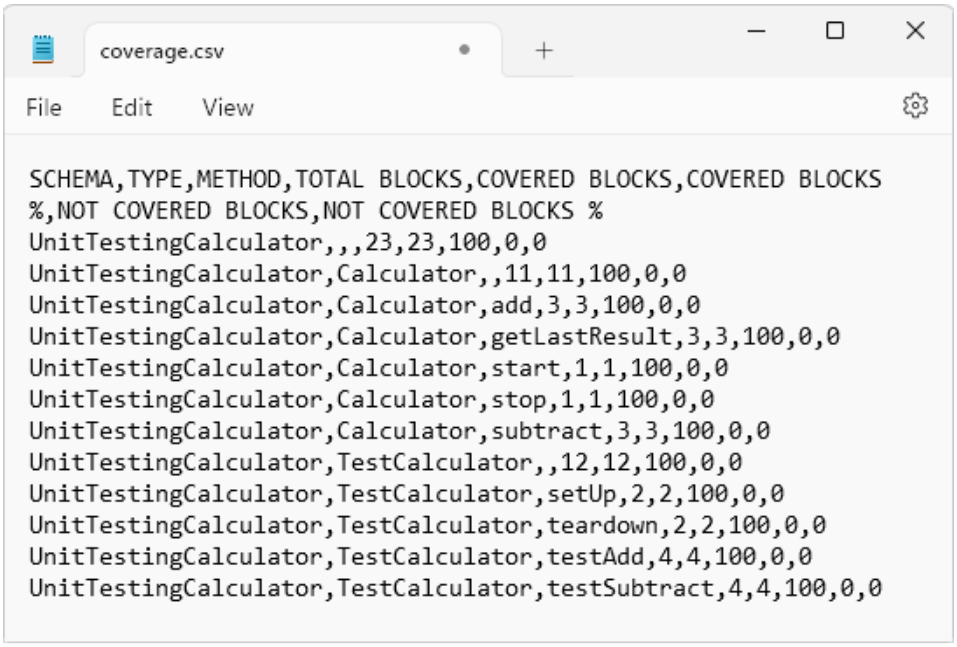
In this exercise, save code coverage results to a Comma-Separated Values (.**csv**) file.

1. Open the Code Coverage Results Browser as you did in the previous exercise.



2. Select the **Save As CSV** command from the File menu. A common dialog is then displayed, to enable you to select the folder to which to save the code coverage results. (The Desktop or the Documents folder are good options.)

3. Open the CSV file from the File Explorer, to view its contents.



Version Control

The Jade Platform provides a variety of tools to assist with version control of Jade schemas.

- Deltas allow for methods to be checked in and out, enabling developers to lock methods while they are working on them.
- Patch versioning allows the setting of a patch version number that records all changes made to schema entities until a new patch version number is set.
- The Jade Git client allows for the use of the Git version control system to maintain a history of changes.

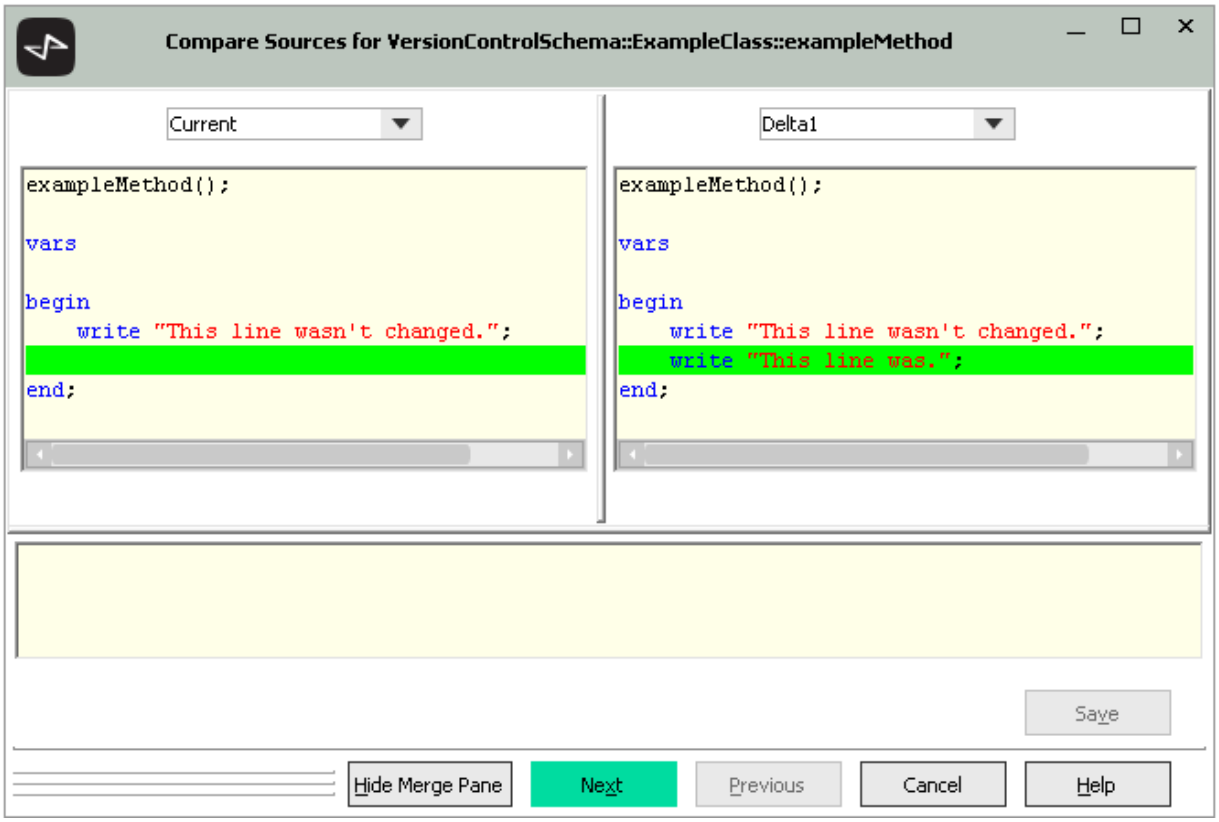
Deltas

Deltas are a tool used for managing changes to a Jade method so that when a method is checked out, any modifications can be reviewed. After review, the method can be accepted (with **Check In**) or reverted (with **Undo Checkout**).

A delta is essentially a list of changes to a method, and as such, is typically used to review changes during a development process; for example, ensuring that the scope of a change is limited to the intended scope so that no extraneous changes are unintentionally made during a method change.

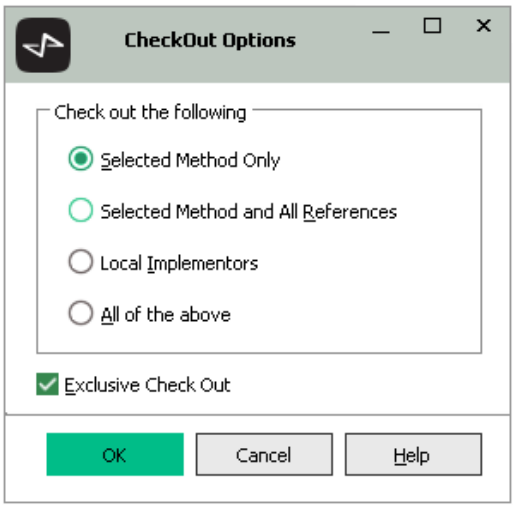
The Compare Sources window provides this functionality, serving as an inbuilt diff tool and highlighting the parts of the method that have changed since checking out the method.

The following is an example of the Compare Sources window.



Note A diff tool shows the differences between two text files. This becomes especially useful when there is a large amount of text and only a very small amount of it has been changed, as well as for verifying that no code has been changed unintentionally.

Deltas in Jade also provide the ability to lock methods while they are checked out. Checking the **Exclusive Check Out** check box when checking out a method prevents any other user from modifying the method until the method is checked back in or reverted.

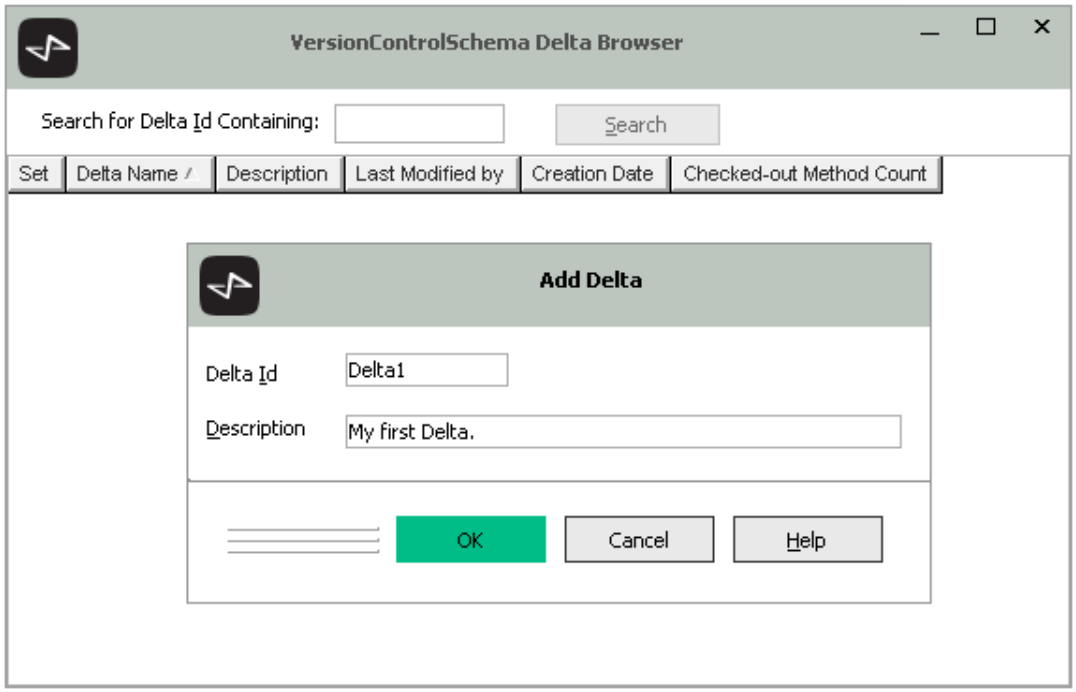


Exercise 1 – Creating a Delta

In this exercise, create a delta and check out a method into it.

1. Create a new schema called **VersionControlSchema**.
2. Add a class called **ExampleClass** to the **VersionControlSchema**.
3. Add a method called **exampleMethod** to the **ExampleClass**. For now, you don't need to code anything in it.
4. Open the Delta Browser by selecting the **Deltas** command from the Browse menu or pressing Ctrl+D.

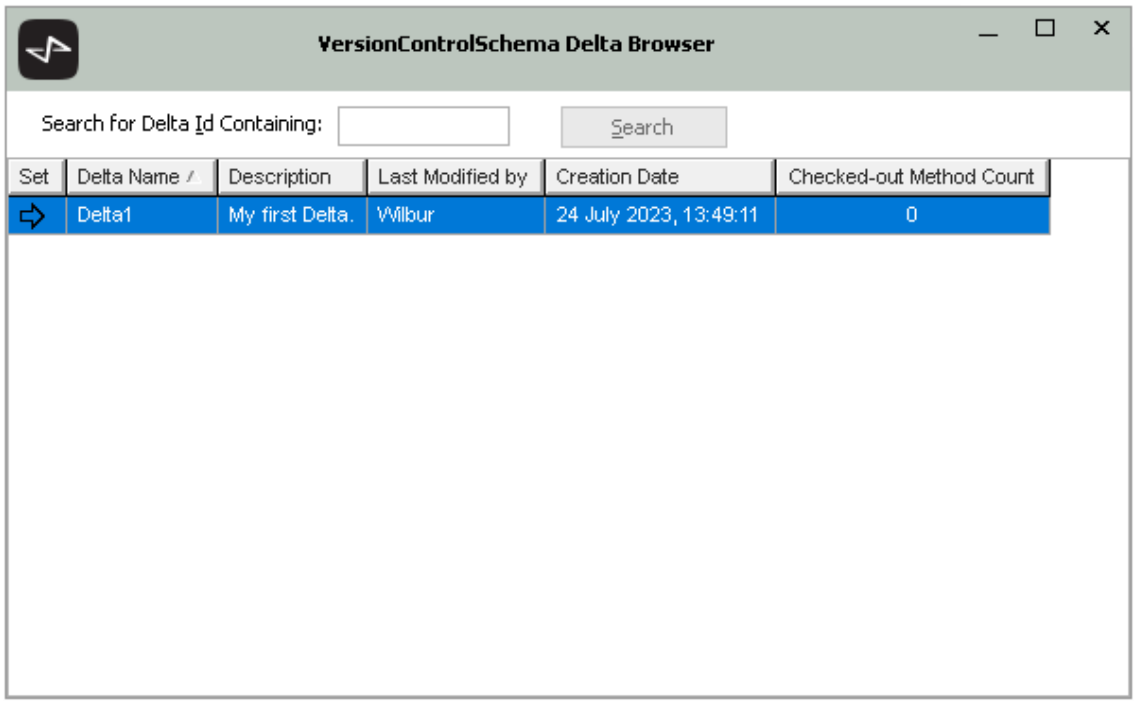
5. Add a delta by selecting the **Add** command from the Delta menu or right-clicking in the Delta Browser and selecting **Add**.



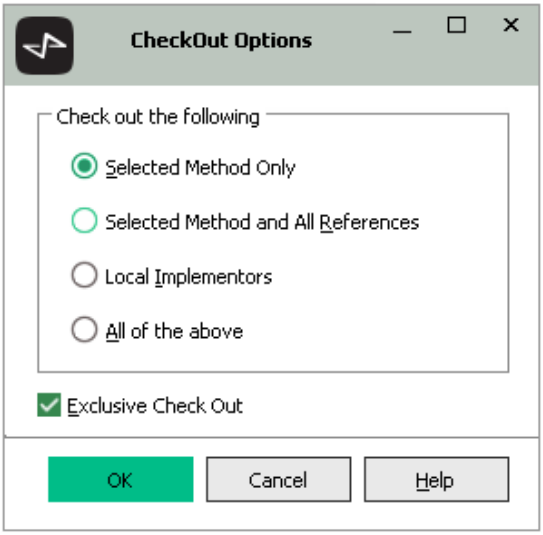
Enter **Delta1** in the **Delta Id** text box and **My first Delta** in the **Description** text box.

6. The delta is displayed in the list of deltas. Right-click on it and the select **Set**.

An arrow is then displayed in the **Set** column at the left of the delta name.



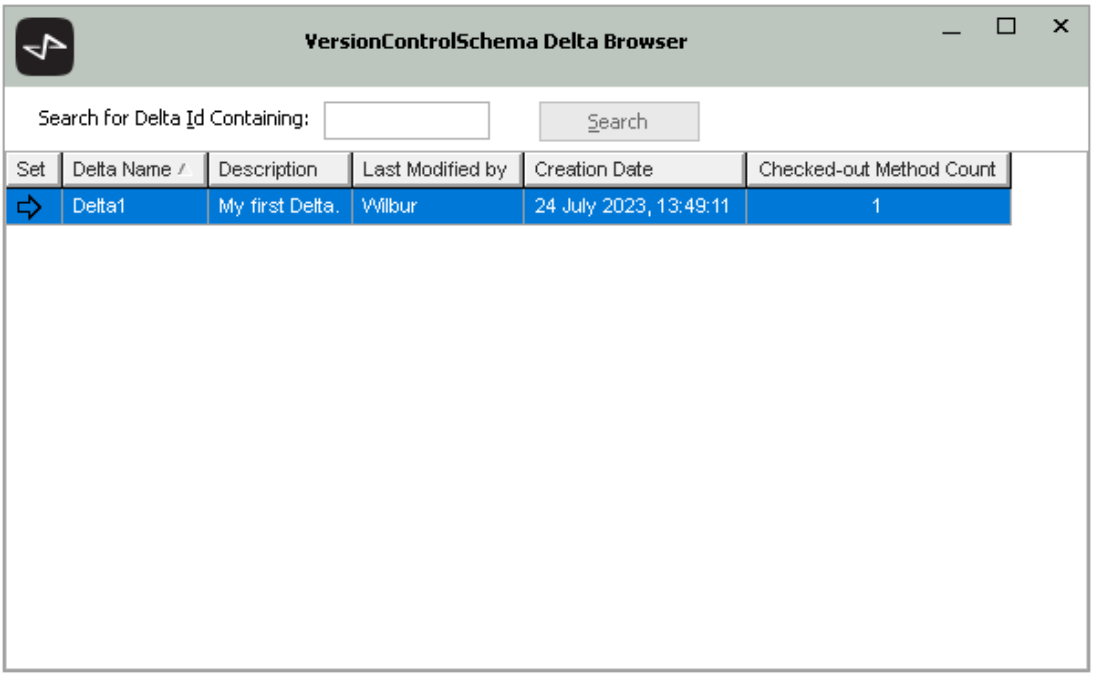
7. Navigate to the **exampleMethod**, right-click on it, and then select **Check Out** from the Delta menu.



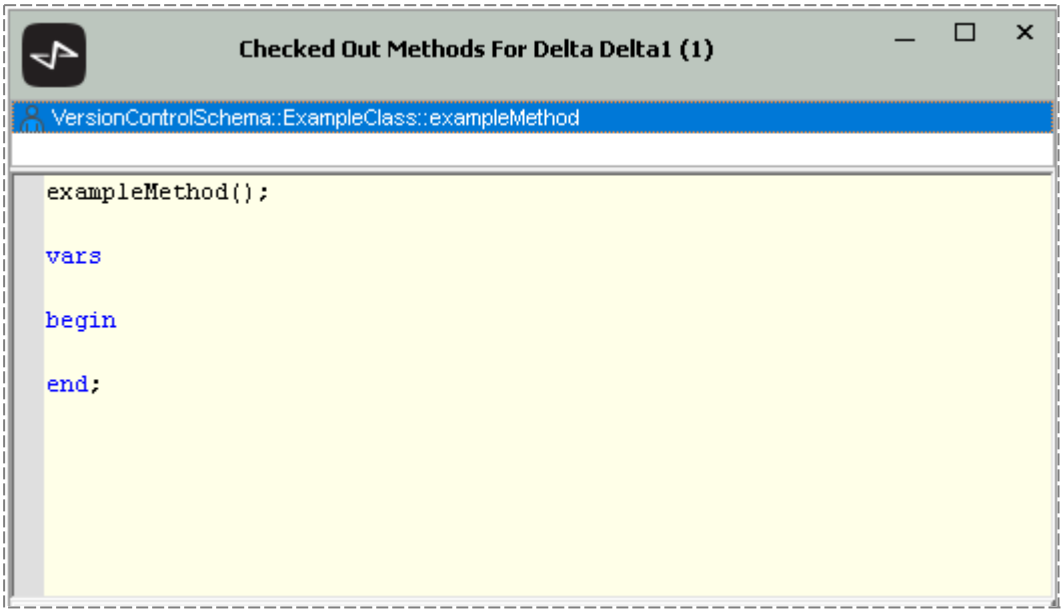
8. Select the **Selected Method Only** option button and check the **Exclusive Check Out** check box, and then press **OK**.

Note *Exclusive check out* means that the method will be locked, and other users cannot modify it until you check it back in or you undo the checkout action.

9. Press Ctrl+D to reopen the Delta Browser. You will see that the checked-out method count is now set to **1**.



10. Right-click the delta and then select **View Methods**. You will see a methods list containing all checked out methods. (Currently there is just the one.)



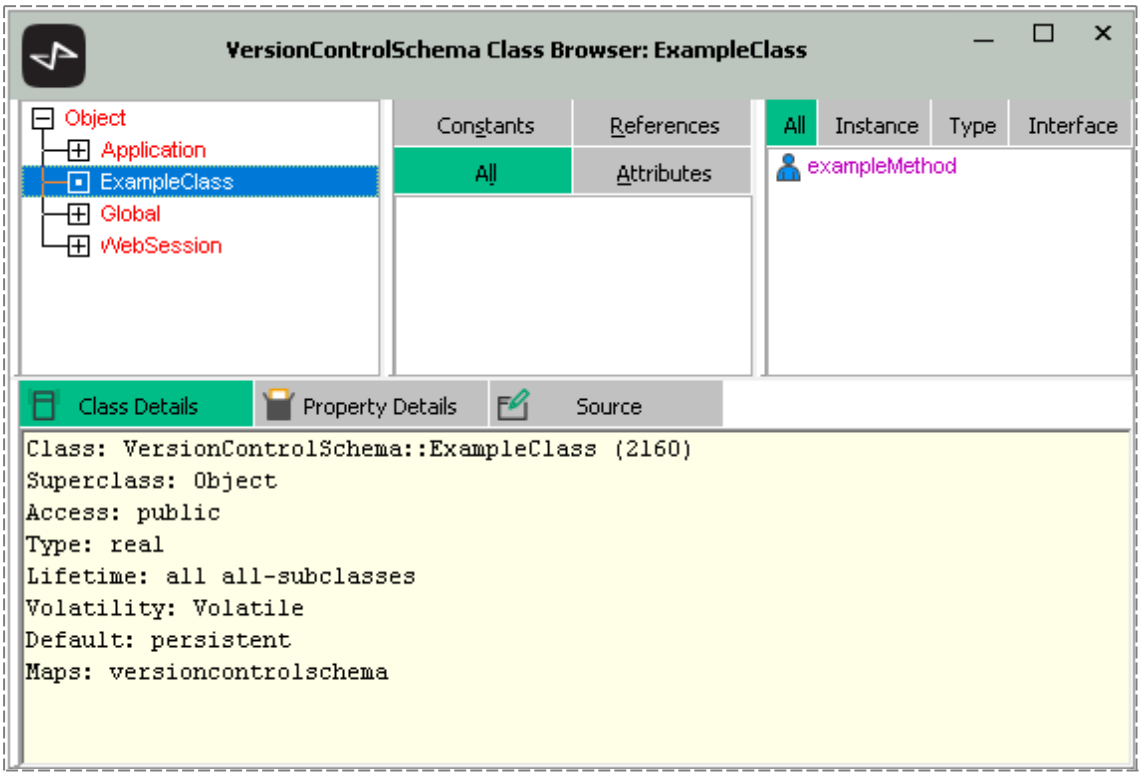
Exercise 2 – Checking in from a Delta

Note This exercise will be easiest if you open your Jade database in multiuser mode; that is, with a database server and two standard (fat) clients.

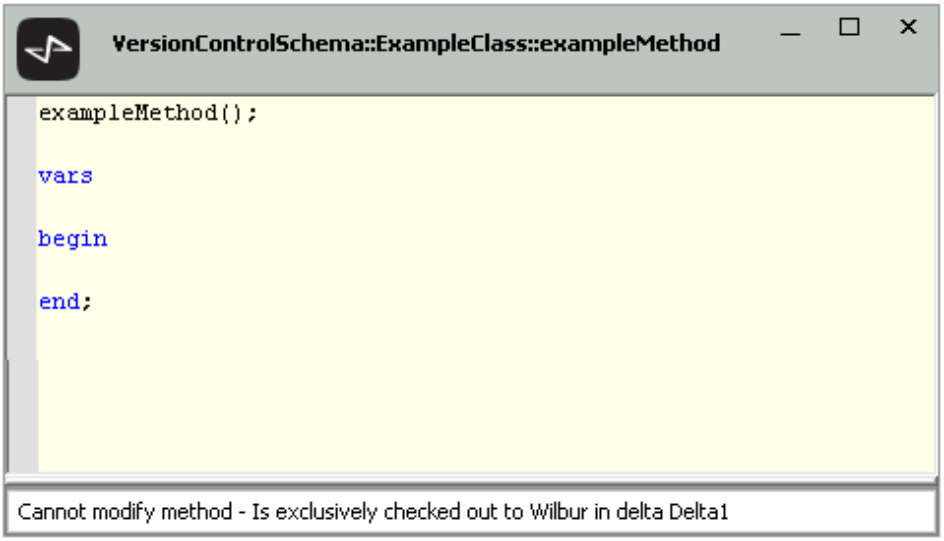
In this exercise, view the checked out method from another user's perspective, modify it, and then check the method back in from the delta.

1. Log on to your Jade database with a different name from that with which you created the delta. (If you are running in single user mode, close and reopen the database or start a standard client if you have a database server and a standard client running.)

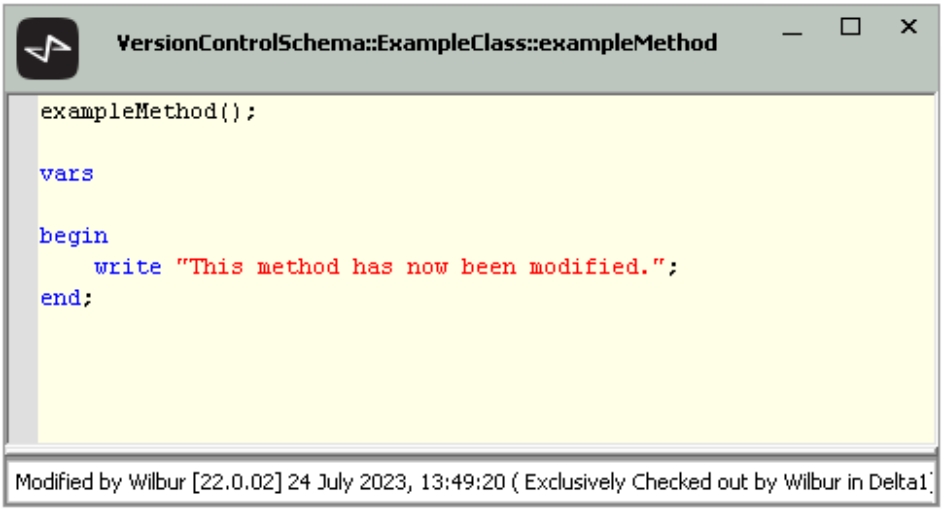
2. Navigate to the **exampleMethod** method. You will see that it is a violet color instead of the usual black.



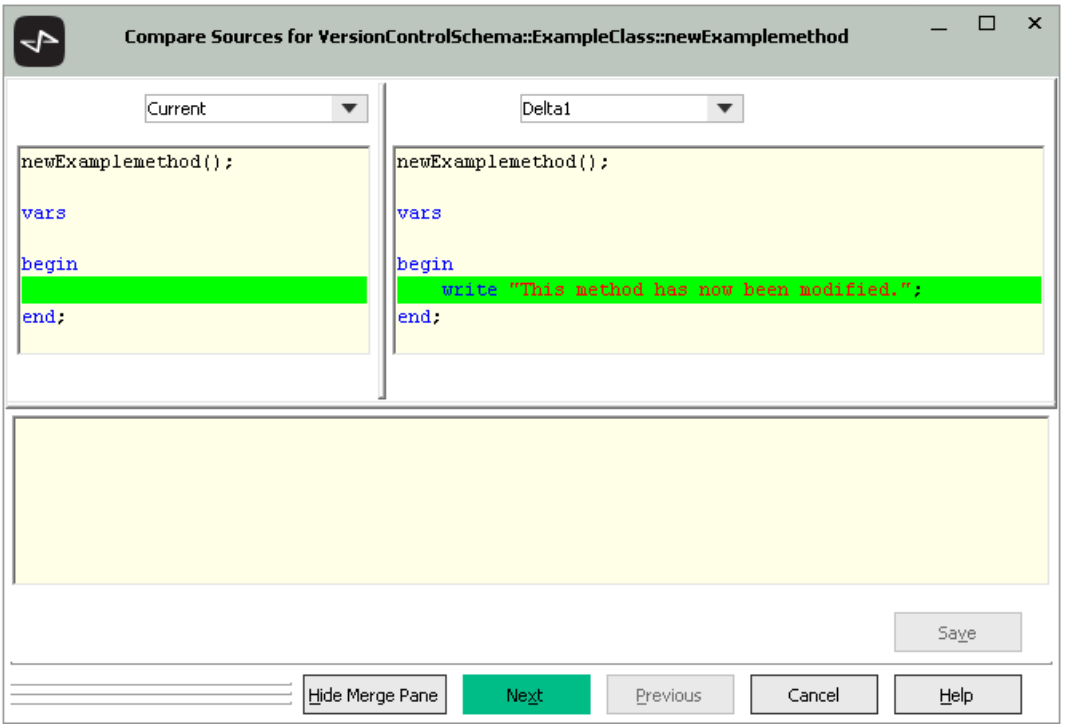
3. Attempt to modify the method. You will see that you cannot enter any text into the method and that the status line reads **Cannot modify method**.



4. Open the database with your original user name and then modify the **exampleMethod** as follows.



5. Right-click the **exampleMethod** in the Class Browser and then select the **Compare** command from the Delta menu.



The Compare Sources window shows the change or changes made to the method.

Tip This tool is useful when making changes to a large method, to ensure that you have changed only what you intended to.

6. Close the window, right-click the **exampleMethod** in the Class Browser, and then select the **Check in** command from the Delta menu. (Click **Yes** in the message box prompting you to confirm that you want to check in the method.)

7. Now that the method is checked back in, it is no longer locked. Open it in your other Jade client node to verify this.

Patch Versioning

Patch versioning allows for the setting of patch numbers for a Jade database and assigning a group of schema changes to that patch.

Bundling sets of changes into patches is a common software development technique, allowing for precision in the scope of changes to be deployed.

Note Patch versioning is completely distinct from deltas, and you can use either without the other. That said, a version control strategy often incorporates both.

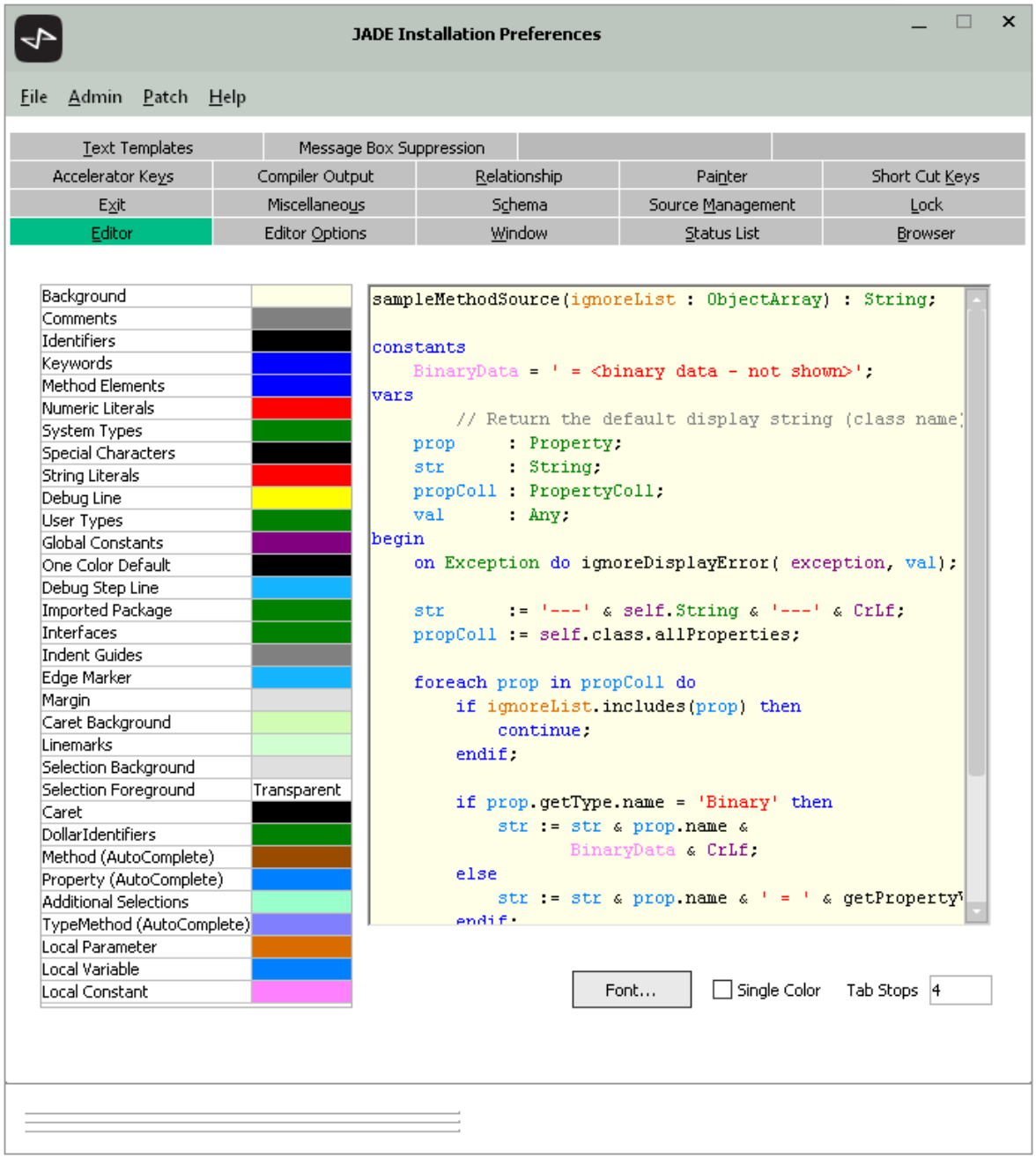
One advantage of employing a patch versioning strategy is that it will cause a history of changes to be generated over the development of a solution. This history can be viewed from the Jade Platform development environment using the Compare Sources window, allowing you to identify what changed, who changed it, and when it changed. This can be useful if a method's behavior changes unexpectedly.

Exercise 3 – Creating a Patch

In this exercise, create a patch for your database and add a number of changes to it.

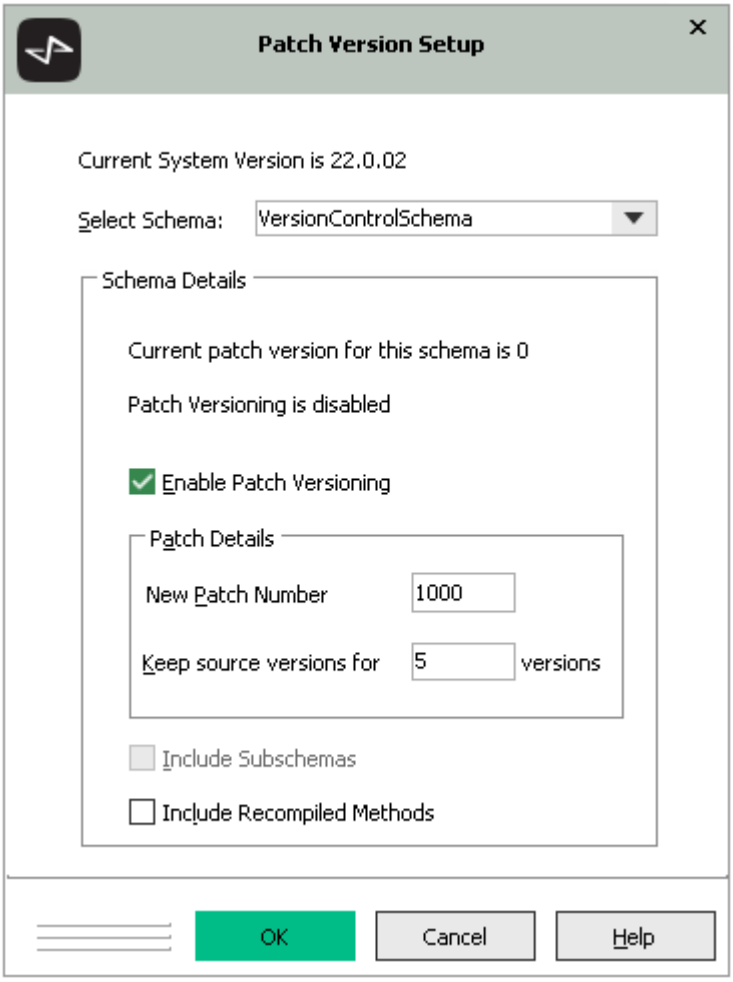
1. Reopen your Jade database, but select the **Administration** option instead of the default **Browse Classes** option on the Jade Platform logon form.

2. The Jade Installation Preferences dialog is then displayed. This is similar to the User Preferences dialog except that it has a **Painter** sheet and menus - specifically the *Patch* menu.



3. When you have enabled patch versioning if it was disabled, select the **Set Patch Number** command from the Patch menu.

Fill out the Patch Version Setup dialog that is then displayed as follows.

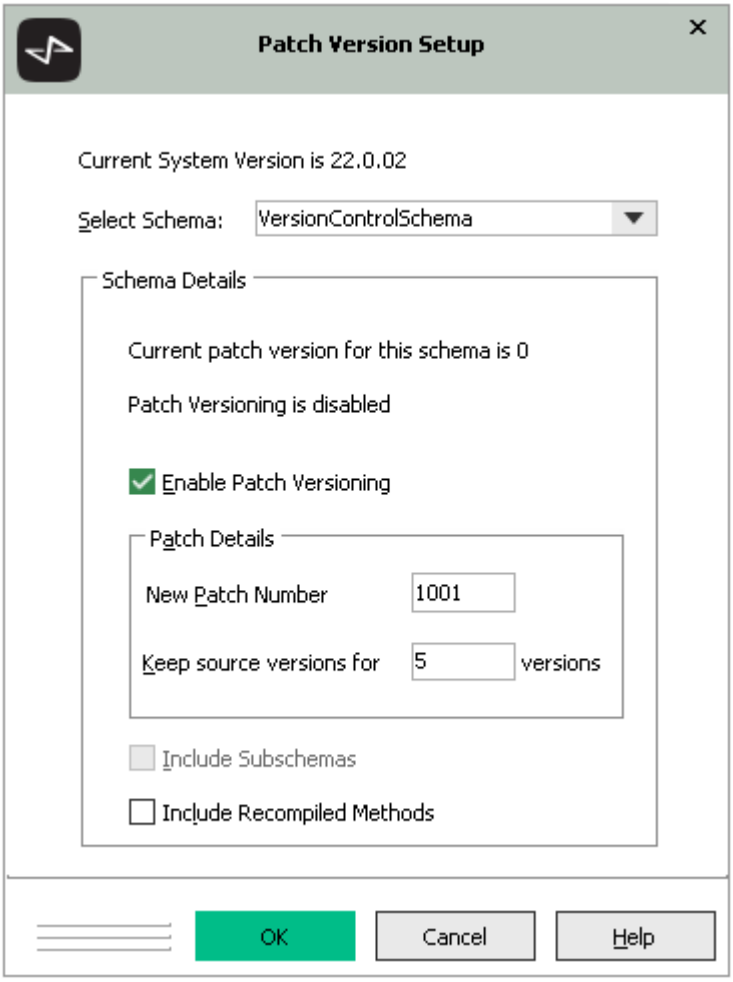


4. Click **OK**, then reopen the database in Browse Classes mode.
5. Navigate to the **exampleMethod** and then modify it as follows.

```
exampleMethod();

vars

begin
    write "This method will have a patch history.";
end;
```

6. Save the **exampleMethod**, then close the database and reopen it in **Administration** mode (as you did in step 1 of this exercise).

In the Patch Version Setup dialog, change the new patch number to **1001**.



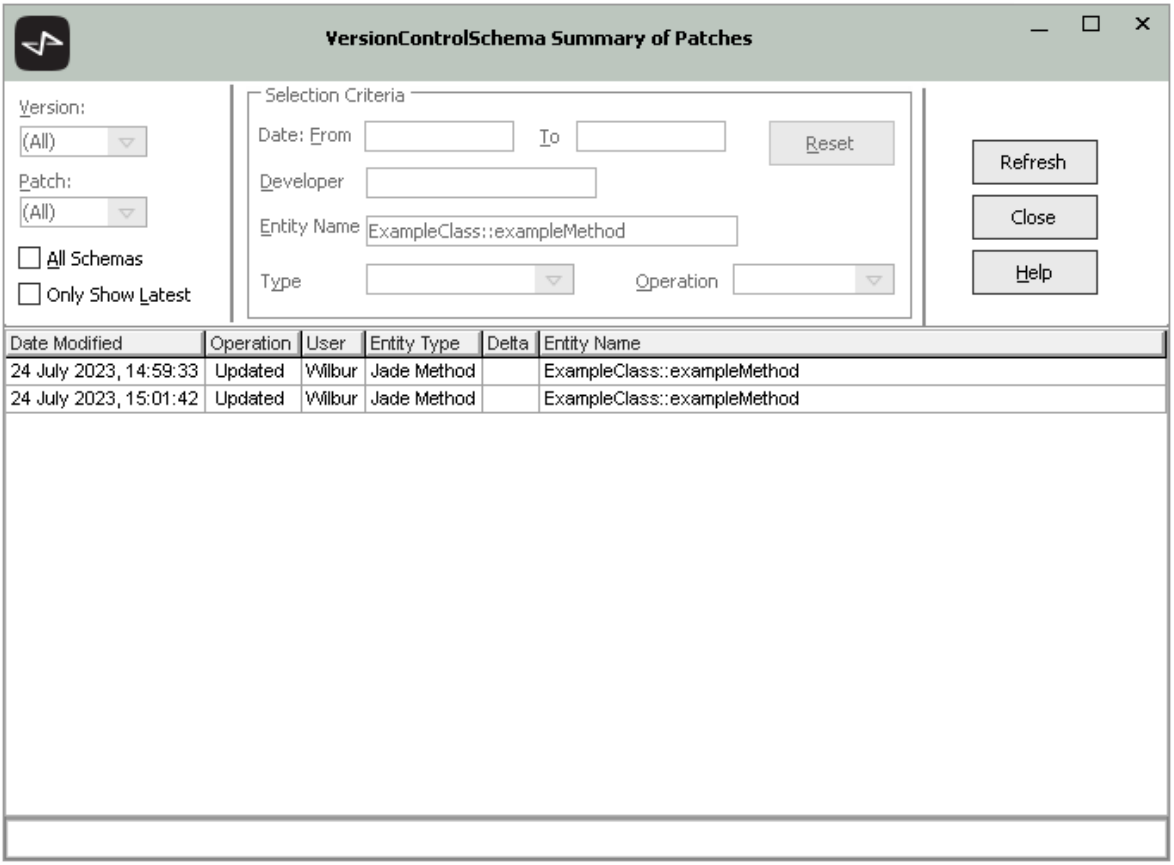
7. Click **OK**, then reopen the database in Browse Classes mode.
8. Navigate to the **exampleMethod** and modify it as follows.

```
exampleMethod();

vars

begin
    write "This is the latest patch.";
end;
```

Save your changes.

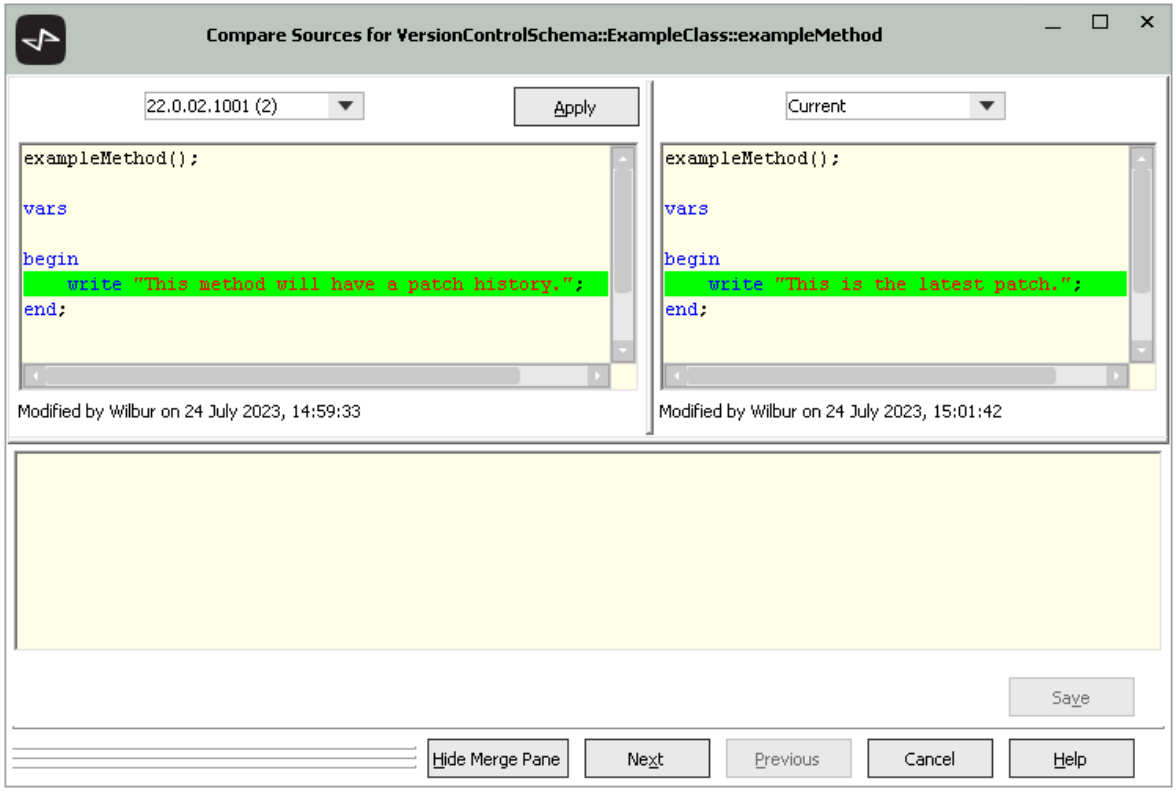
9. Right-click the **exampleMethod** then select **Show History**.

   The Summary of Patches form is then displayed.



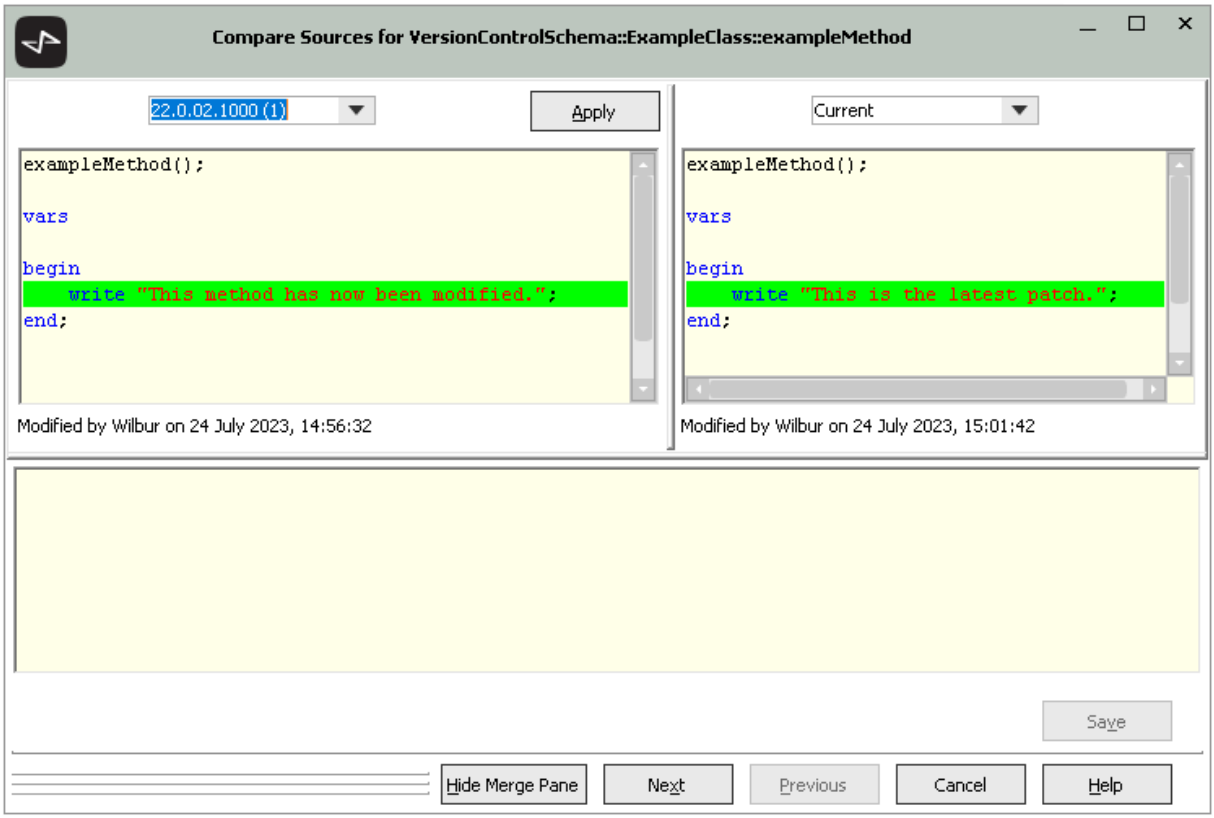
Note This feature is available only while patch versioning is enabled.

10. Double-click the second item in the list, to display the Compare Sources window.



The Compare Sources window shows the differences between the current version of the method and the version as it was when the patch was set, the user who modified the method, and a timestamp of when it was modified.

11. In the combo box for the code source on the left, select the **22.0.02.1000** patch.



You will see the differences between the first patch and the current version.

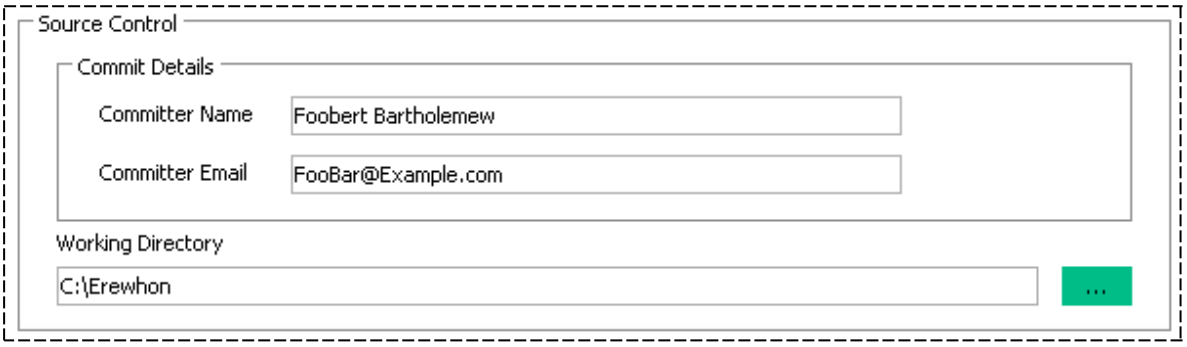
Regular patching allows for the maintenance of a history of patches, documenting how the software has changed over its lifetime. In addition, clicking **Apply** reverts the current source to the source of the selected patch.

Note The **22.0.02** prefix before the **1000** for the patch number is the version number of the Jade Platform itself; for example, **22.0.02** is the version number of the first service pack release of Jade 2022.

Exercise 4 – Cloning a Repository

In this exercise, clone the Erewhon repository from the Jade Software public GitHub and load it into your database.

1. Select the **Preferences** command from the Options menu.
2. On the **Source Management** sheet, fill out the Source Control group box as follows.

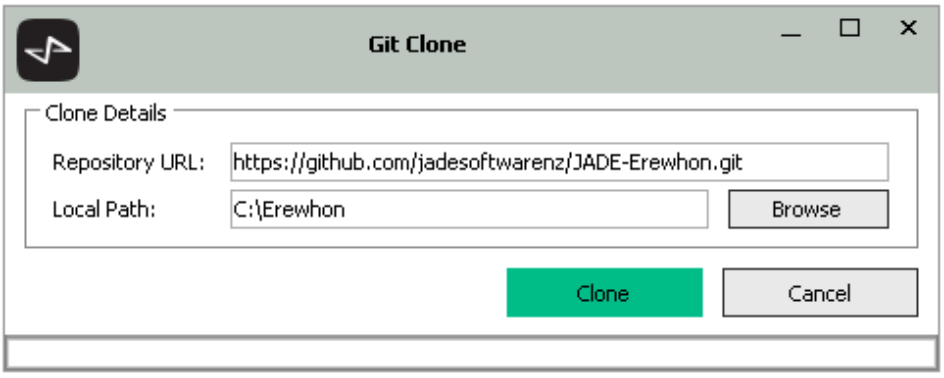


   - For the **Committer Name**, enter your name
   - For the **Committer Email**, enter your email address
   - For the **Working Directory**, select **C:/Erewhon**

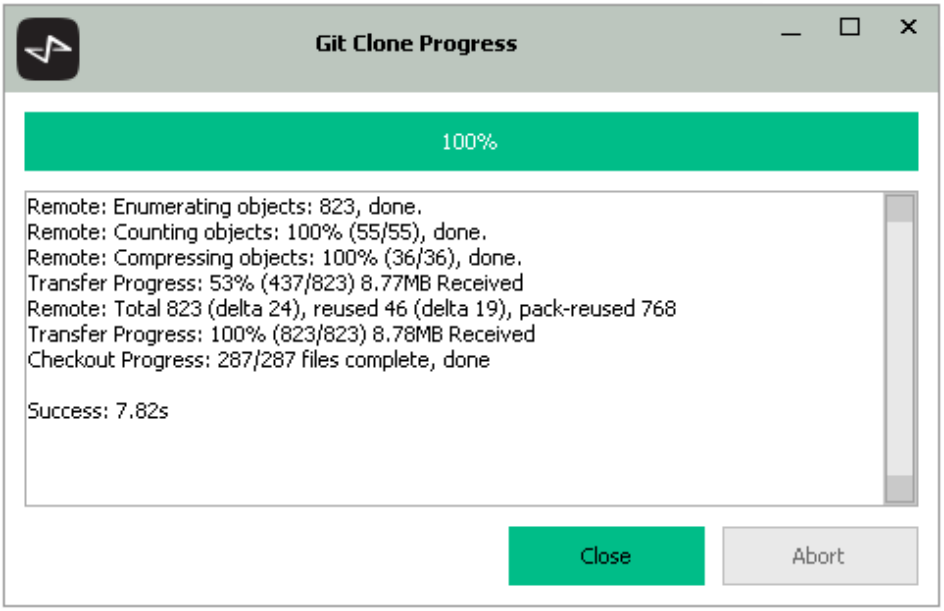
Note The name and email address must be set. However, they are not validated unless you are attempting to push to the remote repository, which we will not be doing in this exercise.

3. Navigate to the **C:/** drive in your file system and add a new directory of **C:/Erewhon**. This will be the directory of your local repository.
4. Select the **Clone** command from the Git Source Control Client submenu of the Browse menu.

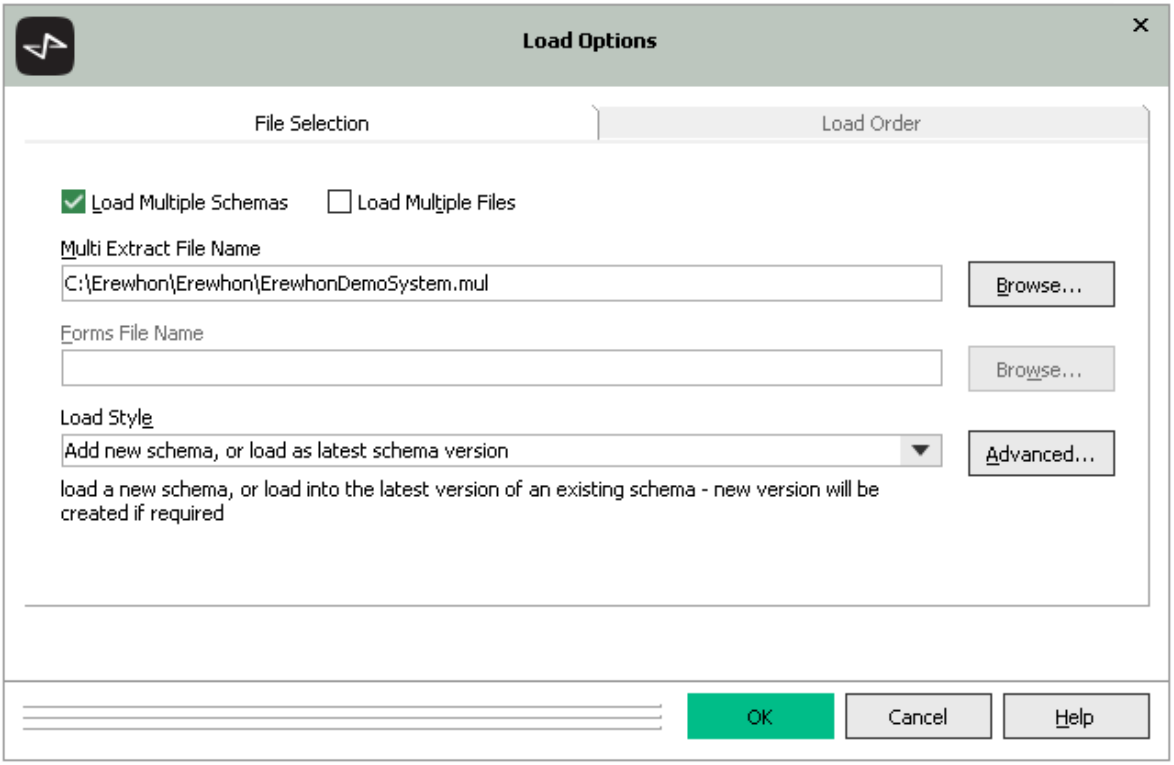
   In the Git Clone dialog that is then displayed, enter the following.



Click Clone. The Git Clone Progress dialog is then displayed.



5. Navigate to the **C:\Erewhon** folder. You will see that the Erewhon files have been cloned into the directory.
6. Select the **Load** command from the Schema menu.
7. On the Load Options dialog, check the **Load Multiple Schemas** check box, click **Browse** at the right of **Multi Extract File Name**, and then select **ErewhonDemoSystem.mul** in the **C:\Erewhon\Erewhon** directory.



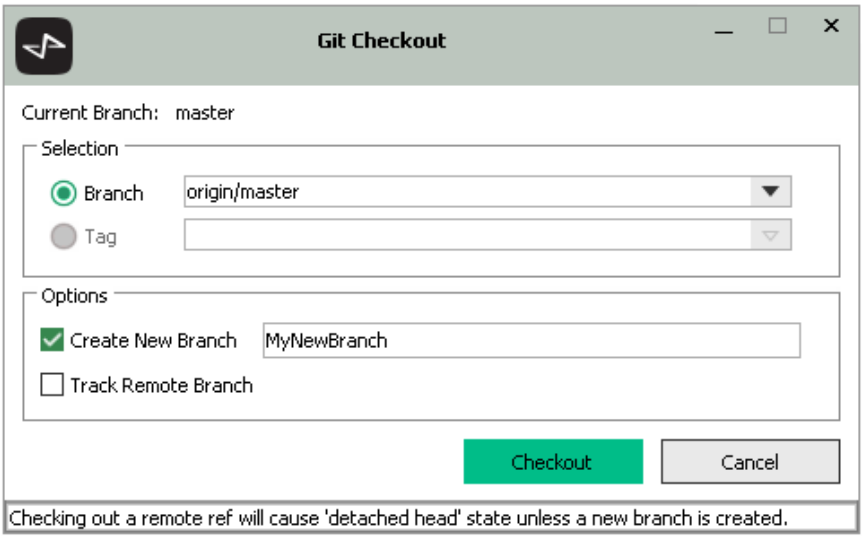
8. Click **OK**.

The Erewhon schemas are large, so it may take around a minute to load, at which point the following message box is displayed.

Exercise 5 – Checking Out a Branch in Git

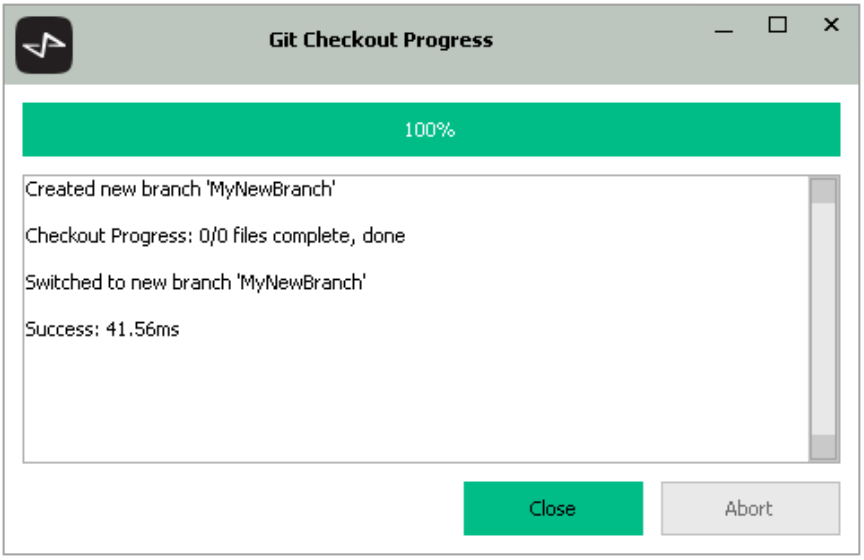
In this exercise, check out a new local branch, modify this branch, and then merge it back into the main branch.

1. Select the **Checkout** command from the Git Source Control Client submenu of the Browse menu.

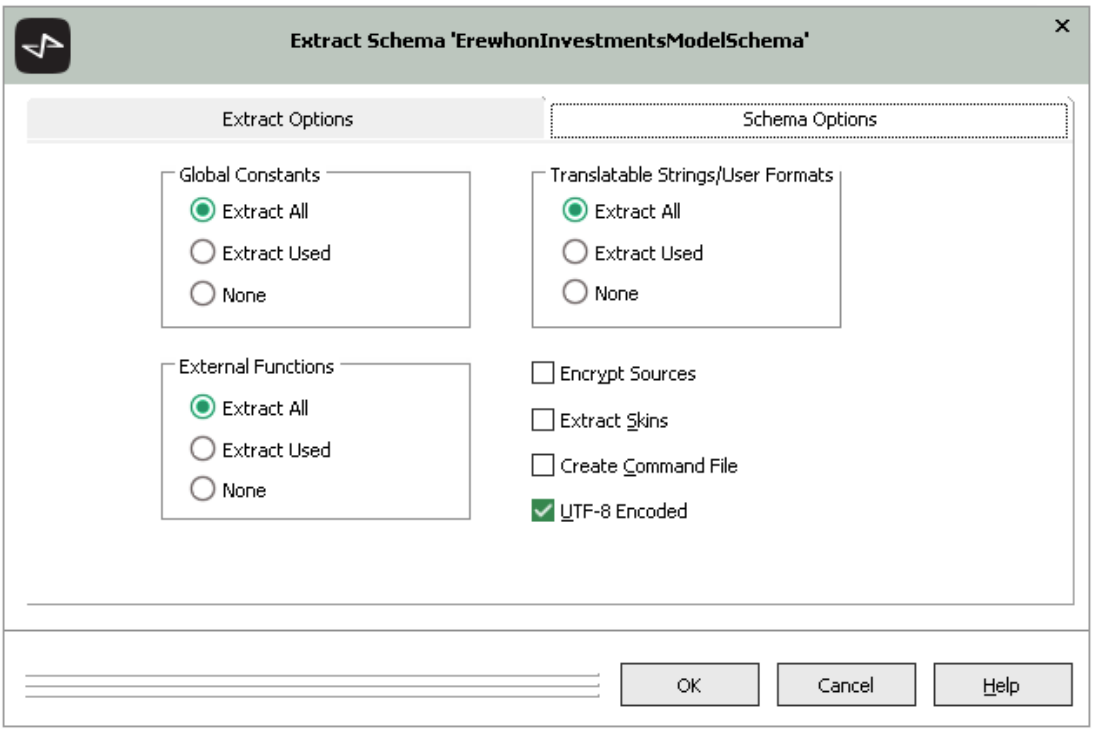


- Uncheck the **Track Remote Branch** check box
- Enter **MyNewBranch** in the **Create New Branch** text box
- Click **Checkout**

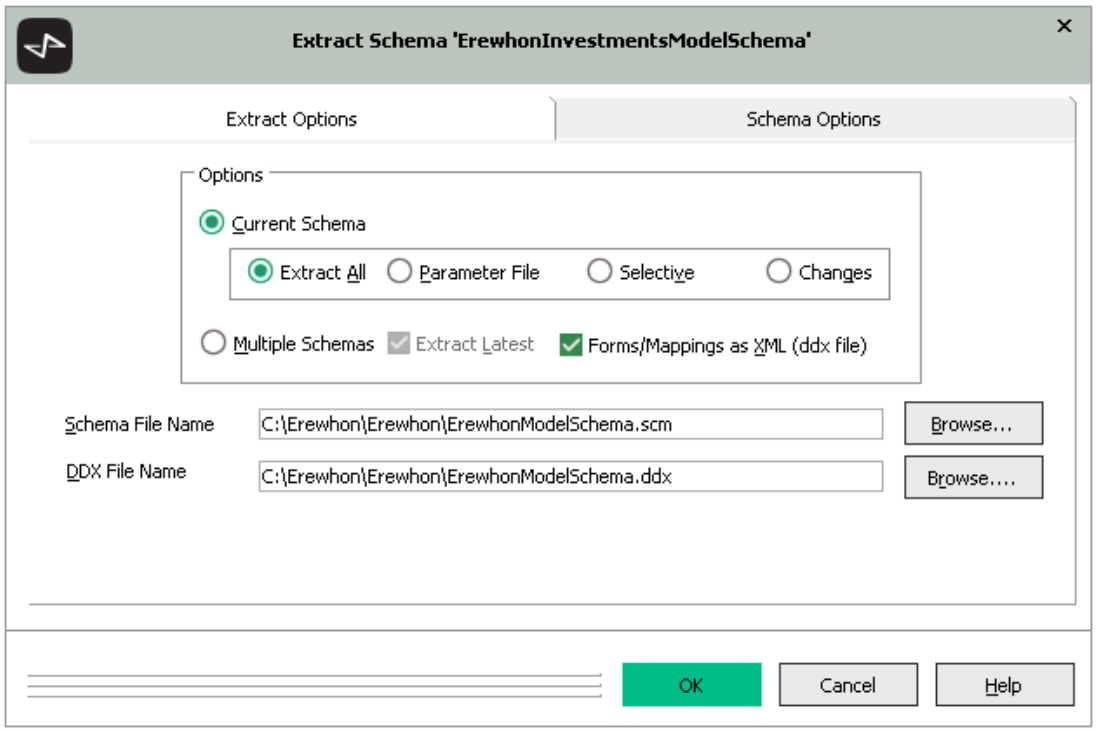
The following confirmation message box is displayed upon success.



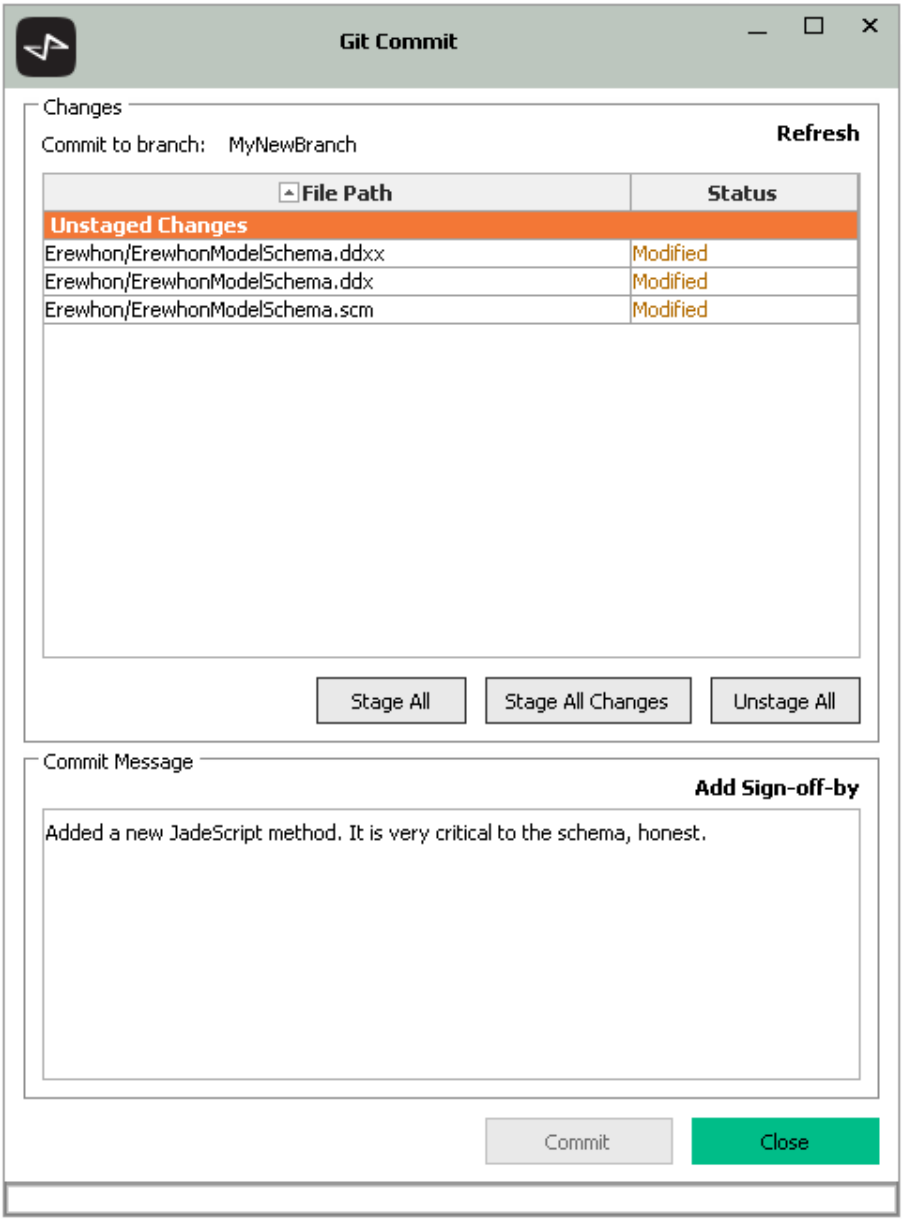
2. In the **ErewhonModelSchema**, create a **JadeScript** class method called **newMethod**. This method does not need any code, as it is needed only to make an arbitrary change to the schema.
3. Right-click **ErewhonModelSchema** in the Schema Browser and then select **Extract**.
4. On the **Schema Options** sheet, uncheck the **Create Command File** check box.



5. On the **Extract Options** sheet, check the **Forms/Mappings as XML (ddx file)** check box, click **Browse** at the right of the **Schema File Name** text box, and then select **C:\Erewhon\Erewhon** as the target directory. Click **OK**.



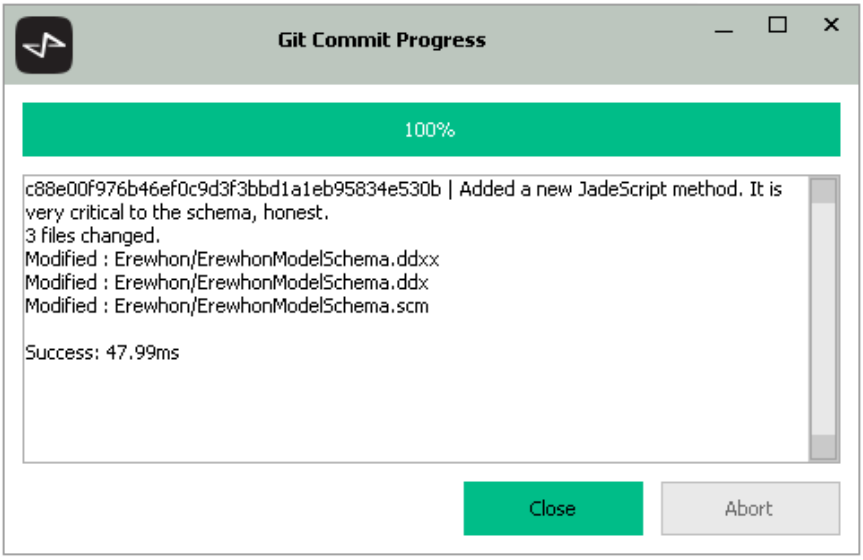
6. Select the **Commit** command from the Git Source Control Client submenu of the Browse menu.



The Git Commit dialog displays the files that have changed since the last commit action and allows you to stage some or all changes and then commit them. Right-clicking any of the unstaged changes allows you to stage the specific file, and clicking **Stage All** or **Stage All Changes** selects all files for the commit. When you are happy with the files to be committed, enter a commit message describing what has changed and why. The **Commit** button is then enabled.

Note The difference between **Stage All** and **Stage All Changes** is that the **Stage All Changes** action stages only those files that already exist in the repository and have been changed, whereas **Stage All** includes files that are newly created and do not yet exist in the repository.

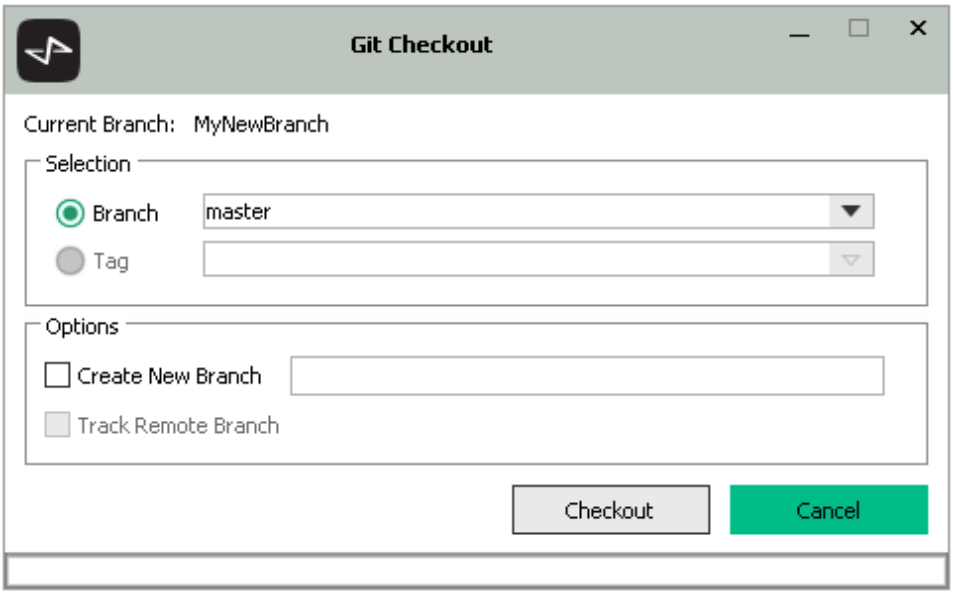
7. Stage all changes, enter any message you like as the commit message, then click **Commit**.



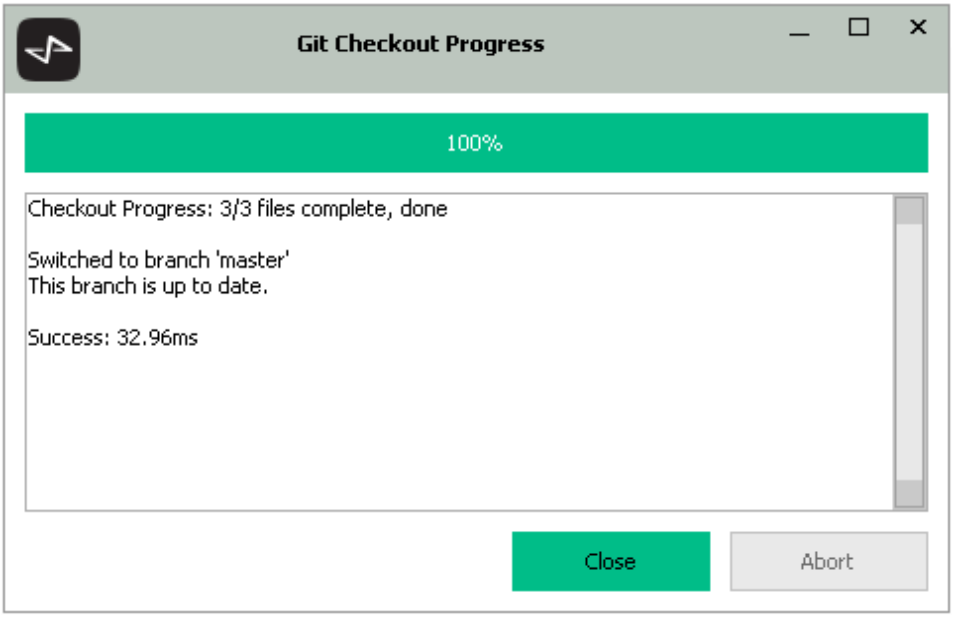
Exercise 6 – Back to the Master Branch

In this exercise, check out back to the master branch, effectively reverting the changes made while in the **MyNewBranch** branch.

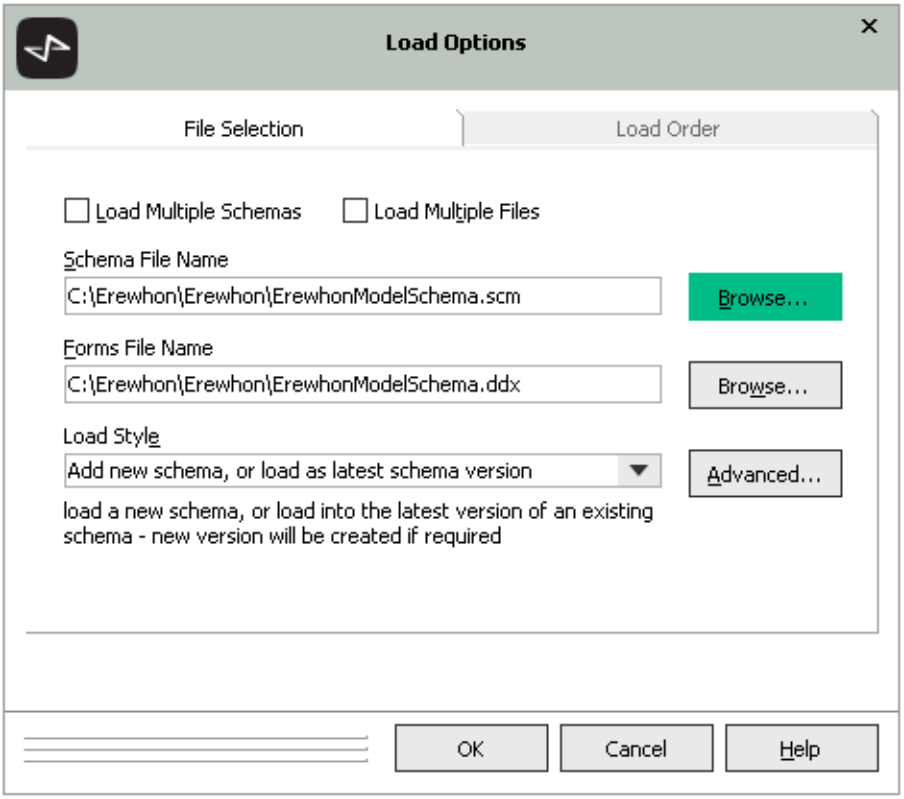
1. Select the **Checkout** command from the Git Source Control Client submenu of the Browse menu.



2. Click **Checkout**, to switch back to master.

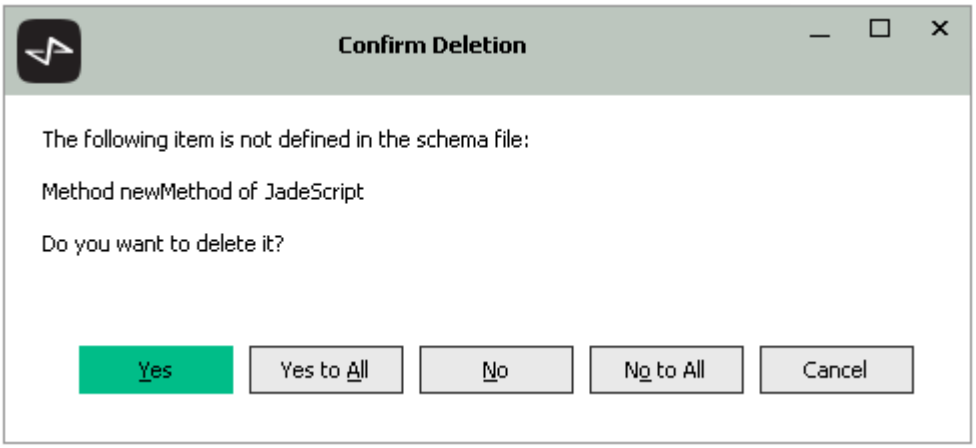


3. Select the **Load** command from the Schema menu.

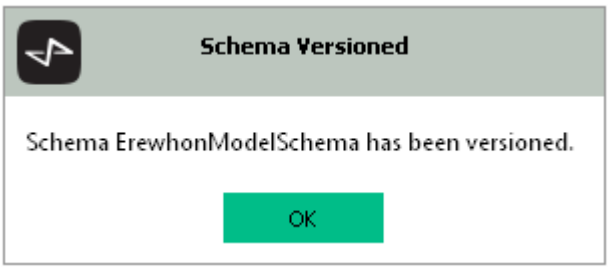


Click **Browse** at the right of the **Schema File Name** text box, navigate to the **ErewhonModelSchema.scm** file in **C:\Erewhon\Erewhon**, and then click **OK**.

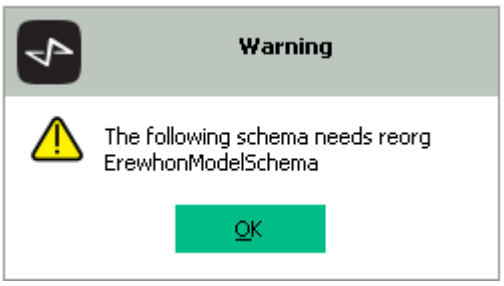
4. Part-way through the load process, the following message box is displayed. Click **Yes**, to remove the **JadeScript** class **newMethod** method from the schema.



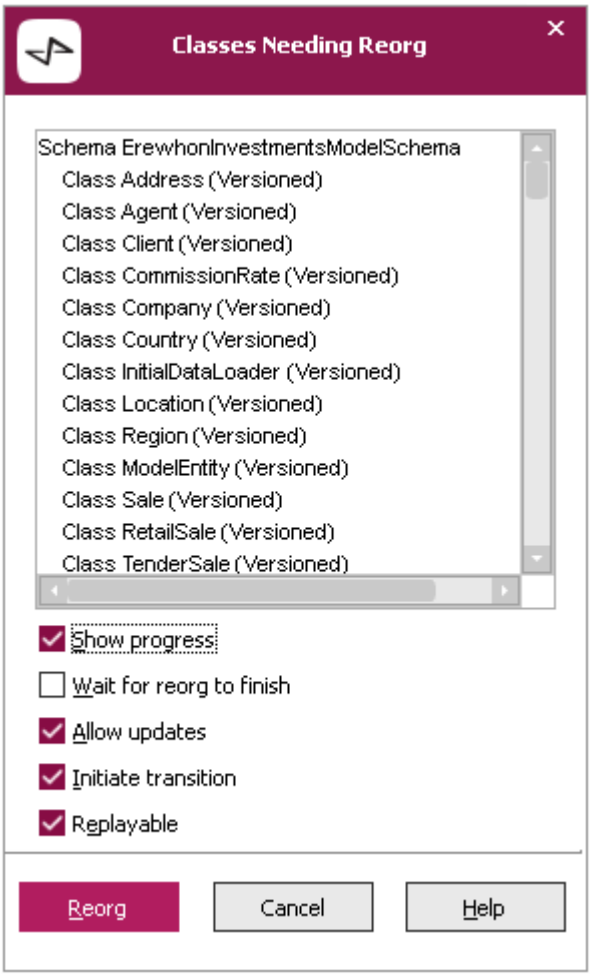
This change causes the schema to be versioned.



Clicking **OK** results in the display of the following message box.



5. Click the **Reorg** toolbar button in the Class Browser and click **Reorg** in the Classes Needing Reorg dialog that is then displayed.



6. When the reorganization completes, navigate to the **JadeScript** class within **ErewhonModelSchema**.

   You will see that the **newMethod** method has been removed; that is, the schema has reverted to the state it was when you switched from the master branch to the **MyNewBranch** branch.

Web Sockets

The WebSocket communications protocol facilitates real-time data streaming between a client and a server.

Use of WebSockets allows Jade server-side applications to send content to clients (whether other Jade systems or external systems) without needing to be first requested by the client. This allows messages to be passed back and forth while keeping the connection open, enabling two-way ongoing conversations between the client and server.

WebSockets and IIS

The Internet Information Server (IIS) is a Microsoft Web Server that allows a standard device (a desktop PC or a laptop) running Microsoft Windows to act as a web server. By proper configuration of IIS, any WebSocket requests that are sent to the device can be handled by a Jade database, which can respond to those requests.

By default, the WebSocket protocol is disabled in IIS. The handling of WebSocket requests must be enabled in the Windows Control Panel windows features so that IIS can then forward the requests to the Jade database.

When IIS is set up to respond to WebSocket requests, it must also be provided with a Native Module, which in the case of Jade, is a link to the **JadeWebSockets_IIS.dll** file in the **bin** directory of your Jade database.

Finally, this Native Module must be associated with the kind of request it is equipped to handle; that is, WebSocket requests (***.ws**).

The JadeWebSocket Initialization File

The **JadeWebSockets_IIS.ini** file has the following sections.

- [JadeWebSocket Rules], which is used for establishing mappings between URIs and specific hosts and port numbers.
- [JadeWebSocket Logging], which controls logging by the **JadeWebSockets_IIS.dll** library file.

The [JadeWebSocket Rules] section must have at least one mapping rule and can have as many mapping rules as required. Mapping rules have the following format.

RuleName=URL-pattern,host,port

In this format:

- ***RuleName*** is any unique name that identifies the rule.
- ***URL-pattern*** is a regular expression. All URLs that match the pattern are mapped to the host and port.
- ***host*** is a network-resolvable host name or IP address for the machine that hosts the Jade WebSocket server.
- ***port*** is the port number of the TCP port being handled by the Jade WebSocket server.

When attempting to match a URL to a host and port, the rules are tried in the order in which they appear in the **JadeWebSockets_IIS.ini** file. As such, if two rules can handle the same URL, the first one defined is used. For example, consider the following set of rules. (Lines beginning with **;** are comments.)

```
[JadeWebSocket Rules]
; ! note the single quotation marks around the URL pattern in the examples !

; http or https to domain.tld with the URL ending in .ws -> localhost:6666
appDomain='https?://domain.tld/.*\.ws',localhost,6666

; https to anything on google.com that ends in blah -> localhost:5555
appGoogle='https://.*\google.com/.*blah',localhost,5555

; http or https that ends in -1.ws -> localhost:5555
ws5555='https?://.*-1.ws',localhost,5555

;http or https anything -> localhost:4444
default='https?://.*',localhost,4444
```

In this example, **https://domain.tld/WS-1.ws** goes to **localhost:6666**, although it also matches the rule for **localhost:5555** and **localhost:4444**. This is because **localhost:6666** appears earlier in the [JadeWebSocket Rules] section of the file.

The [JadeWebSocket Logging] section controls logging by the **JadeWebSockets_IIS.dll** library file and it has the following parameters.

Parameter	Value	Purpose
Trace	Boolean	Controls the logging of **JadeWebSockets_IIS.dll** library activity. If **false**, no logging is performed. If **true**, logged messages acknowledge only that messages have been sent or received; not the content of those messages. The default value, if unset, is **false**.
TraceFile	String	Optional parameter that allows for the specification of a file path to override the default **jadeWebSockets_IIS<timestamp>.log** log file value.
TraceFileSize	Integer	Specifies the maximum file size, in bytes, before switching log files. The default value, if unset, is **1000000** (one million bytes).

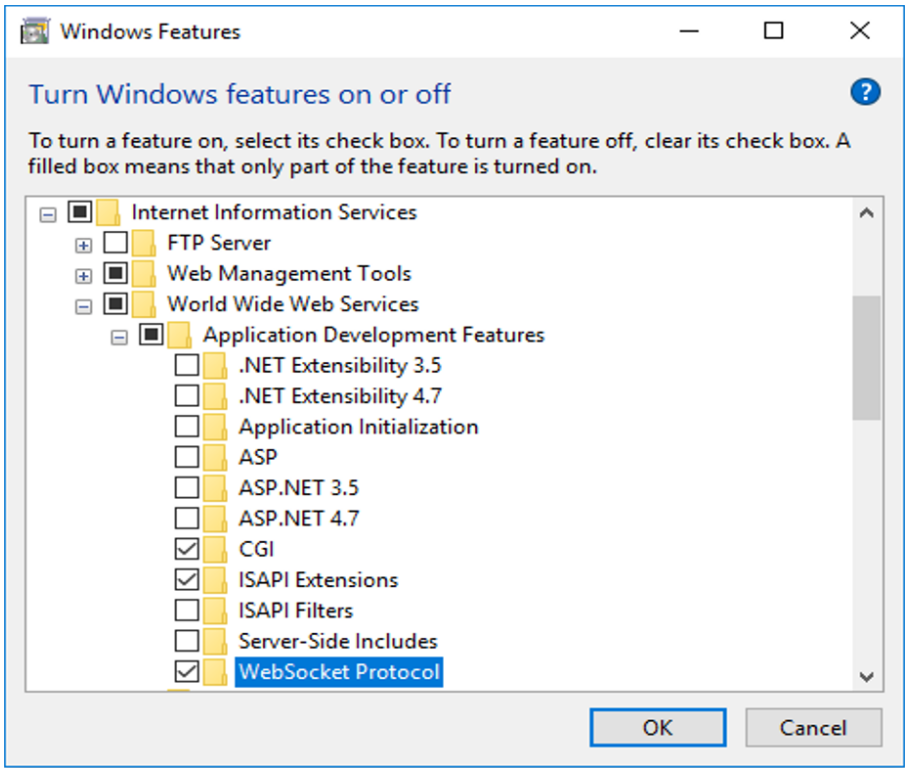
Exercise 1 – Enabling the Winsock Interface for IIS

In this exercise, ensure that the WebSocket protocol is enabled for IIS on your machine.

Note These instructions assume you are using Windows 10. Some details can be slightly different for earlier versions of Windows. In addition, to use WebSockets in Jade, IIS 8 or later is required.

1. Open the Windows Features window, by searching for **Turn Windows features on or off** in Windows search or finding **Turn Windows features on or off** in the panel at the left of the **Programs and Features** category of the Windows Control Panel.

2. Check the **WebSocket Protocol** check box in the expanded **Application Development Features** under **Internet Information Services**.

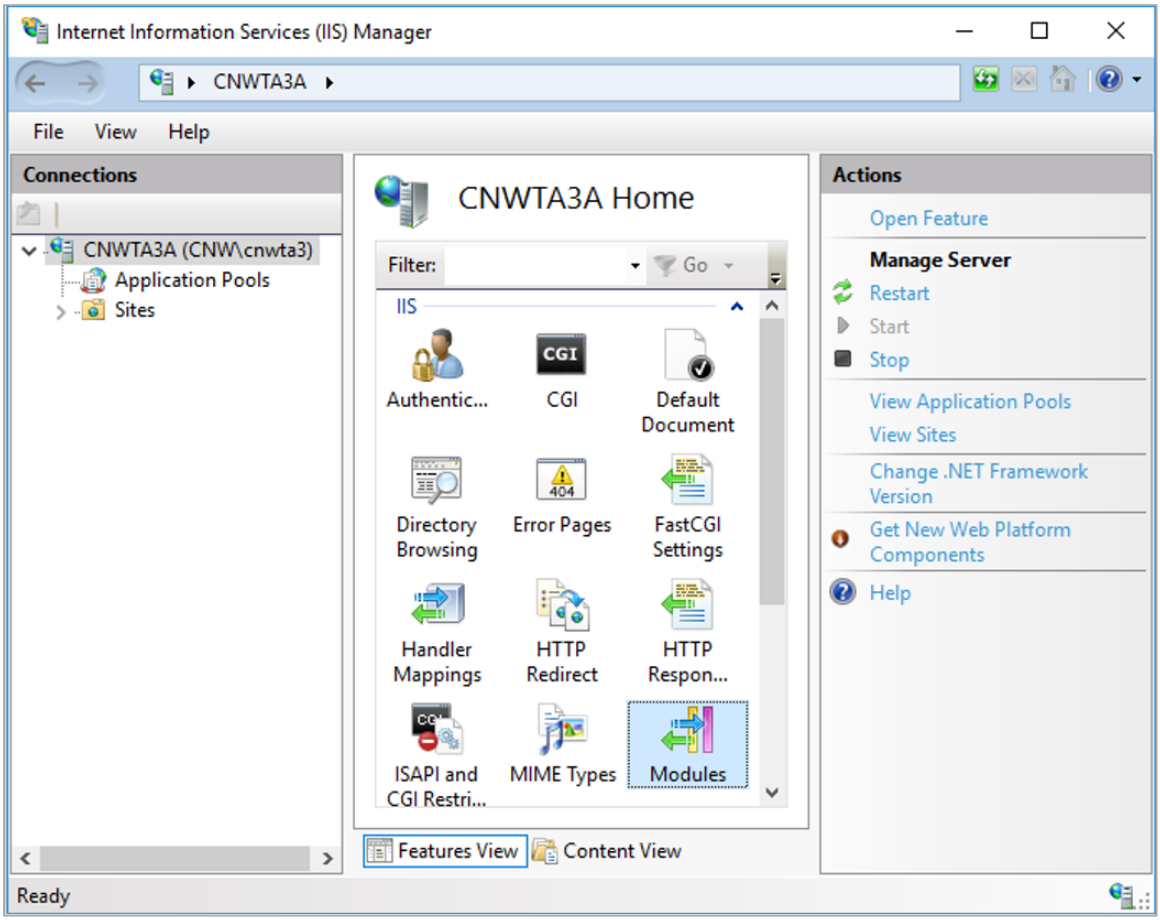


Exercise 2 – Installing the JadeWebSocket Module

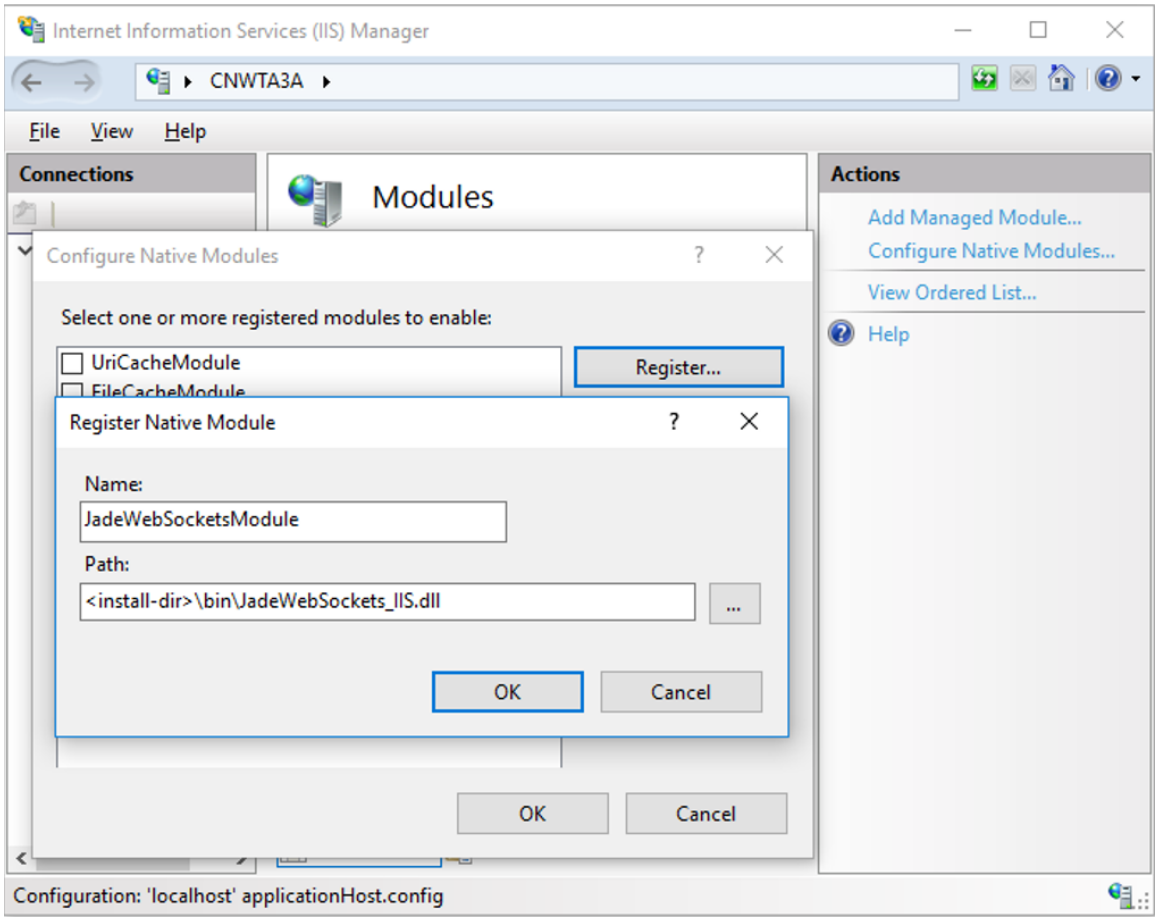
In this exercise, install **JadeWebSocket** as a module in IIS.

1. Open the IIS Manager by searching for **IIS** in Windows search or finding **Internet Information Services (IIS) Manager** in the Windows Control Panel **Administrative Tools** category.

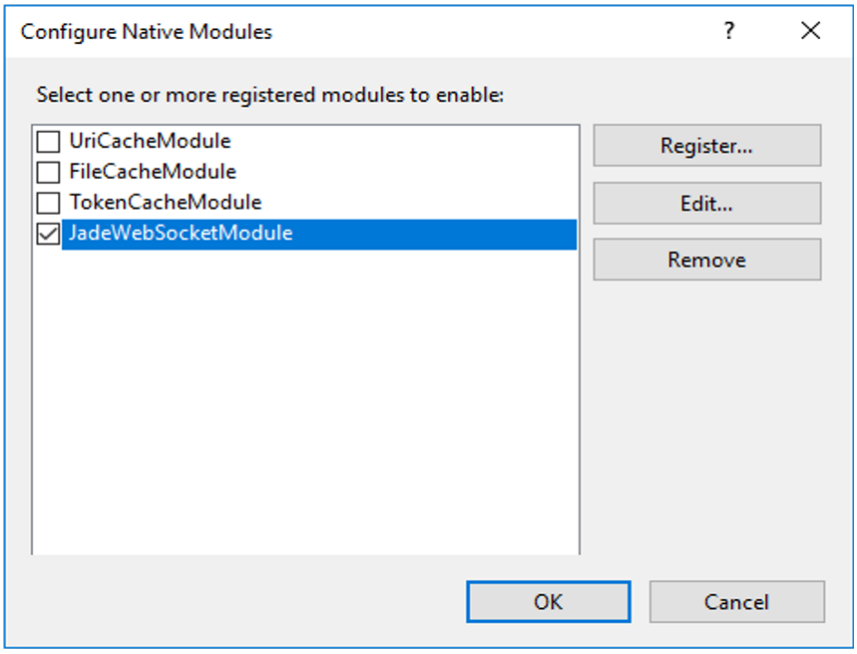
2. Select your machine from the **Connections** panel (**CNWTA3A**, in the following example), then double-click **Modules** in the center panel.



3. Select **Configure Native Modules** from the **Actions** panel, click **Register**, and fill out the dialog as follows. (The ***<install-dir>*** value is the location of your Jade Platform installation.)



4. Check the **JadeWebSocketModule** check box on the Configure Native Modules dialog and then click **OK**.



The **JadeWebSocketModule** is then displayed in the list on the **Modules** page.

5. IIS loads modules for requests in a specified order. Note that the **WebSocketModule** must be loaded before the **JadeWebSocketModule**.

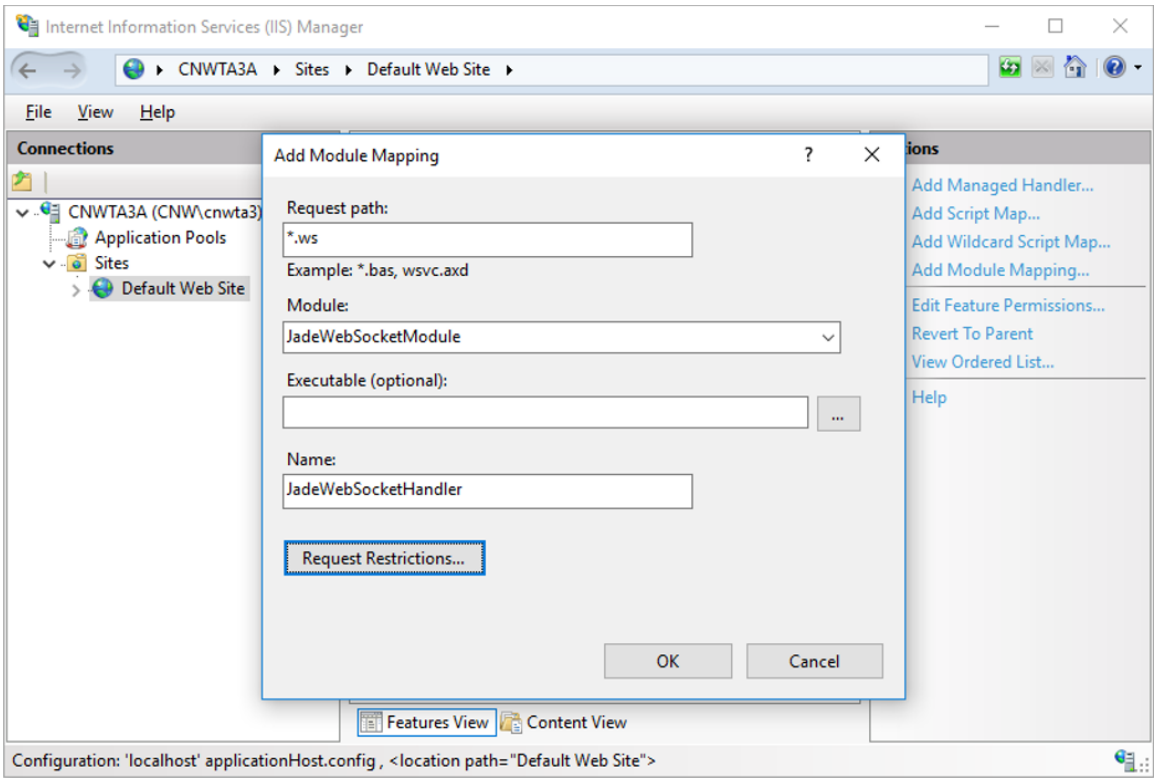
   a. Select **View Ordered List** from the **Actions** panel. The modules are then listed in the order in which they are loaded.

   b. Select the **WebSocketModule** in the list and then select **Move Up** or **Move Down** from the **Actions** panel until the **WebSocketModule** is displayed before the **JadeWebSocketModule** in the module order.

Exercise 3 – Enabling the JadeWebSocketModule

In this exercise, set all WebSocket requests to use the **JadeWebSocketModule** that was installed in the previous exercise.

1. In the **Connections** panel on the left, select **Default Web Site**.
2. From the middle panel, double-click **Handler Mappings**.
3. Select **Add Module Mapping** from the **Actions** panel on the right.
4. Fill out the Add Module Mapping dialog as follows and then click **Request Restrictions**.



5. On the **Mapping** sheet of the Request Restrictions dialog, uncheck the **Invoke handler only if request is mapped to:** check box and then click **OK**.

Exercise 4 – Configuring the JadeWebSockets Initialization File

In this exercise, configure the **JadeWebSockets_IIS.ini** initialization file to map URIs to port numbers.

1. Open the **JadeWebSockets_IIS.ini** file, located in **<*install-dir*>\bin_JadeWebSockets_IIS\ini**.
2. Set the following parameter values in the **JadeWebSockets_IIS.ini** file.

```
[JadeWebSocket Rules]
; note the single quotation marks around the URL pattern in the example !

;http or https anything -> localhost:4444
default='https?://.*',localhost,4444

[JadeWebSocket Logging]
verbose=true
trace=true
traceFile=default
traceFileSize=1000000000
```

The JadeWebSocketServer Class

The **JadeWebSocketServer** class of **RootSchema** handles all incoming TCP/IP connections (for example, WebSocket) coming over IIS from a specific interface and TCP port.

The **JadeWebSocketServer** class provides the following methods.

Method	Description
getWebSocket	Takes a WebSocket ID parameter and returns the **JadeWebSocket** with that ID.
run	Takes a **localInterface**, a **tcpPort**, and a **webSocketClass** parameter and begins assigning all connections on the specified **tcpPort** that meet the specified **localInterface** to the specified **webSocketClass**. This method continues executing until the **stop** method is called.
stop	Signals any active **run** methods to stop listening on the local interface and TCP/IP port, and terminates all WebSocket connections.

Connecting to a Jade WebSocket Server

When a Jade WebSocket server is set up and accepting WebSocket connections, external applications can send messages to the Jade database over the WebSocket protocol. For example, a web site could access the database with some JavaScript code.

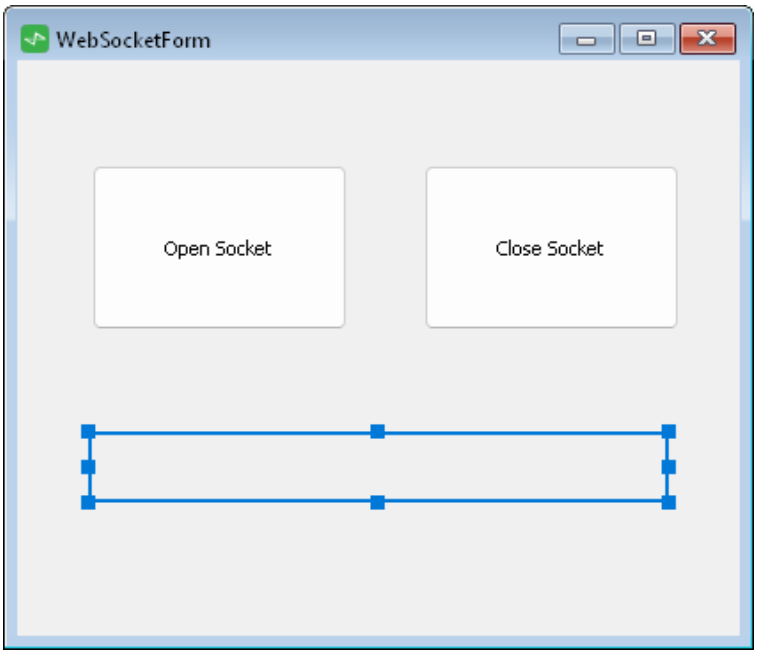
A full explanation of how to implement WebSocket clients in JavaScript or other technologies is beyond the scope of this module. However, the **WebSocketImage.html** and **WebSocketTest.html** files, found in the Zip file downloaded from https://secure.jadeworld.com/developer-centre/Education/DevCourse/JadeDevCourseFiles.zip, provide working demonstrations.

Exercise 5 – Creating a WebSocket Server Application

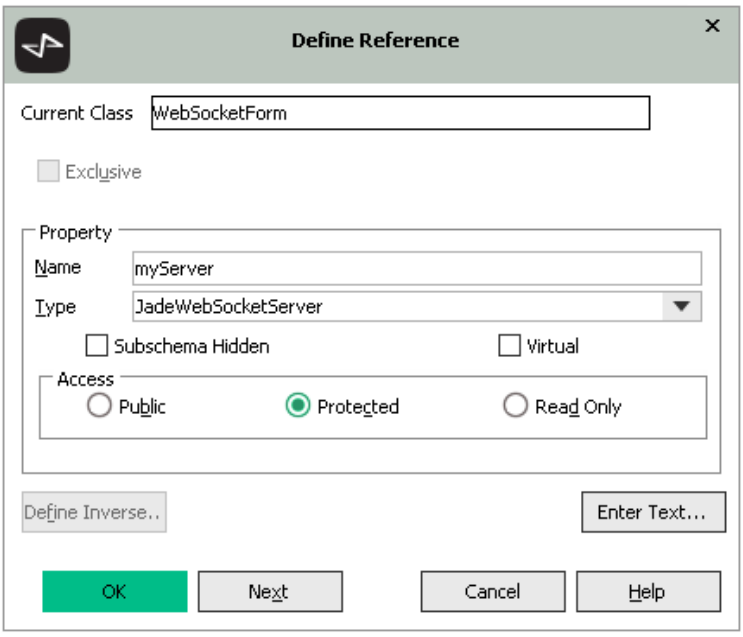
In this exercise, create a simple Jade application that can accept a connection from an external WebSocket client.

For this exercise, verify only that you can open a connection, although in future exercises you will add the ability to send and receive messages over WebSockets.

1. Create a schema called **WebSocketSchema** and from this schema, open the Jade Painter.
2. Create a form called **WebSocketForm**, with two buttons called **btnOpen** and **btnClose**, and a label called **lblState**, as follows.



3. Create a reference called **myServer** of type **JadeWebSocketServer** in the **WebSocketForm** class, as follows.



4. Code the **btnOpen_click** method as follows.

```
btnOpen_click(btn: Button input) updating;

begin
    create myServer transient;
    self.lblState.caption := "Socket is OPEN";
    self.myServer.run(null, 4444, JadeWebSocket);

epilog
    delete myServer;
    self.lblState.caption :="Socket is CLOSED";
end;
```

5. Code the **btnClose_click** method as follows.

```
btnClose_click(btn: Button input) updating;

begin
    if not myServer = null then
        self.myServer.stop();
    endif;
end;
```

6. Code the **unload** method (under **Form Events**) as follows. This ensures that the server is left in a closed state if the form is closed without closing the WebSocket connection.

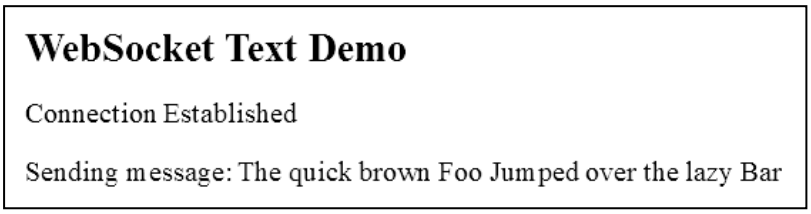
```
unload() updating;

begin
    if not myServer = null then
        self.myServer.stop();
    endif;
end;
```

7. Create a **JadeScript** class method called **startForm** and code it as follows.

```
startForm();

vars
    wsForm : WebSocketForm;

begin
    create wsForm;
    wsForm.showModal();
end;
```

8. Run the **JadeScript** class **startForm** method and then click **Open Socket**.

9. Run the **WebSocketTest.html** file, found the Zip file downloaded from https://secure.jadeworld.com/developer-centre/Education/DevCourse/JadeDevCourseFiles.zip.



As the WebSocket server is currently using the default implementation, it will not respond to the message. You should, however, see the WebSocket client successfully open a connection and send the message.

The JadeWebSocket Class

The **JadeWebSocket** class is the base class for handling a WebSocket connection. It differs from the **JadeWebSocketServer** class in that the **JadeWebSocketServer** class listens on a port and forwards connections to a **JadeWebSocket** subclass, which then can process or send messages.

The **JadeWebSocket** base class provides the following methods, which can be reimplemented on user defined subclasses.

Method	Description
onOpen	Automatically called when the WebSocket is opened from the client side of the connection. It receives one parameter, **fullUrl**, which is the full URL of the WebSocket that was used by the WebSocket initiating client.
onMsg	Automatically called when the WebSocket receives a message from the client side of the connection. It receives three parameters: the binary message that was sent, a boolean representing whether that message was UTF8-encoded, and a boolean representing whether the message is the final fragment; that is, if **false**, there is more message to come.
onClose	Automatically called when the client closes the WebSocket connection. This method receives no parameters.
send	Sends a binary message to the client via the WebSocket connection and requires two parameters: the binary message to send to the client, and a boolean representing whether this is the final fragment or whether there is more message to send.
sendText	Like the **send** method, except the message is sent as a UTF8 string rather than as a binary. As with the **send** method, it requires two parameters. However, the **msg** parameter must be of type UTF8 rather than binary.

As seen in the previous exercise, if the **JadeWebSocketServer** class **run** method is given the base class, it accepts connections but does not reply to any messages. This is because the default implementation simply passes without action when any of the methods in that class are called.

To implement behavior for the **JadeWebSocket** class, you must subclass it and reimplement the **onOpen**, **onMsg**, and **onClose** methods.

You can also reimplement the **send** and **sendText** methods, if required. Although it is not necessary to reimplement them, if you do, you should call the **inheritMethod** instruction within the reimplementation so that those methods still send messages to the client when called.

Exercise 6 – Responding to Messages

In this exercise, implement a subclass of the **JadeWebSocket** class to send a reply whenever it gets a message.

1. From the Class Browser or **WebSocketSchema**, press F4 to find the **JadeWebSocket** class and then create a subclass called **WSEcho**.

2. Add a method called **onMsg** to the **WSEcho** class.

   A warning is displayed, saying that you are implementing a superclass method. Click **Yes**.

3. Implement the method as follows.

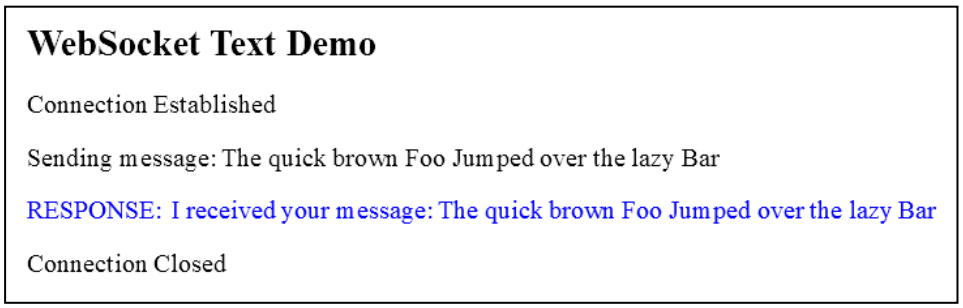
```
onMsg(msg: Binary; utf8encoded: Boolean; finalFragment: Boolean) updating, protected;

vars
    textMsg : StringUtf8;
begin
    textMsg := msg.StringUtf8;
    self.sendText("I received your message: " & textMsg, finalFragment);
end;
```

4. Modify the **btnOpen_click** method to replace **JadeWebSocket** with **WSEcho**, as follows.

```
btnOpen_click(btn: Button input) updating;

begin
    create myServer transient;
    self.lblState.caption := "Socket is OPEN";
    self.myServer.run(null, 4444, WSEcho);

epilog
    delete myServer;
    self.lblState.caption :="Socket is CLOSED";
end;
```

5. Run the **JadeScript** class **startForm** method and open the socket.

   When it's open, run the **WebSocketTest.html** again. You should see the following.



Exercise 7 – Reimplementing onOpen, onClose, and sendText

In this exercise, reimplement the **onOpen** and **onClose** methods to write to the Jade Interpreter Output Viewer whenever a connection is opened or closed, and the **sendText** method to write the sent message to the Jade Interpreter Output Viewer before sending it.

1. Add the following methods to the **WSEcho** class. Click **Yes** when warned about reimplementing a superclass method.

```
onOpen(fullUrl: String) protected;

begin
    write "Opened connection from URL: " & fullUrl;
end;
```

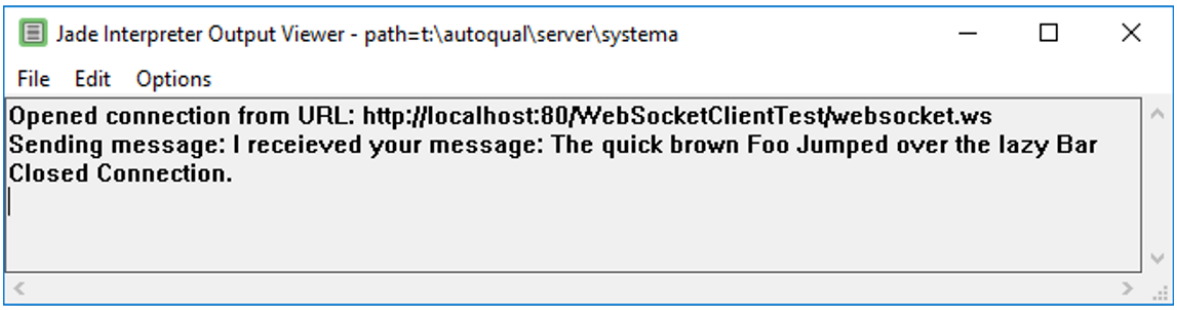
```
sendText(msg: StringUtf8; finalFragment: Boolean);

begin
    write "Sending message: " & msg;
    inheritMethod(msg, finalFragment);
end;
```

```
onClose() protected;

begin
    write "Closed Connection.";
end;
```

Note The **onOpen** and **onClose** methods have empty default implementations, so it is not useful to call the **inheritMethod** instruction for those methods. However, as the default implementation of the **sendText** method is responsible for sending the message to the client, it is therefore important to call it somewhere in the reimplementation by using the **inheritMethod** instruction.

2. Ensuring that **WebSocketForm** is running and the socket is open, run the **WebSocketTest.html** file.

   In addition to the display in the web browser, the following should be displayed in the Jade Interpreter Output Viewer.



Exercise 8 – Sending Images over WebSockets

In this exercise, use the WebSocket protocol to send an image from Jade to an HTML JavaScript test page.

1. Create a **C:/Images** folder and copy the **JADE.jpeg** file, which can be found in the Zip file downloaded from https://secure.jadeworld.com/developer-centre/Education/DevCourse/JadeDevCourseFiles.zip, to the newly created folder.

2. Add a **sendImage** method to the **WSEcho** class, coded as follows.

```
sendImage();

vars
    file : File;
begin
    write "Got message to start sending a picture";
    create file;
    file.kind := File.Kind_Binary;
    file.openInput('C:/Images/JADE.jpeg');
    self.send(file.readBinary(file.fileLength), true);
epilog
    delete file;
end;
```

3. Modify the **WSEcho** class **onMsg** method as follows.

```
onMsg(msg: Binary; utf8encoded: Boolean; finalFragment: Boolean) updating, protected;

vars
    textMsg : StringUtf8;
begin
    textMsg := msg.StringUtf8;
    if textMsg = "Image Please" then
        self.sendImage();
    else
        self.sendText("I received your message: " & textMsg, finalFragment);
    endif;
end;
```

4. Run the **WebSocketImage.html** file, found in the Zip file downloaded from https://secure.jadeworld.com/developer-centre/Education/DevCourse/JadeDevCourseFiles.zip. You should see the **JADE.jpeg** image displayed as the returned message.

XML in Jade

XML, or eXtensible Markup Language, is a human-readable, text-based language that consists of *nodes*, containing data, and tags, describing the type of the data between the tags.

Although XML can be used for a variety of applications, the most common is transferring data over the internet. The Jade Platform provides several **RootSchema** classes for the conversion of data between Jade database classes and XML documents.

Structure of XML

XML has a similar structure to other markup languages (for example, HTML) in that an XML document consists of data contained within tags, potentially nested. As with other markup languages, tags are of the form **<TagName>** ***data*** **</TagName>**. However, unlike many other markup languages, XML allows custom tags.

Despite the flexibility to markup data with whatever term best describes it, there are still rules as to how the document must be laid out. For example, an XML document must have a 1:1 pairing between opening and closing tags and nesting integrity must be maintained.

When all XML rules are followed, the document is deemed to be *well-formed*. Being well-formed means the XML document is syntactically correct according to the World Wide Web Consortium (W3C) XML Specification. It does not, however, guarantee that the data is correct.

The following XML document is well-formed.

```
1  <?xml version="1.0"?>
2  <!--This is a comment -->
3  <company>
4      <employee id="jtm314">
5          <name>Jack Mason</name>
6          <role>Abstraction Instantiator</role>
7          <salary>99550</salary>
8      </employee>
9
10     <customer id="CUST00349">
11         <name>Lee Sah</name>
12         <address>42 Fiction Ave</address>
13         <email>ls99@e.mail</email>
14     </customer>
15 </company>
```

The following XML document is not well-formed, because the employee opening tag does not have a closing tag.

```
<?xml version="1.0"?>
<!--This is a comment -->
<company>
    <employee id="jtm314">
        <name>Jack Mason</name>
        <role>Abstraction Instantiator</role>
        <salary>99550</salary>

    <customer id="CUST00349">
        <name>Lee Sah</name>
        <address>42 Fiction Ave</address>
        <email>ls99@e.mail</email>
    </customer>
</company>
```

The following XML document is *also* not well-formed, because the customer opening tag is nested inside the company tag, but the closing tag lies outside it.

```
<?xml version="1.0"?>
<!--This is a comment -->
<company>
    <employee id="jtm314">
        <name>Jack Mason</name>
        <role>Abstraction Instantiator</role>
        <salary>99550</salary>
    </employee>

    <customer id="CUST00349">
        <name>Lee Sah</name>
        <address>42 Fiction Ave</address>
        <email>ls99@e.mail</email>
</company>
    </customer>
```

Note Although the indentation is irrelevant to whether the XML is well-formed, having good indentation practices makes XML documents easier to read by humans. When Jade generates XML documents, it handles the indentation for you.

Creating XML

The Jade Platform provides the **JadeXMLDocument** and **JadeXMLElement** classes for generating and presenting well-formed XML.

The **JadeXMLDocument** class represents an XML document as a tree of nodes and defines the root; that is, the owning object of all objects in the tree.

The **JadeXMLElement** class represents an XML element in a document tree and along with a unique name, can contain any of the following.

- Child nodes; that is, other **JadeXMLElement** objects
- Attributes; for example, **id="CUST00349"**
- Text; for example, **Lee Sah** in a **name** node

To add **JadeXMLElement** nodes to the main document (a **JadeXMLDocument** object) or another **JadeXMLElement** node, both classes provide the **addElement** method. This method takes a **tagName** parameter of type **String**, which represents the name of the tag.

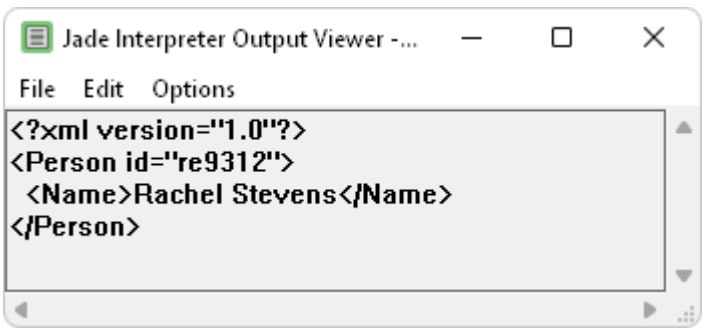
To add attributes to a node, the **JadeXMLElement** class provides the **addAttribute** method, which takes two parameters of type **String**. The **name** parameter represents the name of the attribute (for example, **id**) and the **value** parameter represents the value of the attribute (for example, **rs9312**).

Text can be added to a node by using the **JadeXMLElement** class **setText** method, which takes a single **String** parameter for the text to add to the node.

The following example shows the method used to create an XML document.

```
exampleXML();

vars
    xmlDoc   : JadeXMLDocument;
    xmlNode1 : JadeXMLElement;
    xmlNode2 : JadeXMLElement;

begin
    // xmlDoc represents the whole XML document,
    // and the root of the tree.
    create xmlDoc transient;

    // xmlNode1 represents a <Person> tag in the xml document,
    // and a node in the tree.
    xmlNode1 := xmlDoc.addElement("Person");

    // xmlNode1 has one attribute, which will,
    // look like <Person id="rs9312"> in the xml.
    xmlNode1.addAttribute("id", "re9312");

    // xmlNode2 represents a <Name> tag within the xml,
    // and is a child of <Person>.
    xmlNode2 := xmlNode1.addElement("Name");

    // The text inside the <Name> tag will be
    // Rachel Stevens.
    xmlNode2.setText("Rachel Stevens");

    // Write the XML document to the output window.
    // Can also save this to a file (usually more useful).
    write xmlDoc.writeToString;
epilog
    delete xmlDoc;
end;
```

The following example shows the output of the above method used to create an XML document.

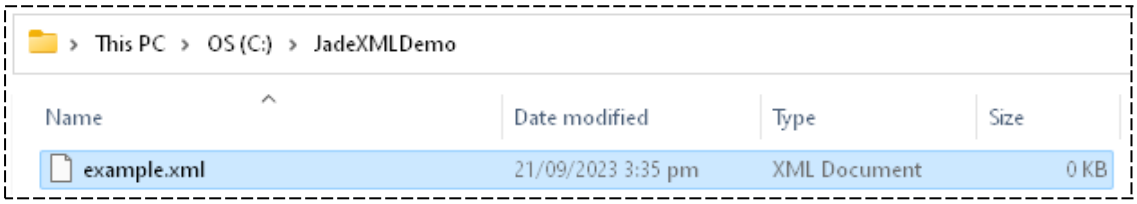


The **JadeXMLDocument** class also provides the **writeToFile** method, which takes a **filePath** as a parameter of type **String**, and outputs the XML document to the specified file. Note that if there is no file at the **filePath** location, it creates one, and if the file exists, the contents of that file are overwritten following confirmation.

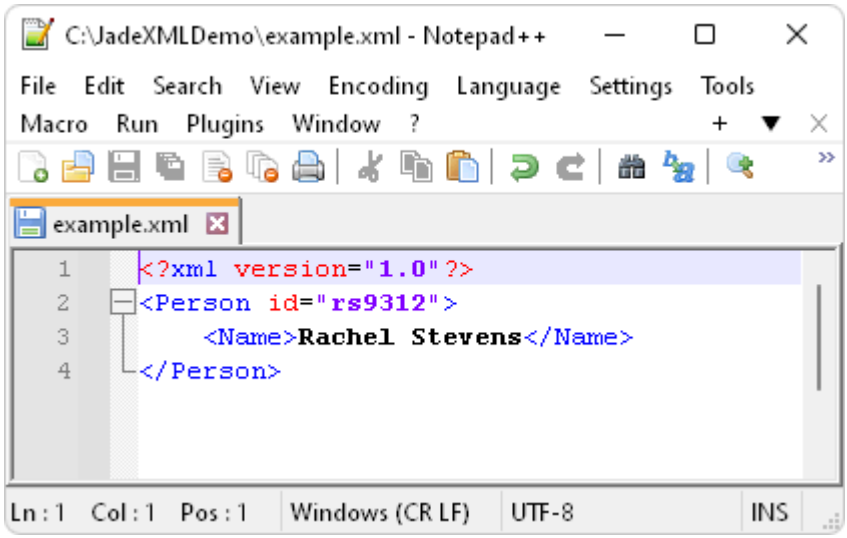
The following code fragment changes from **writeToString** to **writeToFile**.

```
// Writing the XML to a file instead....
xmlDoc.writeToFile("C:\JadeXMLDemo\example.xml");
```

The above change results in the creation of an XML file at the file path in the following example.



This file path location contains the XML document shown in the following example.



Exercise 1 – Creating an XML Document

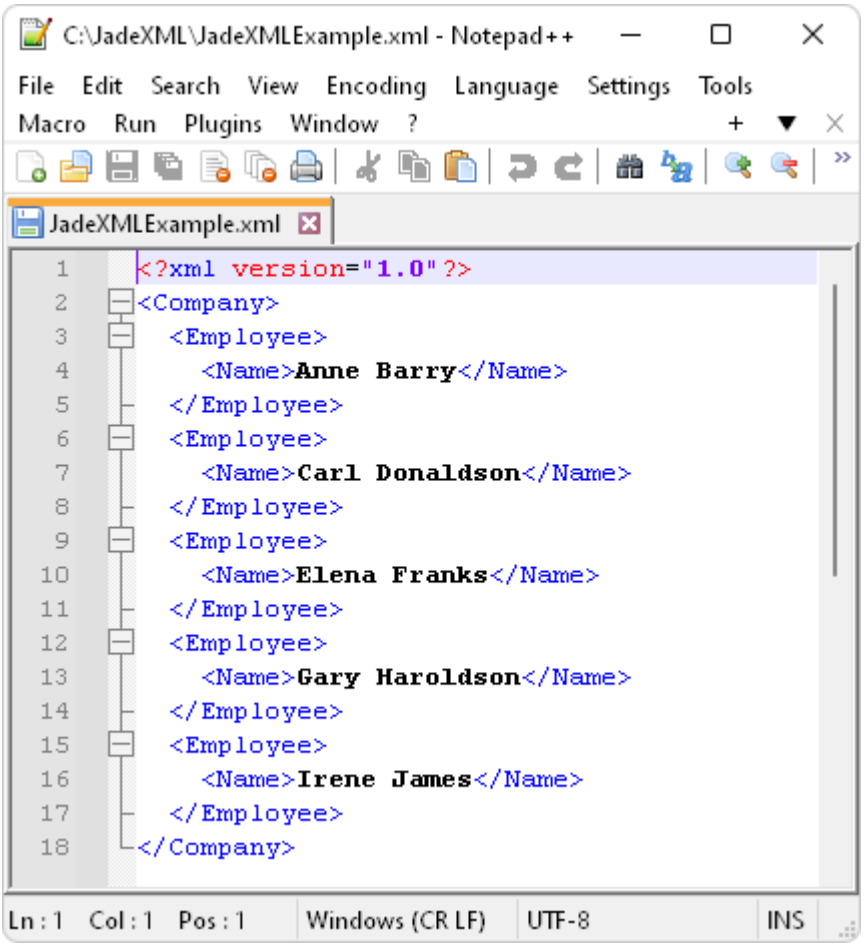
In this exercise, create an XML document, add nodes to form a tree structure, and then write the XML document to a file.

1. Create a new schema called **XMLSchema**.
2. Create a new **JadeScript** class method called **createXML** and code it as follows.

```
createXML();

vars
    xmlDoc      : JadeXMLDocument;
    xmlCompany  : JadeXMLElement;
    xmlEmployee : JadeXMLElement;
    xmlCustomer : JadeXMLElement;
    xmlData     : JadeXMLElement;
begin
    create xmlDoc transient;

    xmlCompany := xmlDoc.addElement("Company");
        xmlEmployee := xmlCompany.addElement("Employee");
            xmlData := xmlEmployee.addElement("Name");
            xmlData.setText("Anne Barry");
        xmlEmployee := xmlCompany.addElement("Employee");
            xmlData := xmlEmployee.addElement("Name");
            xmlData.setText("Carl Donaldson");
        xmlEmployee := xmlCompany.addElement("Employee");
            xmlData := xmlEmployee.addElement("Name");
            xmlData.setText("Elena Franks");
        xmlEmployee := xmlCompany.addElement("Employee");
            xmlData := xmlEmployee.addElement("Name");
            xmlData.setText("Gary Haroldson");
        xmlEmployee := xmlCompany.addElement("Employee");
            xmlData := xmlEmployee.addElement("Name");
            xmlData.setText("Irene James");
    xmlDoc.writeToFile("c:\JadeXML\JadeXMLExample.xml");
epilog
    delete xmlDoc;
end;
```

3. On your file system, create a new folder on your **C:** drive called **JadeXML**.
4. Execute (run) the **JadeScript** class **createXML** method. The file **JadeXMLExample.xml** should be generated in your **JadeXML** folder.
5. Inspect the created file (using Notepad or Notepad++).

The following XML should be displayed.



Exercise 2 – Converting a Jade Database to XML

In this exercise, convert Jade objects into XML nodes for a small database.

1. Select the **Load** command from the Schema menu and then load the provided **XMLExampleSchema.scm**.
2. Copy the provided **customers.txt** and **employees.txt** files to the **C:\JadeXML** folder you created in the previous exercise in this module.
3. From **XMLExampleSchema**, run the **JadeScript** class **loadAll** method, which populates the database with several employees and customers.

4. Create a new **JadeScript** class **buildXML** method and code it as follows.

```
buildXML();

    vars
        xmlDocument : JadeXMLDocument;
        xmlCompany  : JadeXMLElement;
        xmlCustomer : JadeXMLElement;
        xmlEmployee : JadeXMLElement;
        xmlProperty : JadeXMLElement;
        company     : Company;
        customer    : Customer;
        employee    : Employee;
    begin
        create xmlDocument transient;
        company := Company.firstInstance();

        xmlCompany := xmlDocument.addElement("Company");

        foreach customer in company.allMyCustomers do
            xmlCustomer := xmlCompany.addElement("Customer");
            xmlCustomer.addAttribute("id", customer.id);
            xmlProperty := xmlCustomer.addElement("name");
            xmlProperty.setText(customer.name);
        endforeach;

        foreach employee in company.allMyEmployees do
            xmlEmployee := xmlCompany.addElement("Employee");
            xmlEmployee.addAttribute("id", employee.id);
            xmlProperty := xmlEmployee.addElement("name");
            xmlProperty.setText(employee.name);
            xmlProperty := xmlEmployee.addElement("salary");
            xmlProperty.setText(employee.salary.String);
        endforeach;

        xmlDocument.writeToFile("C:\JadeXML\JadeCompanyXML.xml");

    epilog
        delete xmlDocument;
    end;
```

5. Press F9 to execute the method and then inspect the **C:\JadeXML\JadeCompanyXML.xml** file to view the contents of the database in XML format.

Loading XML

In addition to generating XML documents, the **JadeXMLDocument** can also load existing XML documents, creating a tree structure suitable for traversing and extracting data.

The **JadeXMLDocument** class provides the **parseFile** and **parseString** methods, which essentially mirror the **writeToFile** and **writeToString** methods. The difference between the **writeToFile** and **parseFile** methods is that **parseFile** creates a new **JadeXMLDocument** based on an existing XML file, while **writeToFile** takes an existing **JadeXMLDocument** and creates a new XML file.

To use either method, you need only create the transient **JadeXMLDocument** and call the appropriate parse method; for example, to read an XML file using the **parseFile** method it needs only the following.

```
parseXML();

vars
    xmlDocument : JadeXMLDocument;

begin
    create xmlDocument transient;
    xmlDocument.parseFile("C:\JadeXML\JadeCompanyXML.xml");
epilog
    delete xmlDocument;
end;
```

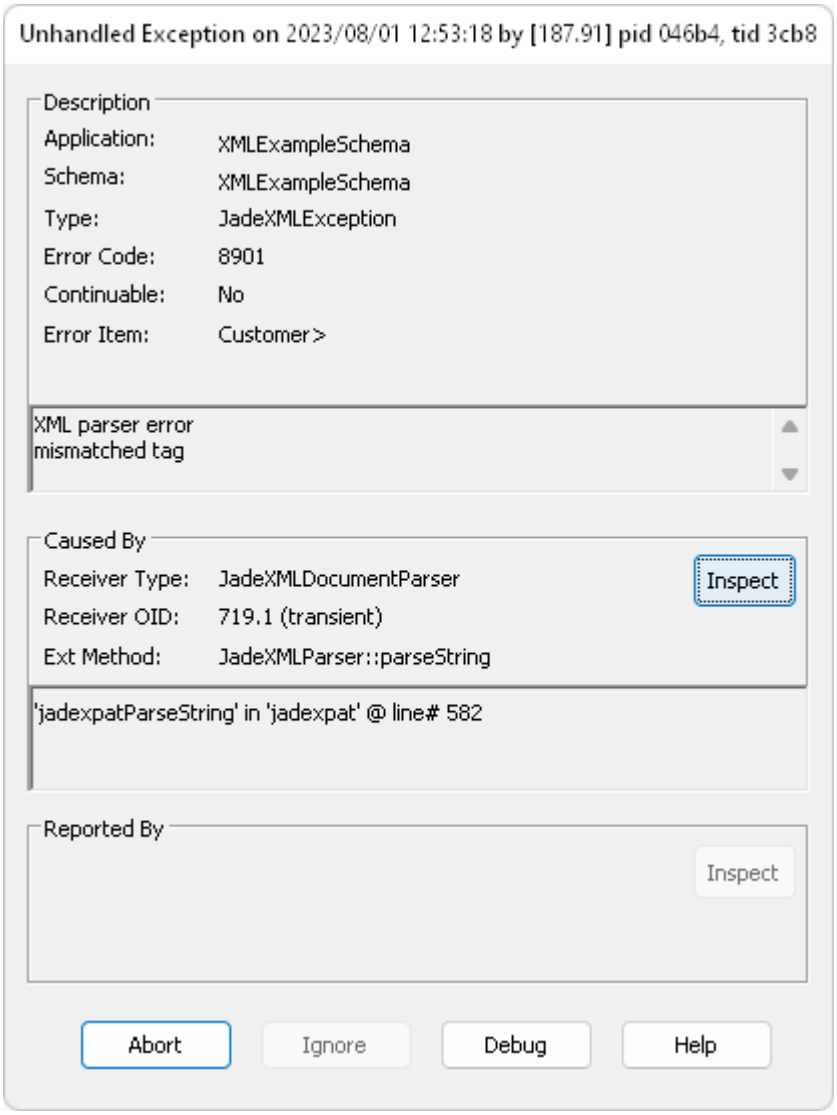
Handling Jade XML Exceptions

Any XML document generated using Jade is always well-formed XML. However, when parsing XML from external sources, there is a chance that the XML being parsed may not be well formed.

If the XML being parsed is not well-formed, the **parseFile** method raises a **JadeXMLException**.

```
parseXML();

vars
    xmlDocument : JadeXMLDocument;
begin
    create xmlDocument transient;
    xmlDocument.parseFile("C:\JadeXML\DodgyXML.xml");
epilog
    delete xmlDocument;
end;
```

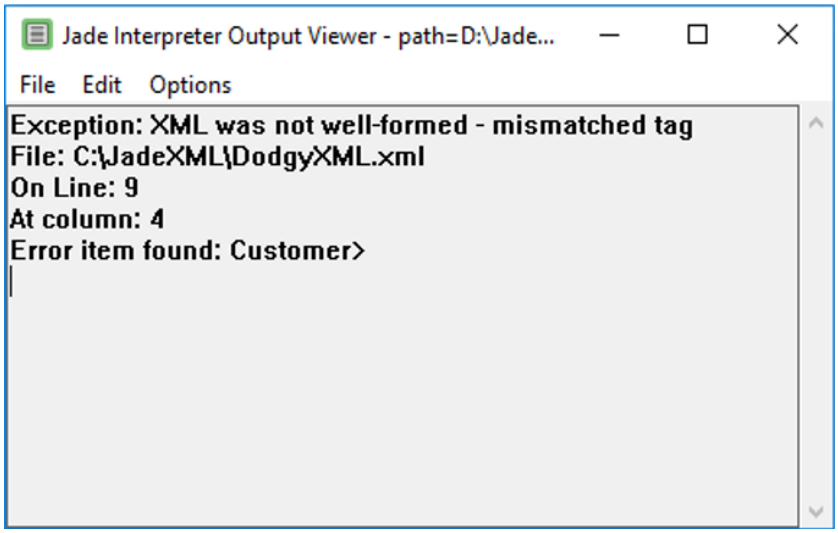
The following is an example of the exception raised by the above method.



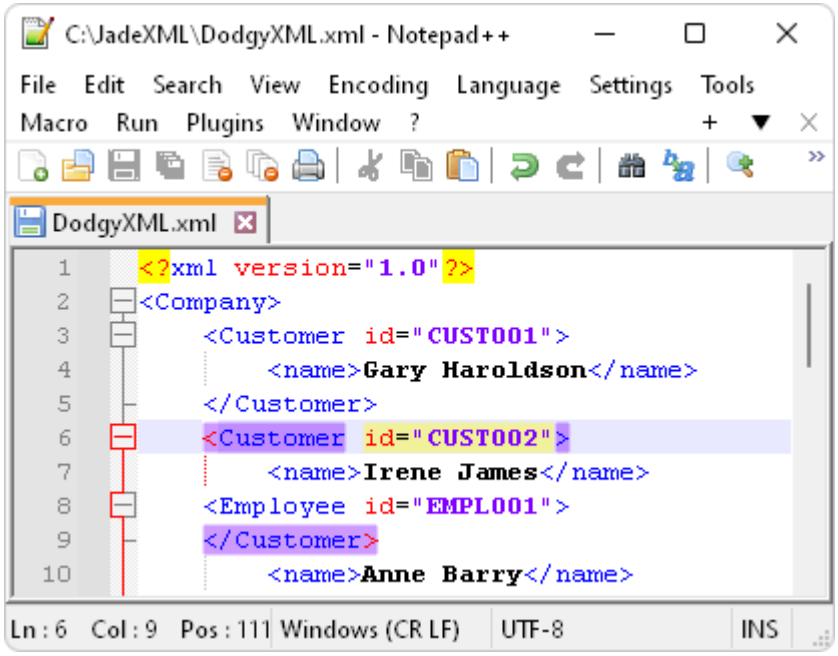
The generated **JadeXMLException** contains the line and column number of the error, as well as the type of error and the item in error. As such, an exception handler can display the following.

```
xmlExceptionHandler(e : JadeXMLException) : Integer;

begin
    write "Exception: XML was not well formed - " & e.extendedErrorText;
    write "File: " & e.fileName;
    write "On line: " & e.lineNumber.String;
    write "At column: " & e.columnNumber.String;
    write "Error item found: " & e.errorItem;
    return Ex_Abort_Action;
end;
```

The above method writes the following to the Jade Interpreter Output Viewer.



This allows for the quick and easy identification of where exactly, and what exactly, is the problem in the XML source. In this example, the **Customer** tag is mismatched, as the **Employee** tag was opened but not closed when the **Customer** tag was closed.



Searching JadeXMLDocuments

Once an XML document is successfully loaded and a **JadeXMLDocument** created, the following methods can be used to locate specific elements in the tree.

Method	Purpose
getElementByTagName	Takes a **tagName** parameter of type **String** and returns the first **JadeXMLElement** that has a matching **tagName** property.

Method	Purpose
findElementByTagName	Takes a **tagName** parameter of type **String** and returns the first **JadeXMLElement** that has a matching **tagName** property. Has a faster performance than **getElementByTagName** but may not return the document's first instance when there is more than one element of that **tagName**.
getElementsByTagName	Takes a **tagName** parameter of type **String** and an **elements** parameter of type **JadeXMLElementArray** and populates the specified **elements** array (in document order) with all **JadeXMLElements** that have a matching **tagName** property.
findElementsByTagName	Takes a **tagName** parameter of type **String** and an **elements** parameter of type **JadeXMLElementArray** and populates the specified **elements** array with all **JadeXMLElements** that have a matching **tagName** property. Has a faster performance than the **getElementsByTagName** method but the elements are not guaranteed to be sorted by document order.

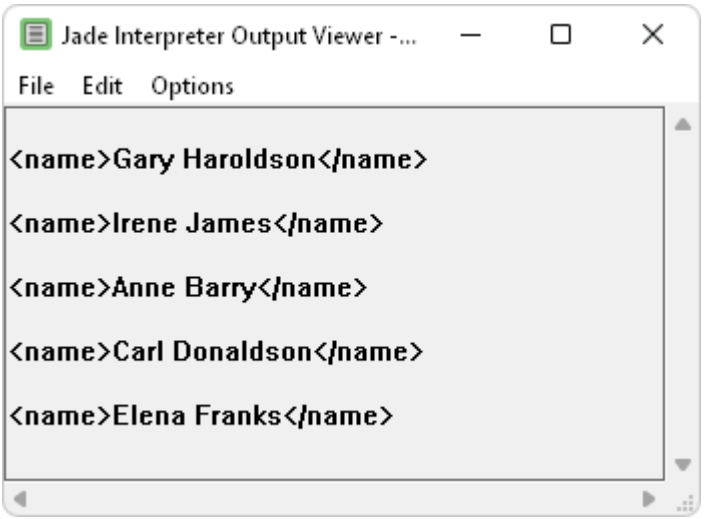
The **findElementByTagName** and **getElementByTagName** methods are most useful for locating elements for which you know there will be no other elements that share a tag; for example, a root object such as **Company**.

For finding elements of which there is expected to be many, the **getElementsByTagName** and **findElementsByTagName** methods are more appropriate, as the populated arrays can be iterated to display or use the data contained in all elements.

For example, to display all name elements (whether **Customer** or **Employee**) in the **JadeCompanyXML.xml** file, the following **JadeScript** class method can be used.

```
displayAllNamesExample();

vars
    xmlDocument : JadeXMLDocument;
    allNames    : JadeXMLElementArray;
    name        : JadeXMLElement;
begin
    create xmlDocument transient;
    create allNames transient;
    on JadeXMLException do xmlExceptionHandler(exception);
    xmlDocument.parseFile("C:\JadeXML\JadeCompanyXML.xml");
    xmlDocument.getElementsByTagName("Name", allNames);
    foreach name in allNames do
        write name.writeToString();
    endforeach;
epilog
    delete xmlDocument;
    delete allNames;
end;
```

This method produces the following output.



For a **JadeXMLElement** object, the following methods are available to extract data from the element.

Method	Purpose
getAttributeByName	Takes a name parameter of type **String** and returns the **JadeXMLAttribute** of the element with that name.
getElementByTagName	Takes a **tagName** parameter of type **String** and returns the first immediate child **JadeXMLElement** with that **tagName**.
getElementsByTagName	Takes a **tagName** parameter of type **String** and an **elements** parameter of type **JadeXMLElementArray** and populates the specified **elements** array with all immediate child elements with a matching **tagName**.
findAllElementsByTagName	Similar to the **getElementsByTagName** method except that it populates the specified **elements** parameter with *all* child elements that match the **tagName**; not just the immediate children.

Exercise 3 – Handling XML Exceptions

In this exercise, load an XML document that is not well-formed and use the **JadeXMLException** object to find and fix the error.

1. Ensure that the **JadeCompanyMalformed**.xml file has been copied to the **C:\JadeXML** folder.
2. Add a **JadeScript** class method called **xmlExceptionHandler**, coded as follows, to the **XMLExampleSchema**.

```
xmlExceptionHandler(e : JadeXMLException) : Integer;

begin
    write "The XML was malformed on line "
        & e.lineNumber.String
        & " at position "
        & e.columnNumber.String;
    write "Error item found: " & e.errorItem;
    return Ex_Abort_Action;
end;
```

3. Add a **JadeScript** class method called **loadMalformedXML**, coded as follows.

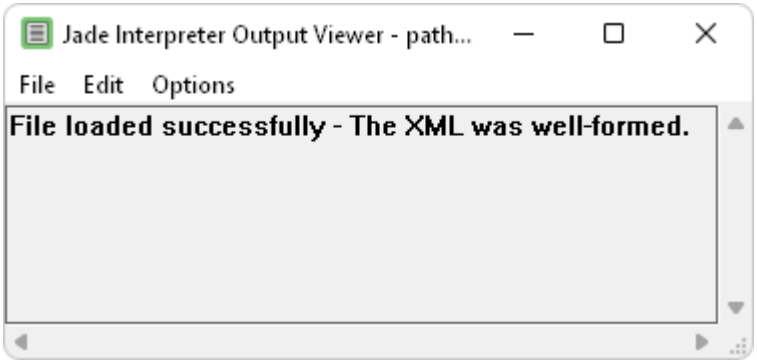
```
loadMalformedXML();

vars
    xmlDocument  : JadeXMLDocument;
    foundElement : JadeXMLElement;
begin
    create xmlDocument transient;
    on JadeXMLException do xmlExceptionHandler(exception);
    xmlDocument.parseFile("C:\JadeXML\JadeCompanyMalformed.xml");
    write "File loaded successfully - The XML was well-formed.";

epilog
    delete xmlDocument;
end;
```

4. Run the method. Note the line and position of the error, output to the Jade Interpreter Output Viewer. The error item is also displayed.

5. Open the **JadeCompanyMalformed.xml** file in a text editor perform one of the following actions.

   - Navigate to the line and position of the error and correct the typographical error in the tag name.
   - Search for the misspelled tag name shown in the error item and then correct it.

6. Rerun the **loadMalformedXML** method. This time it should load without errors and the following should be displayed in the Jade Interpreter Output Viewer.



Exercise 4 – Populating a Database from an XML File

In this exercise, populate the database with the **Customers** and **Employees** in the now well formed **JadeCompanyMalformed.xml** file.

1. Add a **JadeScript** class method called **createCustomers**, coded as follows, to **XMLExampleSchema**.

```
createCustomers(company : Company input; xmlDocument : JadeXMLDocument);

vars
    allCustomerElements : JadeXMLElementArray;
    foundElement        : JadeXMLElement;
    nameElement         : JadeXMLElement;
    idAttribute         : JadeXMLAttribute;
    customer            : Customer;
begin
    create allCustomerElements transient;
    xmlDocument.findElementsByTagName("Customer", allCustomerElements);
    foreach foundElement in allCustomerElements do
        nameElement := foundElement.getElementByTagName("name");
        idAttribute := foundElement.getAttributeByName("id");
        beginTransaction;
        create customer persistent;
        customer.id := idAttribute.value;
        customer.name := nameElement.textData;
        company.allMyCustomers.add(customer);
        commitTransaction;
    endforeach;
epilog
    delete allCustomerElements;
end;
```

2. Add a new **JadeScript** class method called **createEmployees**, coded as follows.

```
createEmployees(company : Company input; xmlDocument : JadeXMLDocument);

vars
    allEmployeeElements : JadeXMLElementArray;
    foundElement        : JadeXMLElement;
    nameElement         : JadeXMLElement;
    idAttribute         : JadeXMLAttribute;
    salaryElement       : JadeXMLElement;
    employee            : Employee;
begin
    create allEmployeeElements transient;
    xmlDocument.findElementsByTagName("Employee", allEmployeeElements);
    foreach foundElement in allEmployeeElements do
        nameElement := foundElement.getElementByTagName("name");
        idAttribute := foundElement.getAttributeByName("id");
        salaryElement := foundElement.getElementByTagName("salary");
        beginTransaction;
        create employee persistent;
        employee.id := idAttribute.value;
        employee.name := nameElement.textData;
        employee.salary :=salaryElement.textData.Integer;
        company.allMyEmployees.add(employee);
        commitTransaction;
    endforeach;
epilog
    delete allEmployeeElements;
end;
```

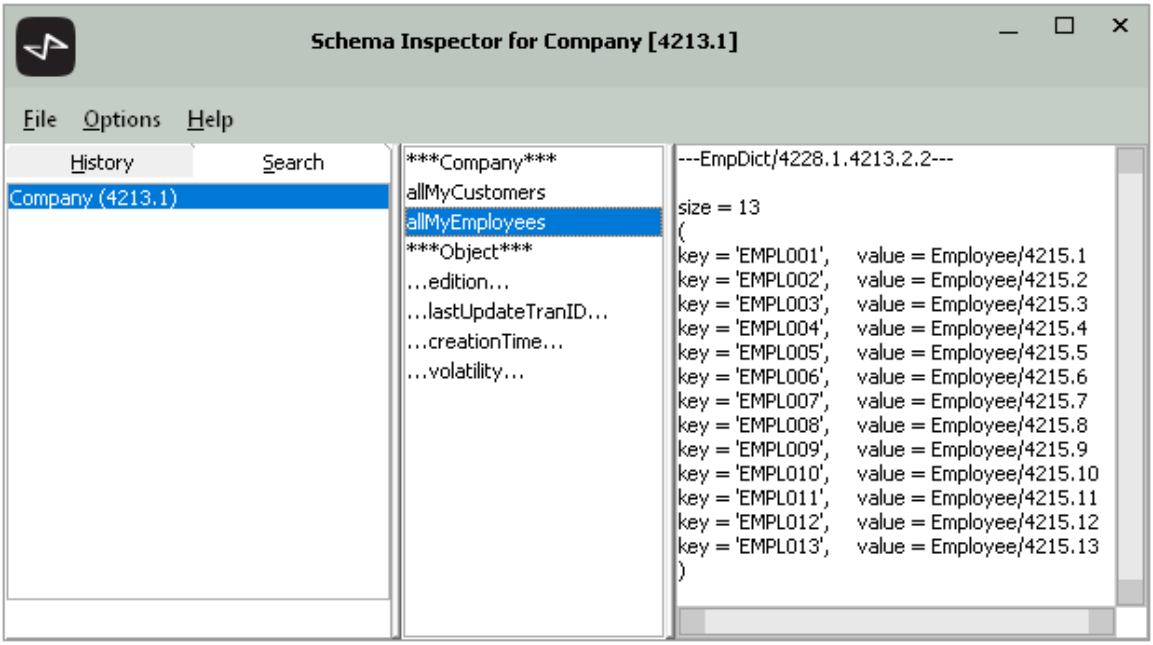
3. Modify the **loadMalformedXML** method as follows.

```
loadMalformedXML();

vars
    xmlDocument : JadeXMLDocument;
    company     : Company;
begin
    create xmlDocument transient;
    on JadeXMLException do xmlExceptionHandler(exception);
    xmlDocument.parseFile("C:\JadeXML\JadeCompanyMalformed.xml");
    write "File loaded successfully - The XML was well-formed.";

    beginTransaction;
    Company.instances.purge();
    Customer.instances.purge();
    Employee.instances.purge();
    create company persistent;
    commitTransaction;

    createCustomers(company, xmlDocument);
    createEmployees(company, xmlDocument);
epilog
    delete xmlDocument;
end;
```

4. Run the **loadMalformedXML** method and then inspect the **Company** object by selecting **Company** in the Class Browser and using the Ctrl+I shortcut keys.

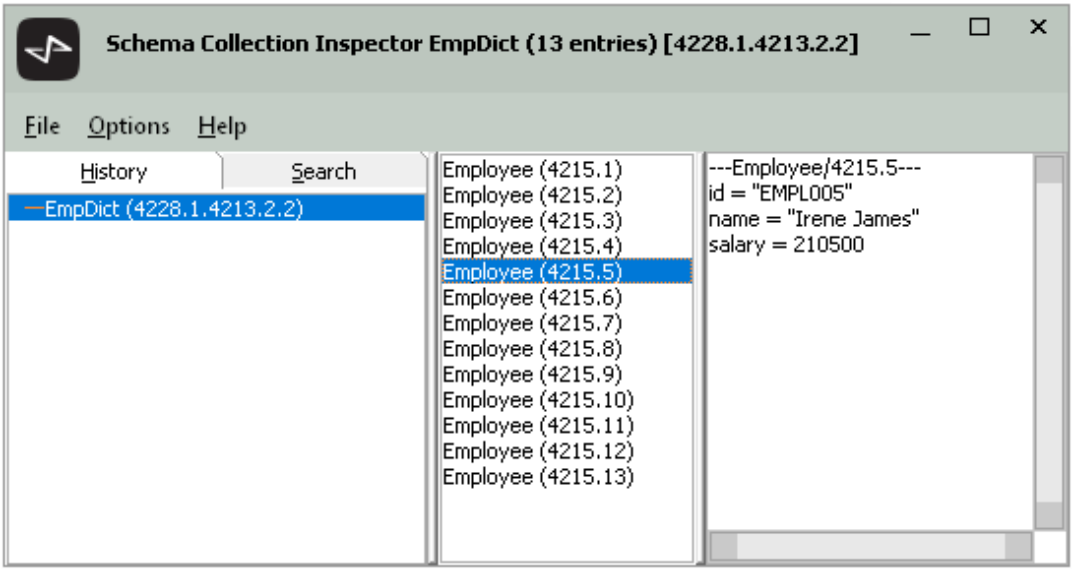
5. Double click the **Company** in the Schema Collection Inspector form to display the Schema Inspector for Company form and then inspect the **allMyCustomers** and **allMyEmployees** collections.

   There should be 12 **Customers** and 13 **Employees**.

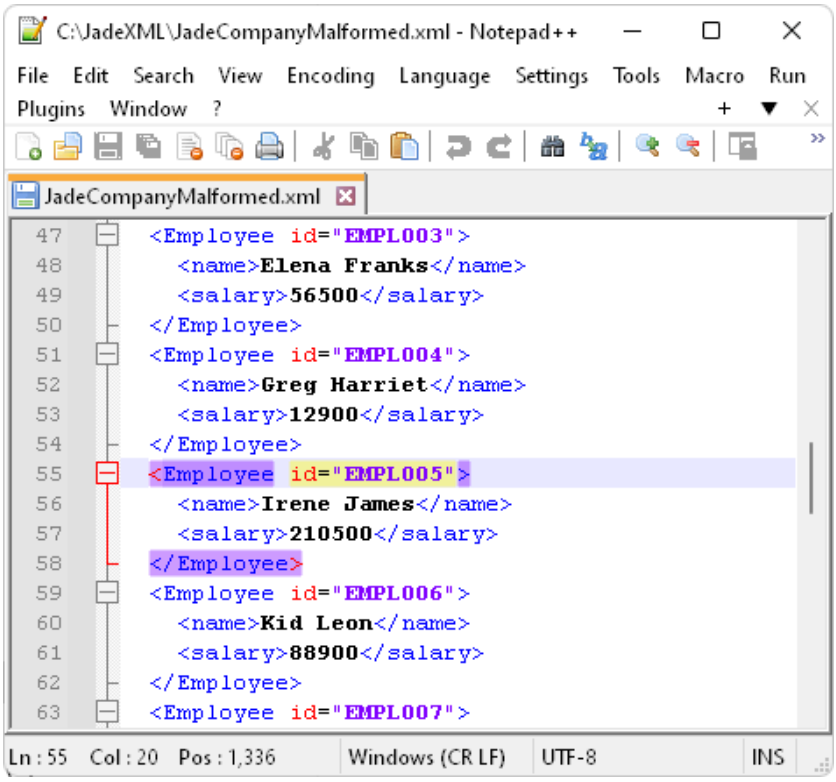


6. Double-click the **allMyEmployees** collection in the Schema Inspector form to display the Schema Collection Inspector for allMyEmployees form.

7. Click on each **Employee** to verify that the details match those in the **JadeCompanyMalformed.xml** file.



The following is the **JadeCompanyMalformed.xml** file.



8. Do the same for each **Customer** in the **allMyCustomers** collection.

Persistent XML

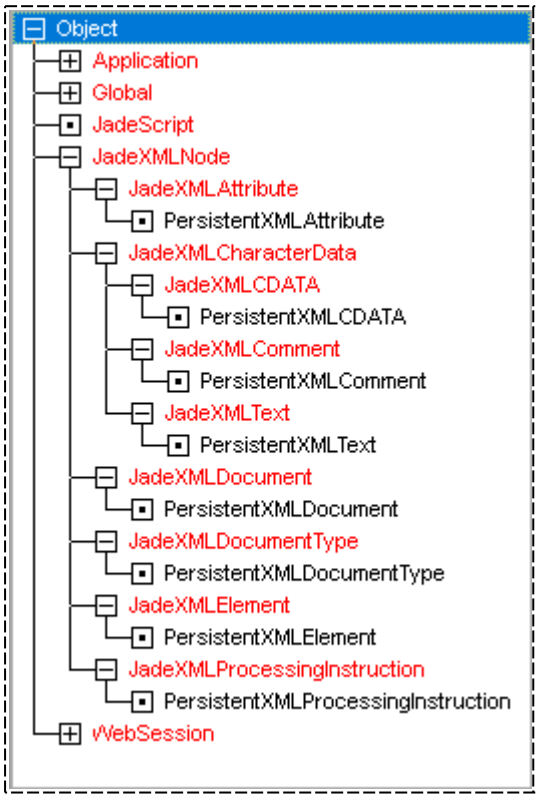
When parsing an XML document in Jade, the **JadeXMLDocument** object and the nodes created with its **parseFile** and **parseString** methods are transient by default, which means that the XML must be parsed again each time it is to be used.

However, Jade provides the **JadeXMLDocumentParser** class that can create persistent implementations of the **JadeXMLDocument** tree structure, using its **parseDocumentString** and **parseDocumentFile** methods.

To create an XML tree structure in the Jade Platform, the following node classes are required, all of which have transient-only persistence.

Class	Represents...
JadeXMLAttribute	An attribute of an XML element.
JadeXMLCDATA	An XML escape character (for example, a **<** character that is not part of a tag).
JadeXMLComment	An XML comment.
JadeXMLDocument	The XML document itself.
JadeXMLDocumentType	The document type declaration in an XML document tree.
JadeXMLElement	An element in an XML document; for example, **<name></name>**.
JadeXMLProcessingInstruction	An XML processing instruction (that is, an application-specific instruction on how to handle an XML document after the document has been parsed).
JadeXMLText	The text contained within an XML element when that element contains both text and other elements.

While the classes in the above table are transient only, persistent subclasses are allowed. To allow for the creation of persistent **JadeXMLDocuments**, you must first create persistent subclasses of each of these transient node classes. For example, the following class hierarchy shows a user subclass for each of the required classes.



When the persistent subclasses are established, the **JadeXMLDocumentParser** provides the **setClassMapping** method, which takes two parameters: a node class and the persistent user subclass to map to.

It also provides the **parseDocumentString** method, which takes a **JadeXMLDocument** (which can be transient, or a persistent subclass) and an XML string. The **parseDocumentFile** method is the same as the **parseDocumentString** method except for a file path string instead of an XML string.

For example, the following code will establish all required mappings, then create a persistent **JadeXMLDocument** using the **parseDocumentString** method of **JadeXMLDocumentParser**.

```
generatePersistentXMLExample();

vars
    xmlParser   : JadeXMLDocumentParser;
    xmlDocument : PersistentXMLDocument;
    xmlString   : String;
begin
    create xmlParser transient;
    xmlParser.setClassMapping(JadeXMLAttribute, PersistentXMLAttribute);
    xmlParser.setClassMapping(JadeXMLCDATA, PersistentXMLCDATA);
    xmlParser.setClassMapping(JadeXMLComment, PersistentXMLComment);
    xmlParser.setClassMapping(JadeXMLText, PersistentXMLText);
    xmlParser.setClassMapping(JadeXMLDocumentType, PersistentXMLDocumentType);
    xmlParser.setClassMapping(JadeXMLElement, PersistentXMLElement);
    xmlParser.setClassMapping(JadeXMLProcessingInstruction, PersistentXMLProcessingInstruction);

    xmlString := "<SomeTag>Some Data</SomeTag>";

    beginTransaction;
    create xmlDocument persistent;
    xmlParser.parseDocumentString(xmlDocument, xmlString);
    commitTransaction;
end;
```

Note All mappings must be set, regardless of whether they are needed for the specific XML being parsed. The **parseDocumentString** and **parseDocumentFile** methods will generate an 8909 exception (*XML class mapping is invalid*) if any are missing. However, the **JadeXMLDocument** subclass does not need to be mapped, as it is passed as a parameter to the method.

Exercise 5 – Creating Persistent JadeXMLNode Subclasses

In this exercise, create persistent subclasses for each of the required nodes in a persistent **JadeXMLDocument**.

1. Create a new schema called **PersistentXMLSchema**.
2. Open the **PersistentXMLSchema** in the Class Browser.
3. With focus on the **Object** class, press the F4 shortcut key to display the Find Type dialog.
4. Search for **JadeXMLAttribute** and then click the **Current Browser** button (or press Enter) to add it to the displayed classes in the Class Browser.
5. Add a subclass to **JadeXMLAttribute** called **PersistentXMLAttribute**.
6. Search for each of the following classes and then add the corresponding persistent subclass.

Class	Subclass
JadeXMLCDATA	PersistentXMLCDATA
JadeXMLComment	PersistentXMLComment
JadeXMLDocument	PersistentXMLDocument
JadeXMLDocumentType	PersistentXMLDocumentType
JadeXMLElement	PersistentXMLElement

Class	Subclass
JadeXMLProcessingInstruction	PersistentXMLProcessingInstruction
JadeXMLText	PersistentXMLText

Exercise 6 – Parsing an XML Document Persistently

In this exercise, use the **JadeXMLDocumentParser** to load an XML file and create a persistent XML tree and then inspect that XML tree using the Schema Inspector form.

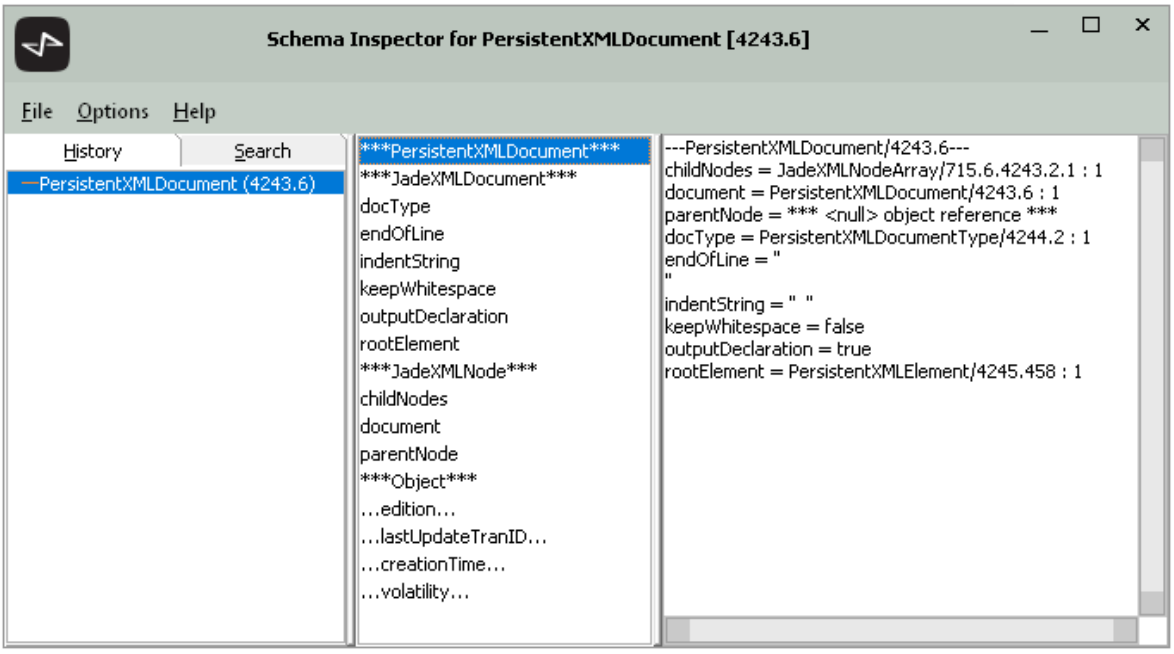
1. Search for the **JadeXMLDocumentParser** class using the Find Type dialog.
2. Add a method called **establishMappings** to the **JadeXMLDocumentParser** class and code it as follows.

```
establishMappings() updating;

begin
    self.setClassMapping(JadeXMLAttribute, PersistentXMLAttribute);
    self.setClassMapping(JadeXMLCDATA, PersistentXMLCDATA);
    self.setClassMapping(JadeXMLComment, PersistentXMLComment);
    self.setClassMapping(JadeXMLText, PersistentXMLText);
    self.setClassMapping(JadeXMLDocumentType, PersistentXMLDocumentType);
    self.setClassMapping(JadeXMLElement, PersistentXMLElement);
    self.setClassMapping(JadeXMLProcessingInstruction, PersistentXMLProcessingInstruction);

end;
```

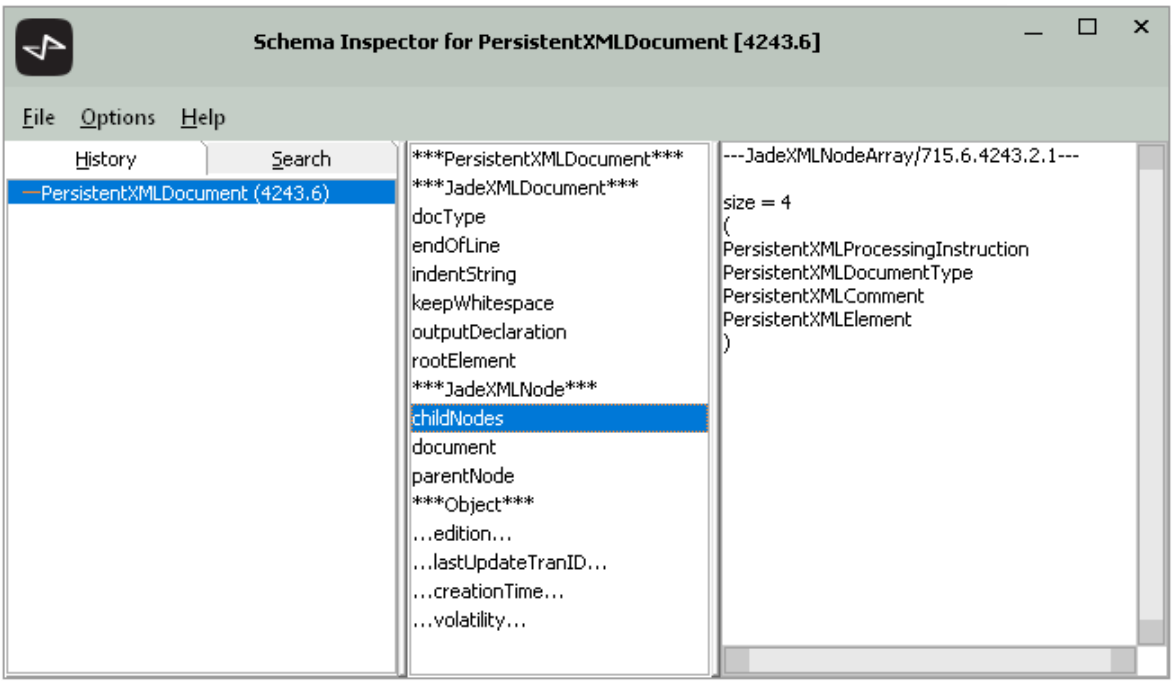
3. Create a **JadeScript** class method called **loadPersistentXML** and code it as follows.

```
loadPersistentXML();

vars
    xmlParser   : JadeXMLDocumentParser;
    xmlDocument : PersistentXMLDocument;
begin
    create xmlParser transient;
    xmlParser.establishMappings();

    beginTransaction;
    create xmlDocument persistent;
    xmlParser.parseDocumentFile(xmlDocument, "C:\JadeXMLWhitePaper\Persistent Example.xml");
    commitTransaction;
end;
```

4. Run the **loadPersistentXML** method, select the **PersistentXMLDocument** class in the Class Browser, and then press Ctrl+I to open it in the Schema Inspector form.



5. Double-click the **PersistentXMLDocument** in the Schema Inspector form and ensure that the **ChildNodes** collection contains the correct child nodes, as follows.



6. Navigate around the Jade XML tree.

   You should see that all the nodes from the **JadeXMLExample.xml** file are contained within the persistent tree in the appropriate structure.