JADE Developer's Course

VERSION 2018

jade

Contents

Overview

The course is a five-day course aimed at people wanting to learn how to develop systems in JADE. There are no prerequisites, although experience in developing in another language would help.

The schedule is as follows.

- Monday
  - Module 1 - Installing JADE
  - Module 2 - Schemas
  - Module 3 - JadeScripts
  - Module 4 - Application Object
  - Module 5 - Primitive Types
  - Module 6 - Classes
- Tuesday
  - Module 6 - Classes
  - 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
- Friday
  - Module 14 - Notifications and Timers
  - Module 15 - Nodes, Processes, and Caches
  - Module 16 - Transactions and Locking
  - Module 17 - Printing

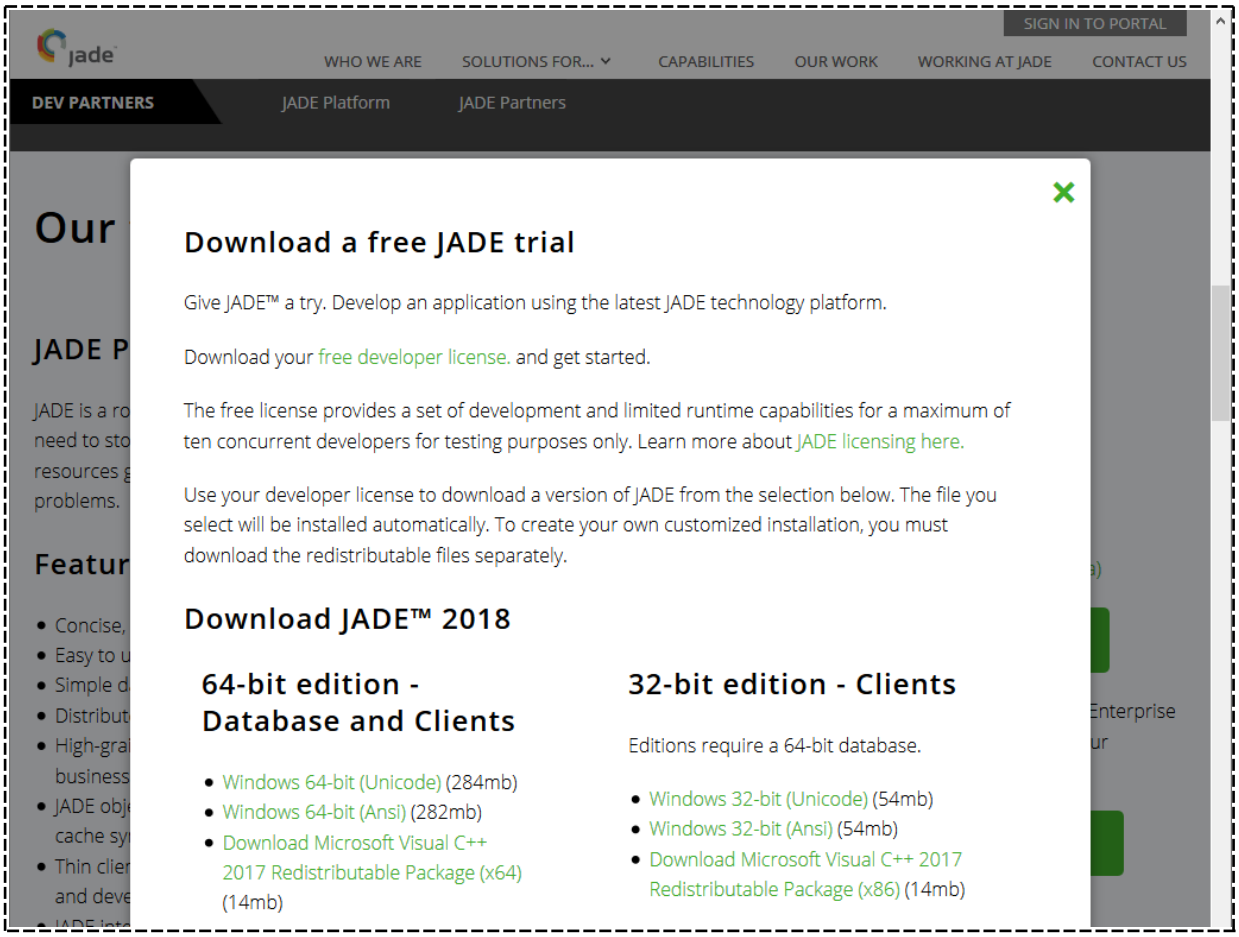
At the end of each module, there are a number of exercises for you to practice to build your skills. The exercises enable you to build a simplified banking system, which despite its simplicity, demonstrates many of the important features of JADE.

Module 1 Installing JADE

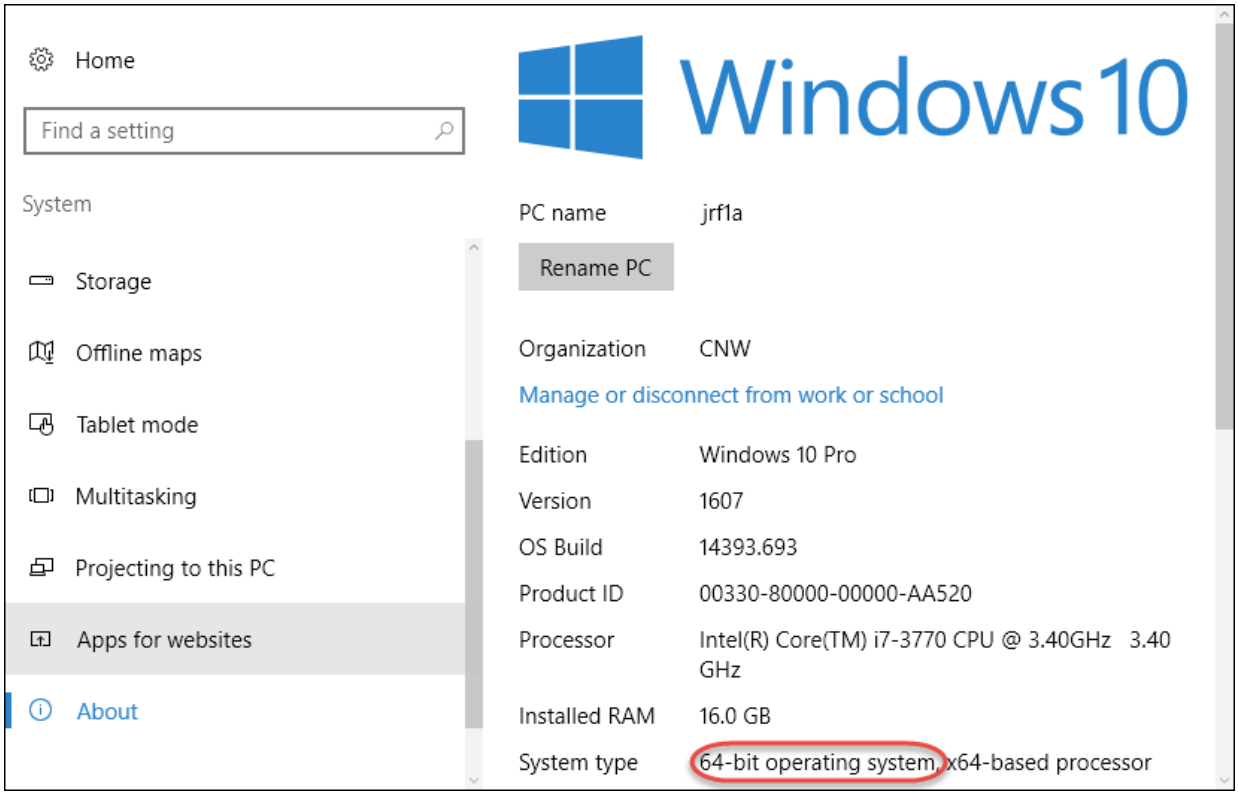
This module contains the following topics.

Introduction

You can download JADE software and obtain a free developer license from the JADE web site, at https://www.jadeworld.com/solutions-for/dev-partners/.



You require the JADE 64-bit version for this course. You can determine your operating system from the System **About** settings or the Control Panel, depending on your operating system, to check that you are running 64-bit Windows.



Install the latest JADE 2018 release.

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

Exercise 1.1 – Installing JADE

Follow these instructions to install JADE on your PC or laptop.

1. Request a free developer license by pasting https://www.jadeworld.com/developer-center/download-jade into your browser and then clicking on **free developer license**. A form is then 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 JADE.

2. On the https://www.jadeworld.com/solutions-for/dev-partners/ web page, download the 64-bit ANSI version of JADE 2018 for database and clients (that is, the **JADEwin64Ansi.exe** file).

3. Download the **JADE 2018 - Documentation Package** (the **JADEWinDocs.exe** file) from https://www.jadeworld.com/developer-center/resource-library/.

4. Run the **JADEwin64Ansi.exe** setup program and complete the steps of the installation with the actions specified in the following table. (The steps in this instruction are based on the Windows 10 operating system and the Mozilla Firefox browser.)

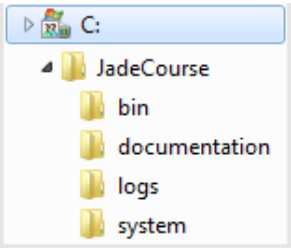
Step	Action
Welcome	Click the **Next** button.
License Agreement	Click the **Yes** button, to agree to the terms of the license.
Installation Type	Select the **Fresh Copy** option, and then click the **Next** button.
Setup Type	Select the **Development** option, and then click the **Next** button.
User 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. Run the **JADEWinDocs.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 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.)

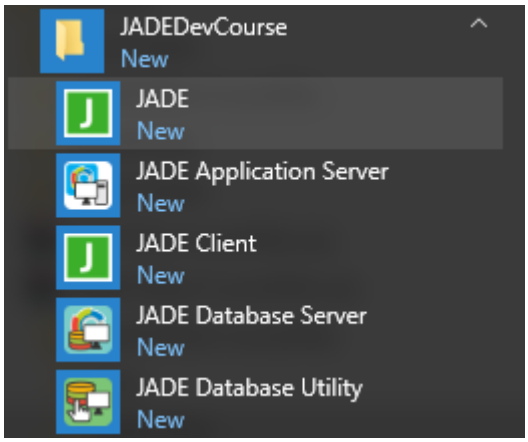
The **logs** folder contains the JADE message log file (**jommsg.log**) and error log files.

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

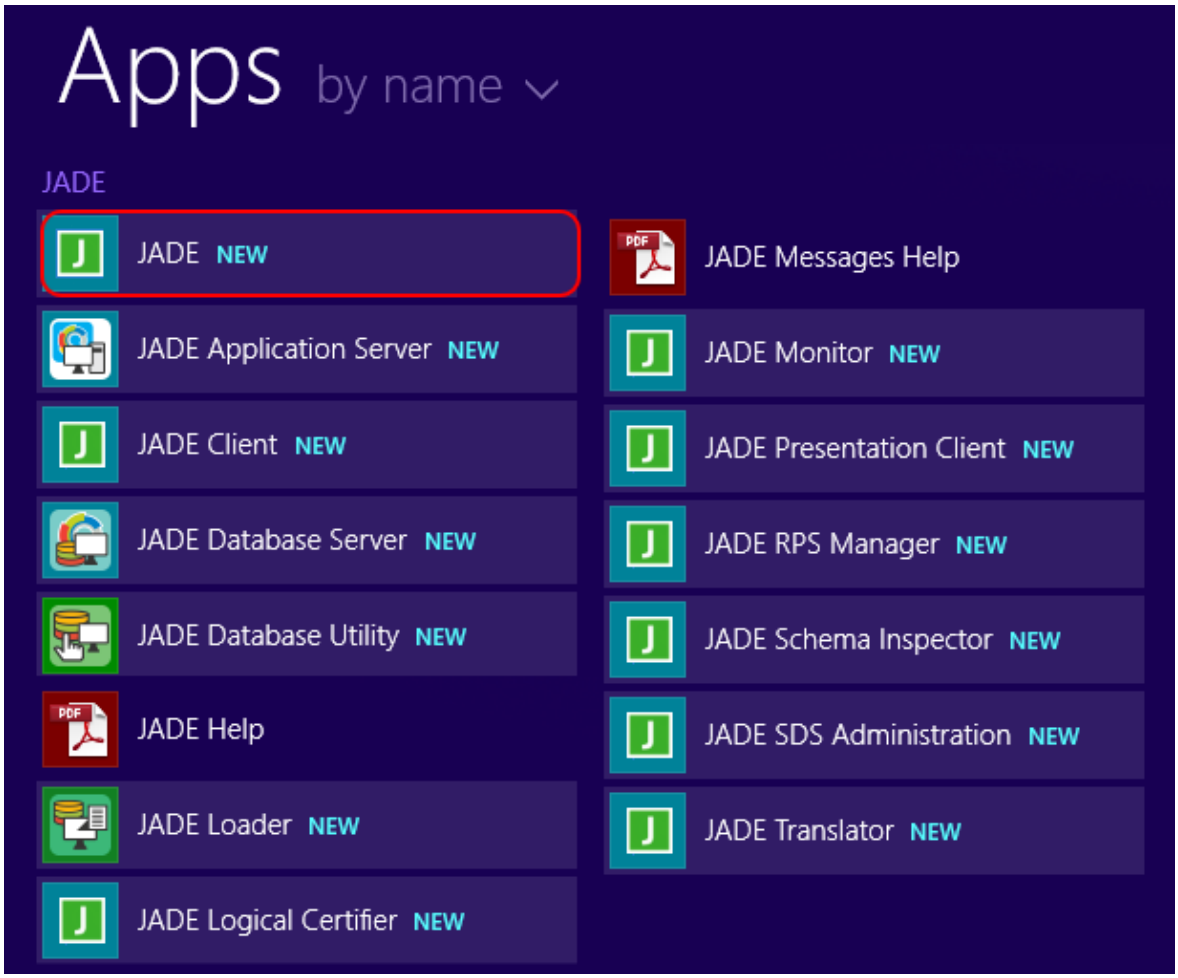
Running JADE in Single User Mode

When you run JADE 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 JADE in single user mode by selecting the **JADE** shortcut from the menu.



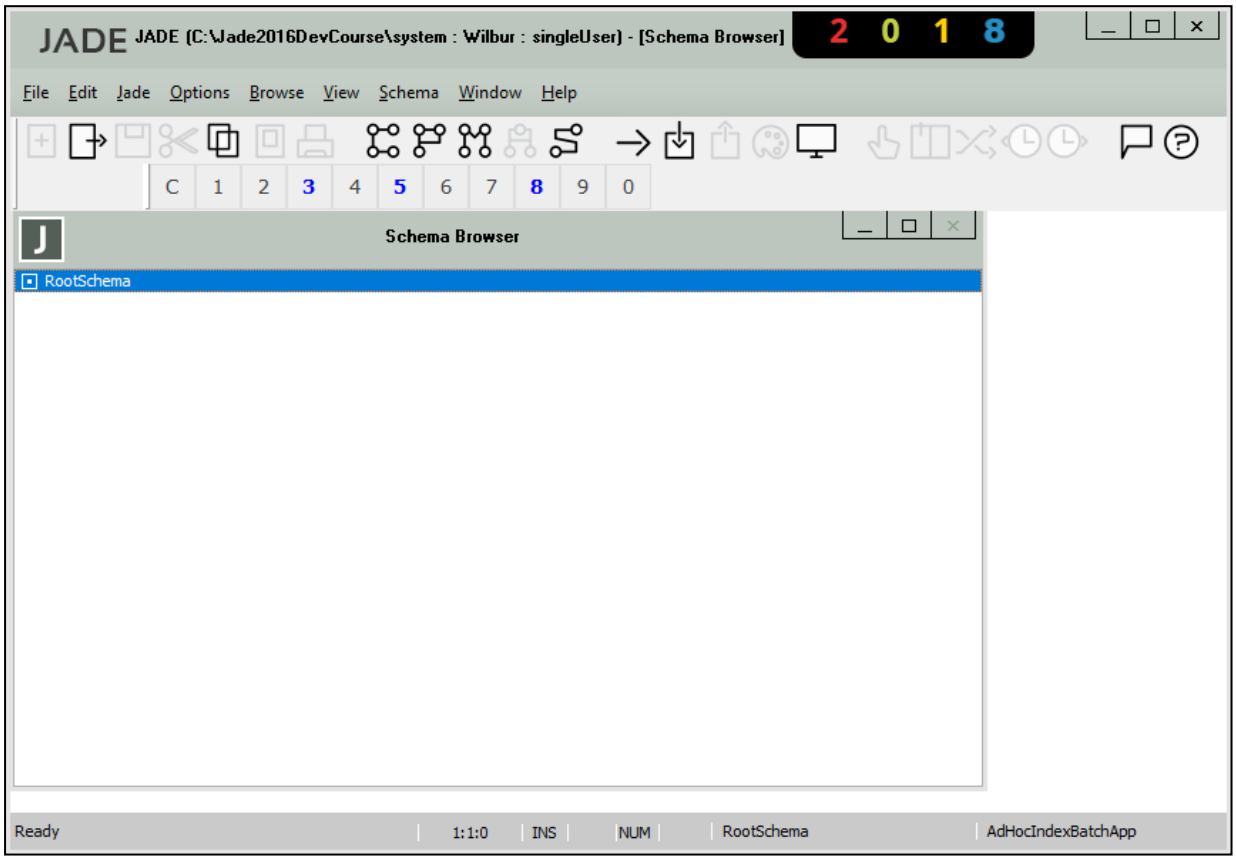
If you are using the Metro interface in a Windows release earlier than Windows 10 (in which it is no longer available), select the first shortcut.



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 **User ID** text box, select the **Browse Classes** option, and then click the **OK** button.

Close the **Jade Release Note**, **Tip of the Day**, and **Start** dialogs. You are now in the JADE development environment, with the Schema Browser displayed.



The JADE 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 development environment enables you to define classes, JADE methods, properties, constants, conditions, and form definitions. (For details, see Chapters 1 through 5 in the *JADE Development Environment User's Guide*; for example, the 2018 product information is available from https://www.jadeworld.com/docs/jade-2018/Default.htm.)

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:

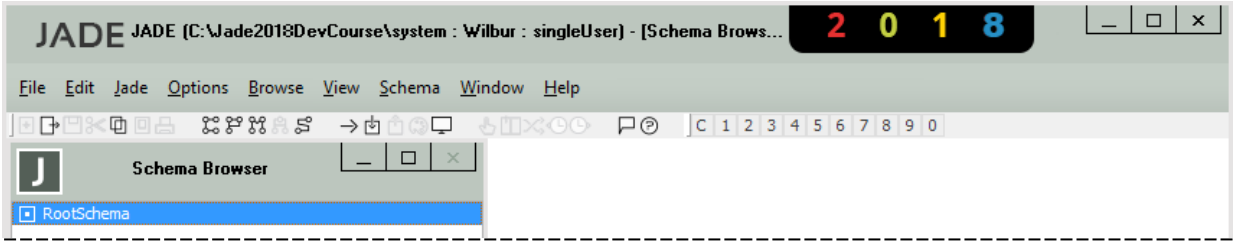
- Define new methods or conditions in the selected class, primitive type, or interface
- Maintain existing methods and conditions using the integrated editor pane in a browser
- Compile methods and conditions
- Execute methods in the **JadeScript** class of the Class Browser (if selected)
- Change or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

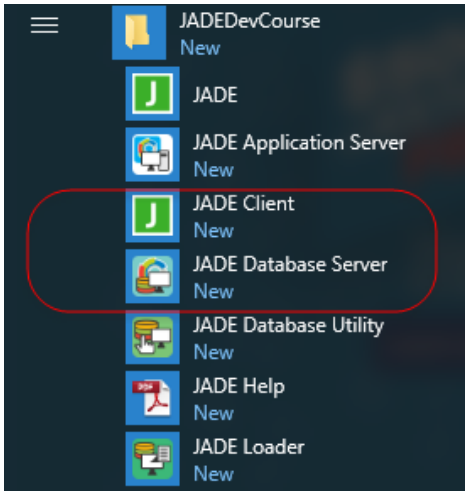
To hide the display of the editor clipboard toolbar or the floated Jade Clipboard Text Contents form, uncheck the **Show Clip Board Toolbar** check box on the **Window** sheet of the Preferences dialog or select the **Show Clipboard Toolbar** command in the View menu. If the editor clipboard toolbar is docked in the toolbar of the main development environment window, hiding the main development environment window toolbar also hides the editor clipboard toolbar.

Tip You can also change the skin, by selecting the **Preferences** command from the Options menu, and then selecting the skin that you want to use (or **<None>**) in the **Select JADE Skin** combo box at the lower left of the **Window** sheet of the Preference dialog.

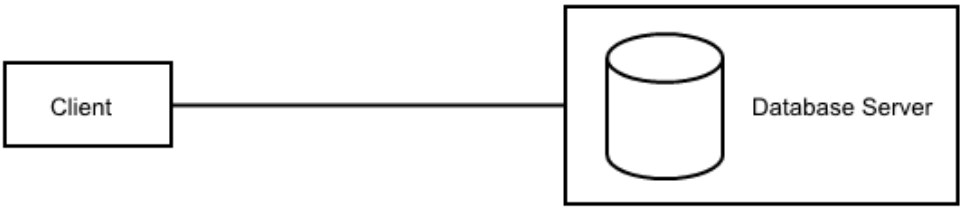
Running JADE in Multiuser Mode

When you run JADE 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.

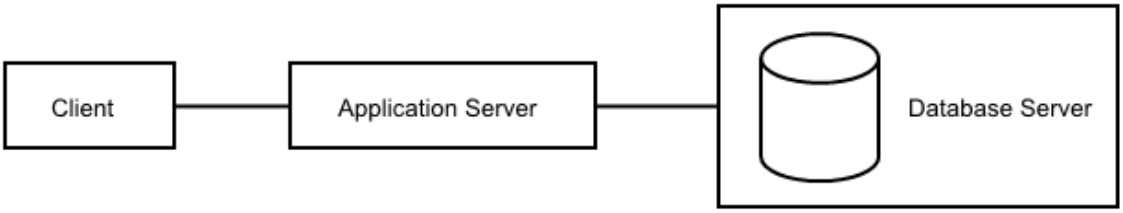
The JADE folder **(JADEDevCourse**, in this example) contains shortcuts for running JADE in multiuser mode.



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 diagram is an example of the maximized database server.

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

Exercise 1.2 – Running JADE

Run JADE 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 JADE. This is hardly surprising, as the JADE development environment *is* a JADE application.

Files for the Course

Copy the **Files** folder on the USB drive to **C:\JadeCourse\Files** on your PC or laptop.

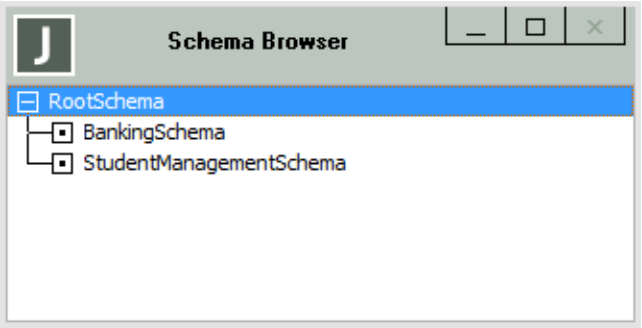
Module 2 Schemas

This module contains the following topics.

Introduction

Schemas provide a mechanism to organize classes. When you install JADE, 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.

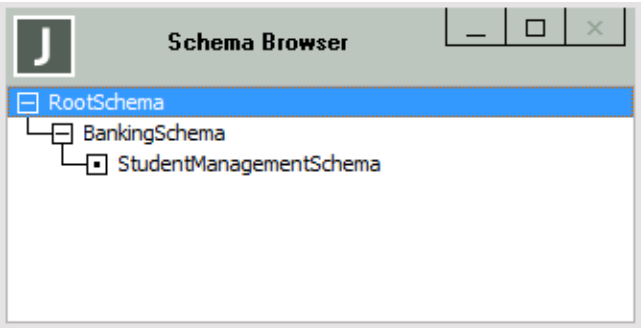
In the following diagram, a **BankingSchema** and a **StudentManagementSchema** have been added.



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 diagram.



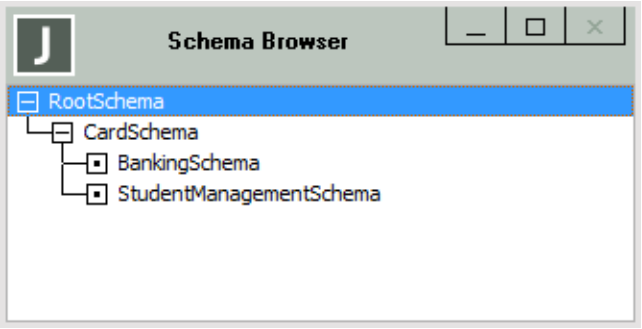
With this hierarchy, the **StudentManagementSchema** inherits all of the classes from the **BankingSchema** along with the system classes from **RootSchema**. This probably does not make a lot of sense.

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

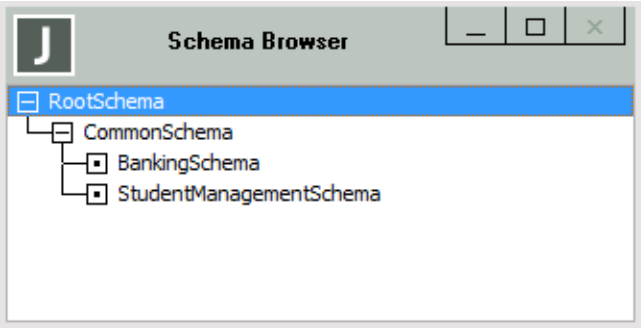
JADE Care is the group within Jade Software Corporation that develops tools to manage JADE systems (and other technologies).

JADE applications that are managed with JADE Care must have the JADE Care 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.ddb** 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.jadeworld.com/developer-center/jade-platform/jade-support/jade-care.

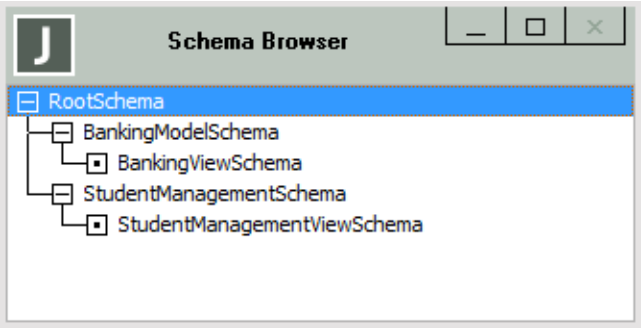
In the following schema hierarchy, **CardSchema** functionality is made available to the **StudentManagementSchema** and to the **BankingSchema**.



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



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

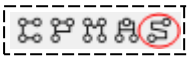
In the Schema Browser, when you select a schema to work with, you can then open other browser windows for that schema; for example, a Class Browser, which you can use for adding classes to the schema.

To open a Class Browser, click the **C** button from the JADE development environment toolbar.

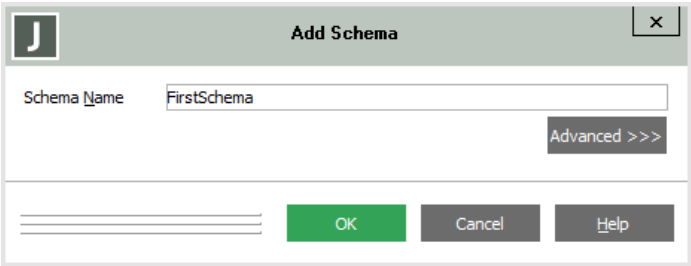
Exercise 2.1 – Adding a Schema

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 development environment toolbar.



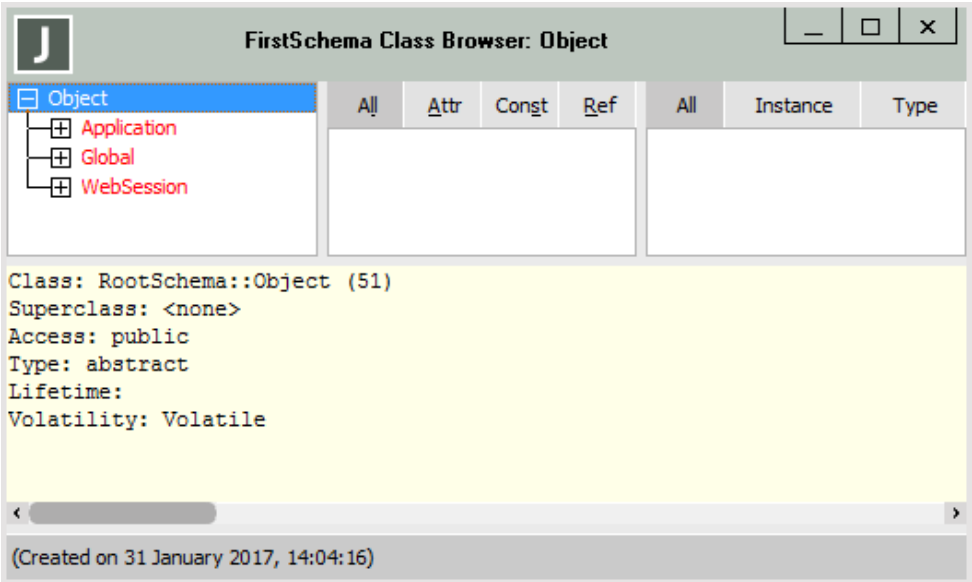
2. Select **RootSchema** in the Schema Browser.
3. Add a schema by selecting the Schema menu **Add** command.
4. Enter **FirstSchema** as the name of the schema, and then click the **OK** button.



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.

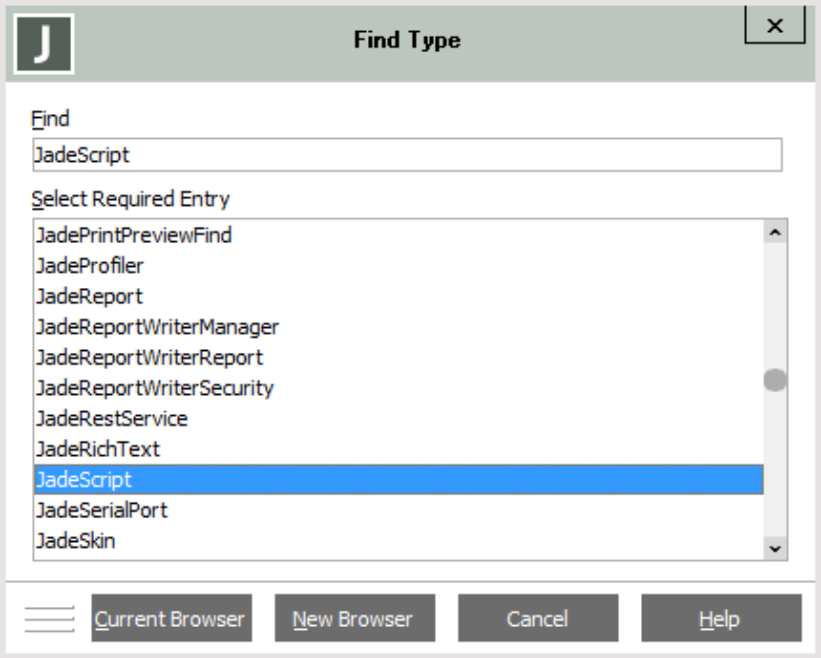
Introduction

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 development environment.

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

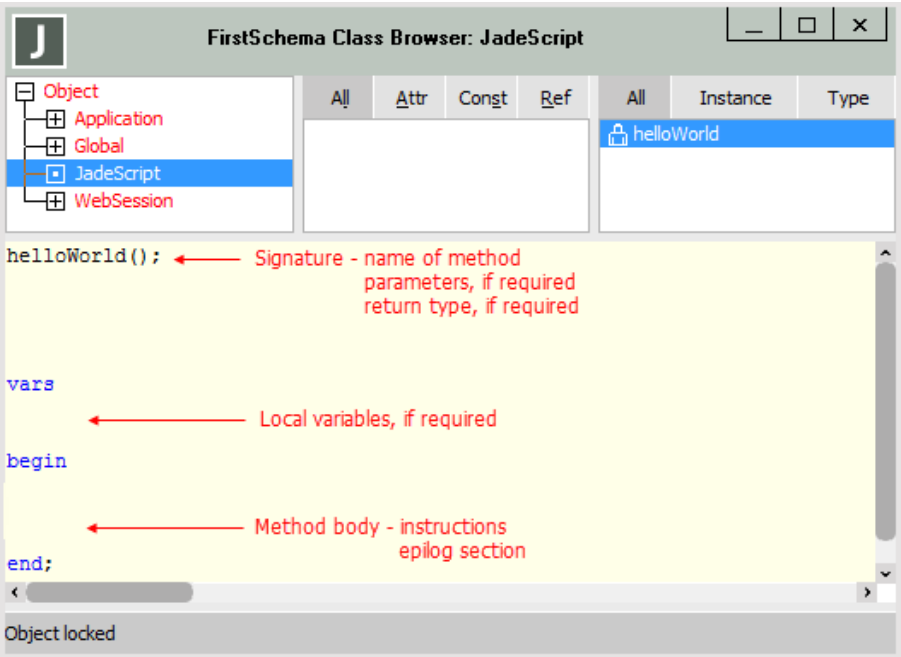
- Create, delete, and fix data
- Experiment, demonstrate, and test code

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.

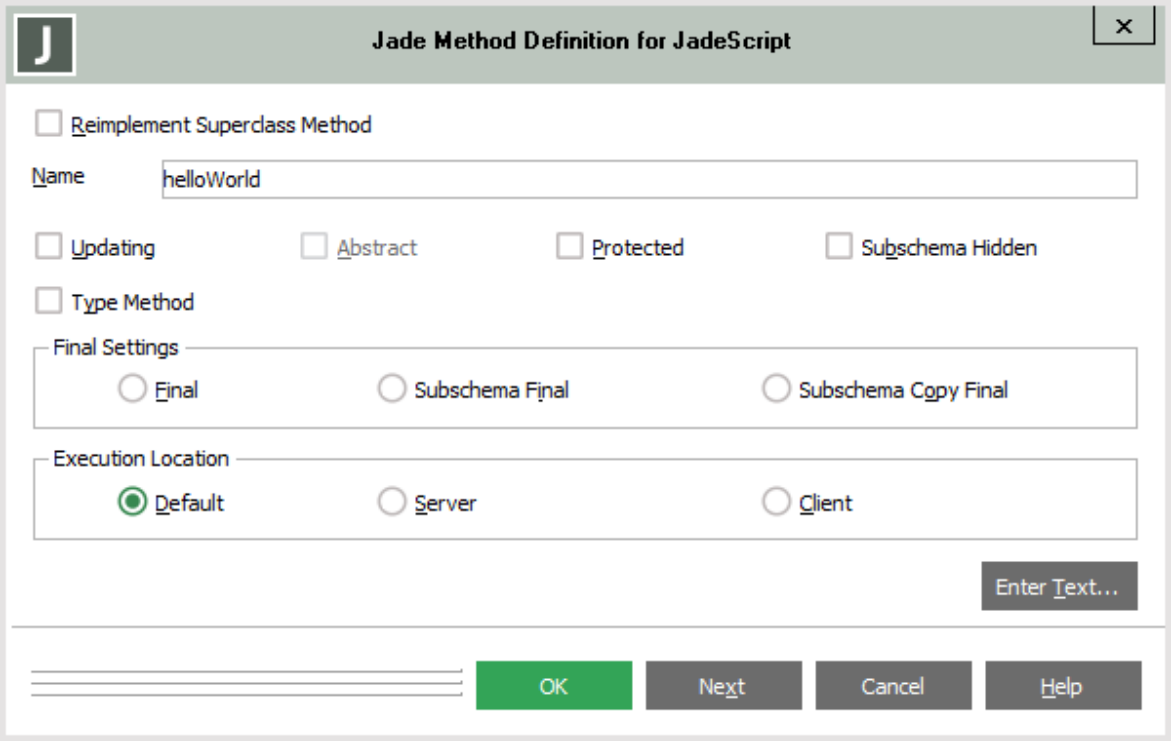
```
begin
    app.mousePointer := Window.MousePointer_HourGlass;
    // other instructions
epilog
    app.mousePointer := Window.MousePointer_Default;
end;
```

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.



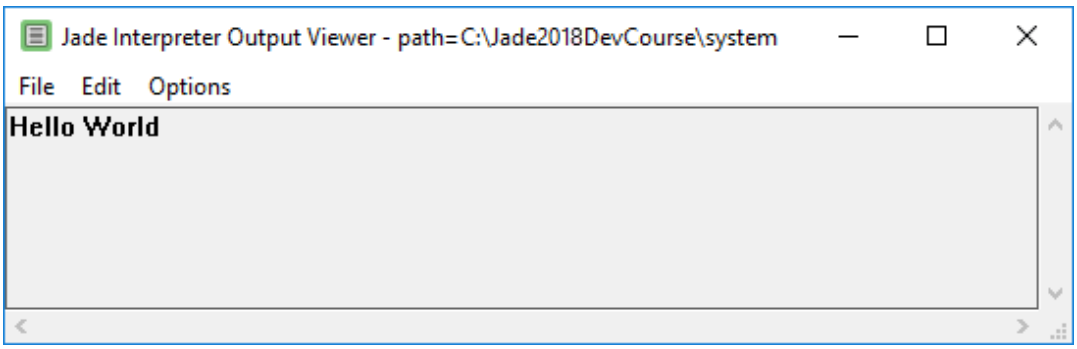
4. Enter the following code.

```
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.

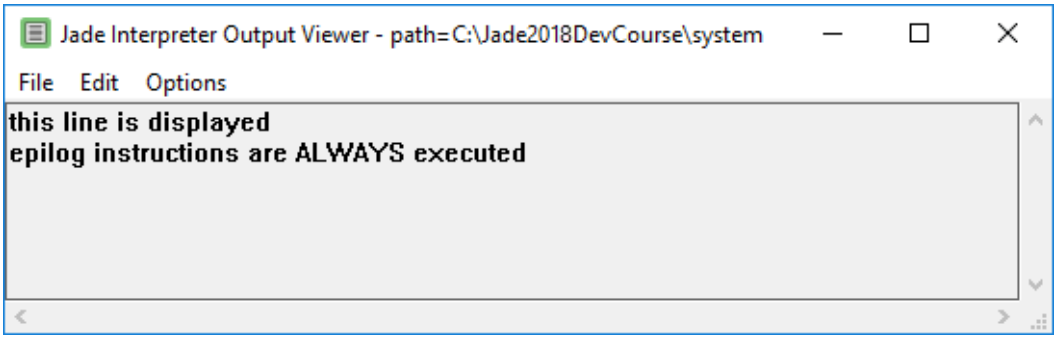
Exercise 3.3 – *return* and *epilog* Instructions

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

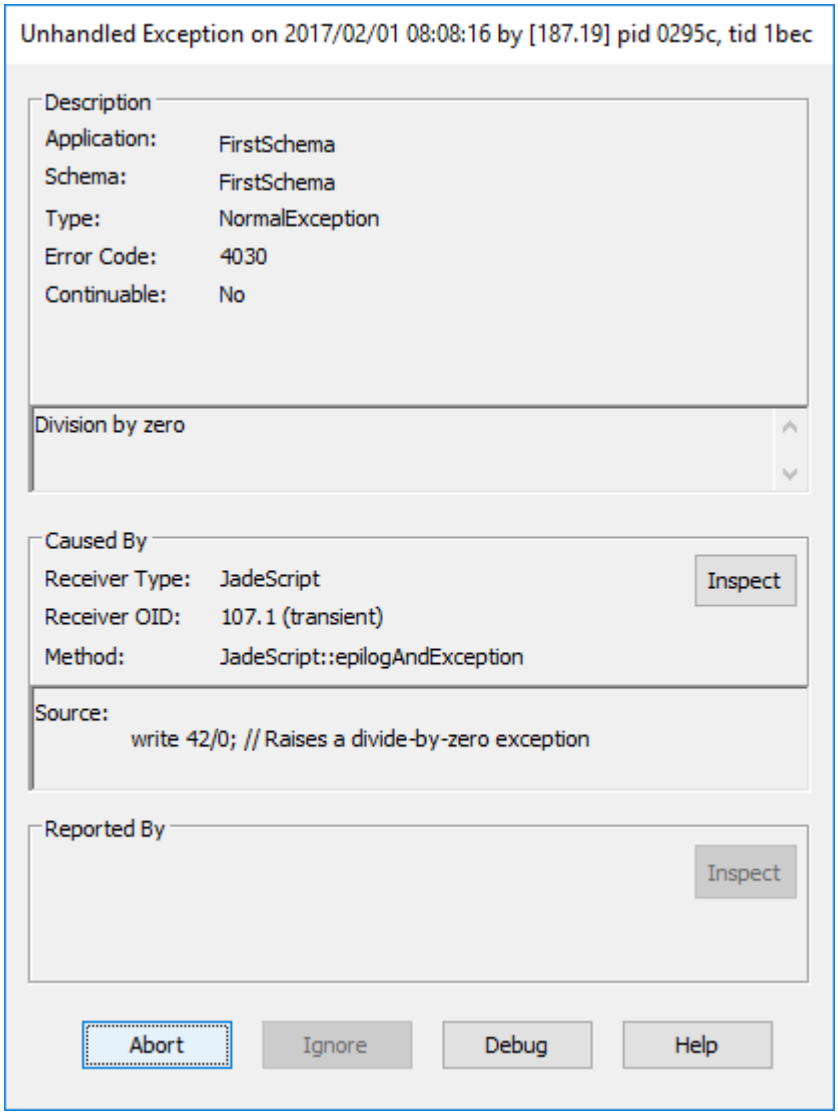
In this exercise, you will code an instruction that JADE cannot execute so that it therefore raises an exception.

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.

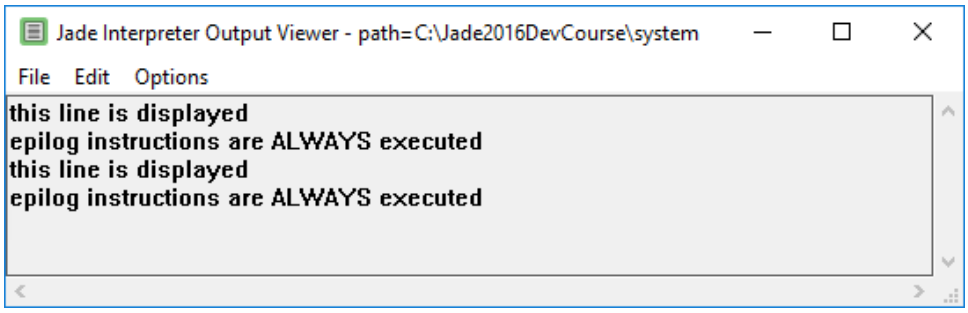
1. Create and execute an **epilogAndException** JadeScript method, as follows.

```
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 *Typecasting* 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.

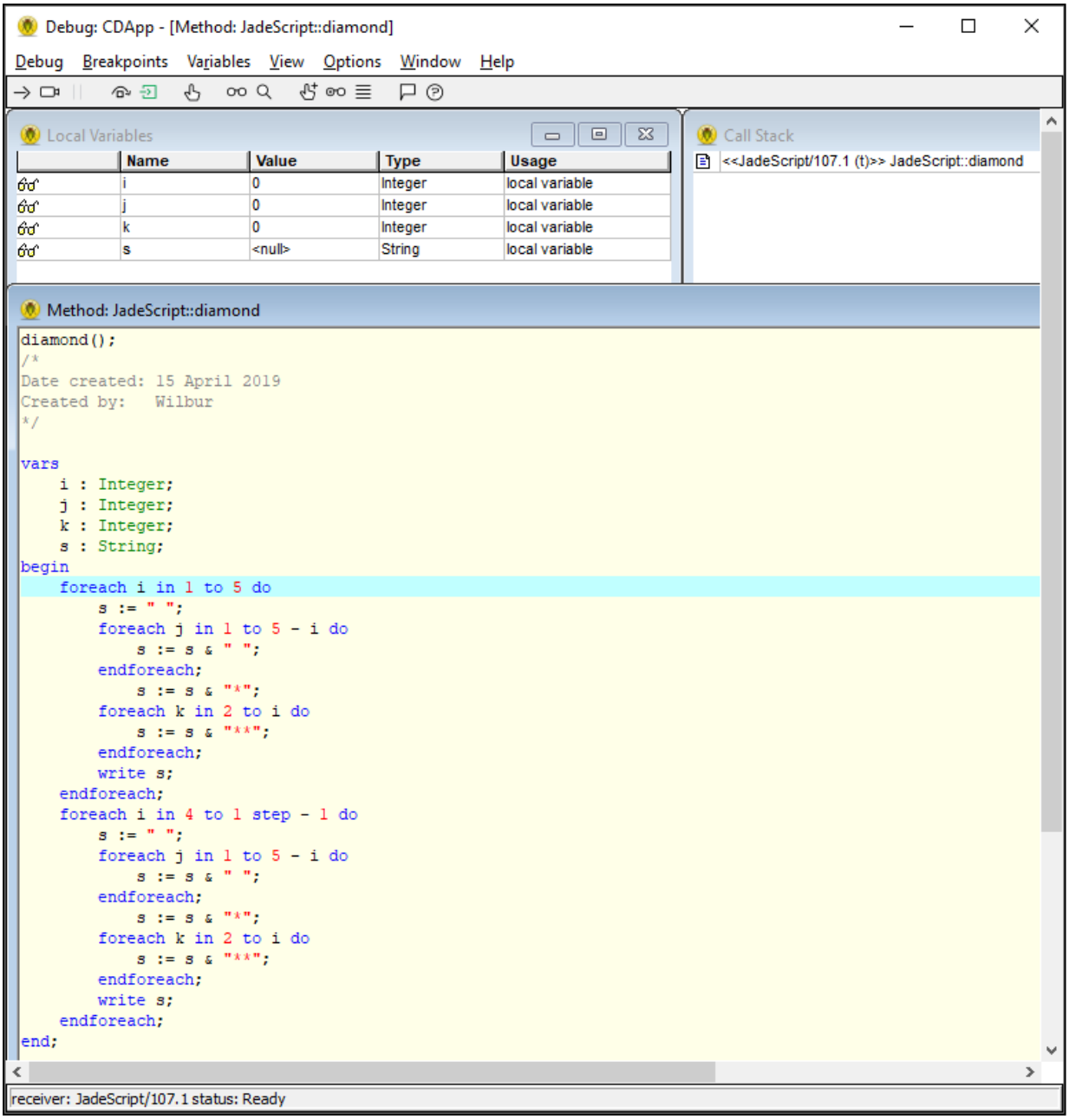
```
loopWithWhile();

vars
    name: String;
    i: Integer;
begin
    read name;
    while i < 10 do
        i := 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.



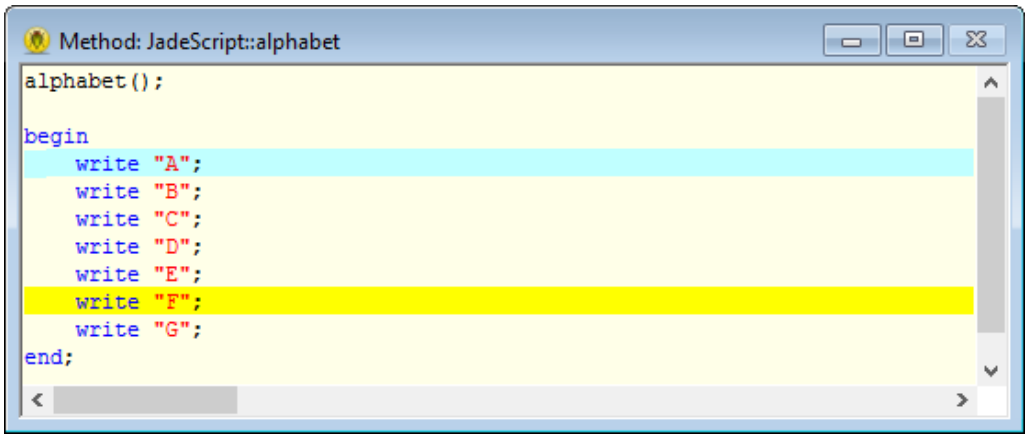
The debugger shows the method code with the next line of code to be executed highlighted with a blue background.

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. The line containing the cursor is highlighted with a yellow background, to indicate that it is a breakpoint.



For details about the debugger, see "Using the JADE Debugger", in Chapter 7 of the *JADE Development Environment User's Guide* (for example, at https://www.jadeworld.com/docs/jade-2018/Default.htm).

Write your own **JadeScript** method and debug it.

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).

```
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).

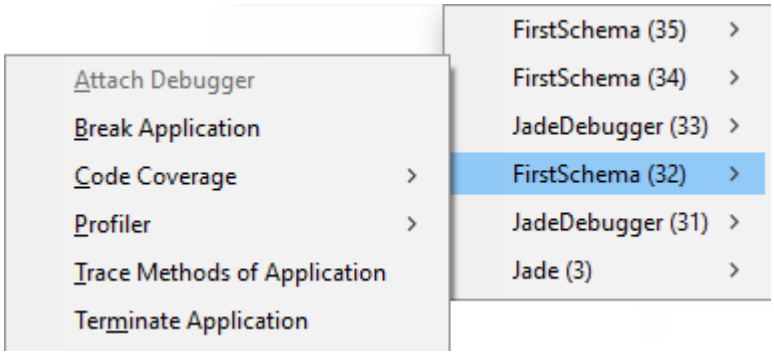
3. Select the Jade menu **Debug** command or press SHIFT+F9.

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**.

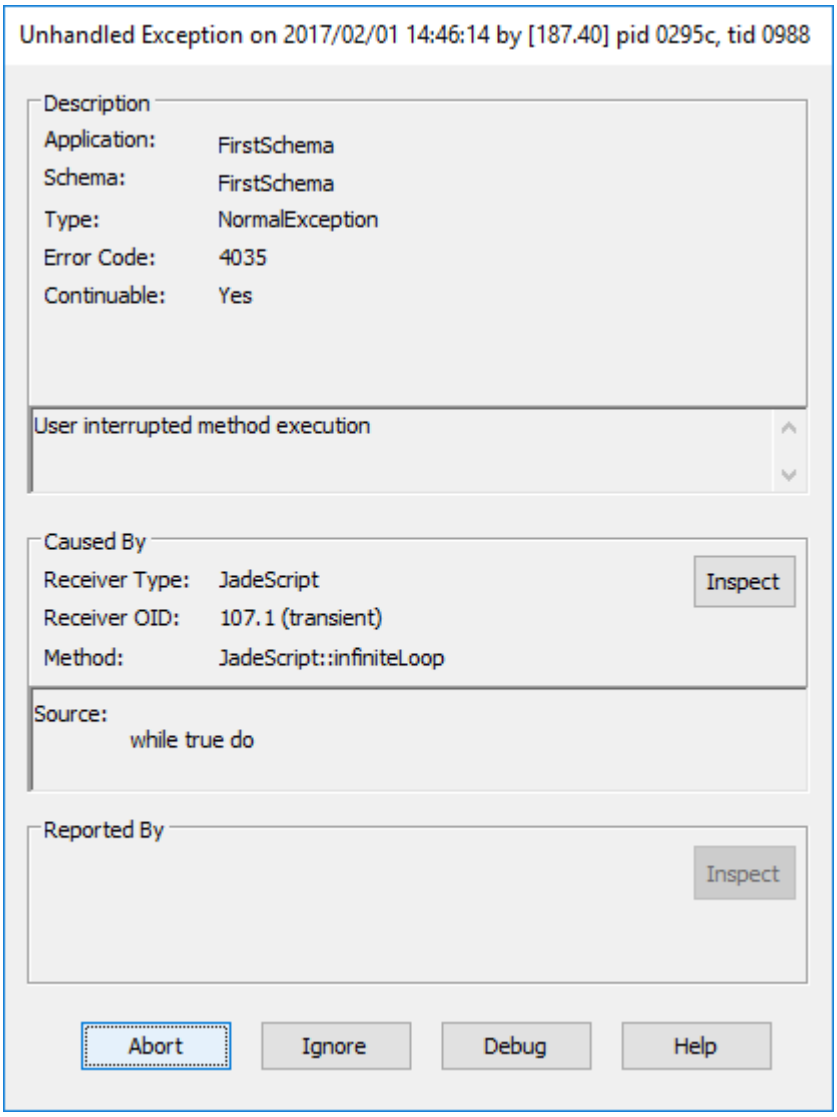
The command options that are available are as follows.

- **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.

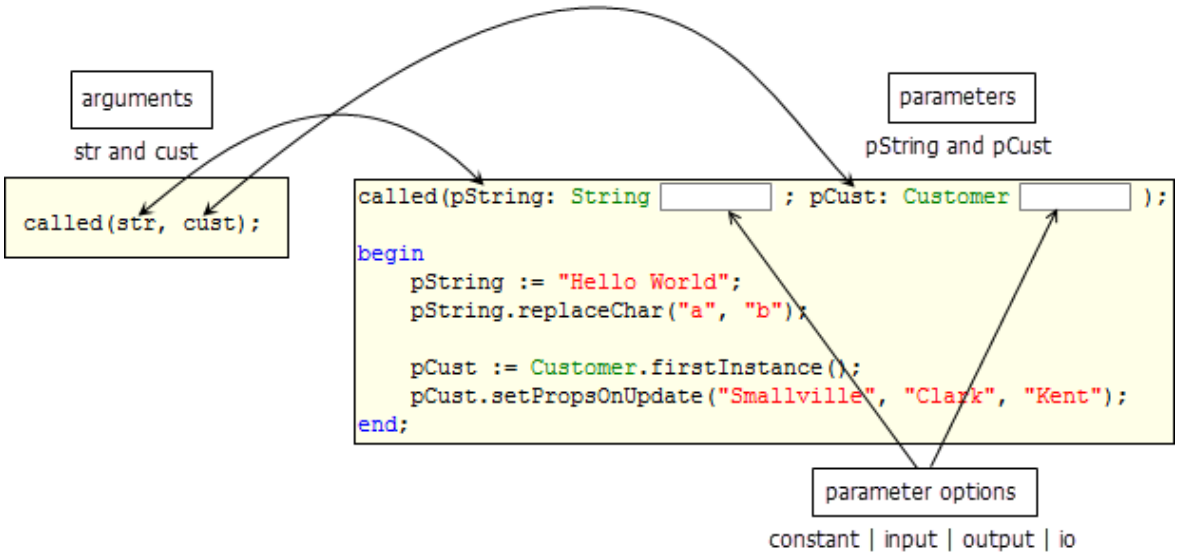
When you use the **Break Application** command, an exception dialog is displayed, enabling you to abort the action.



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 := 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:

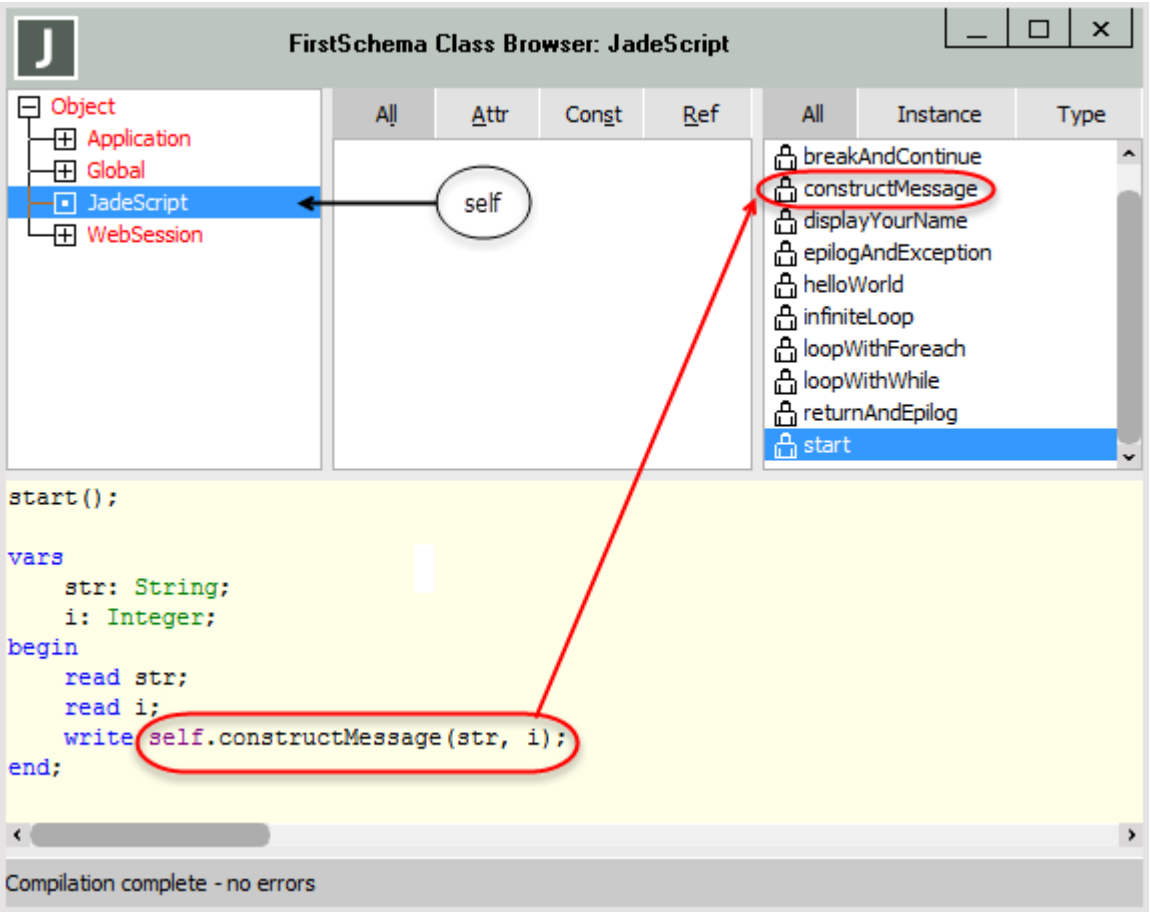
- The assignment operator (**:=**) is used.
- The variable **self** refers to the receiver; that is, the object for which the method is executing, which is a **JadeScript** object.

  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.

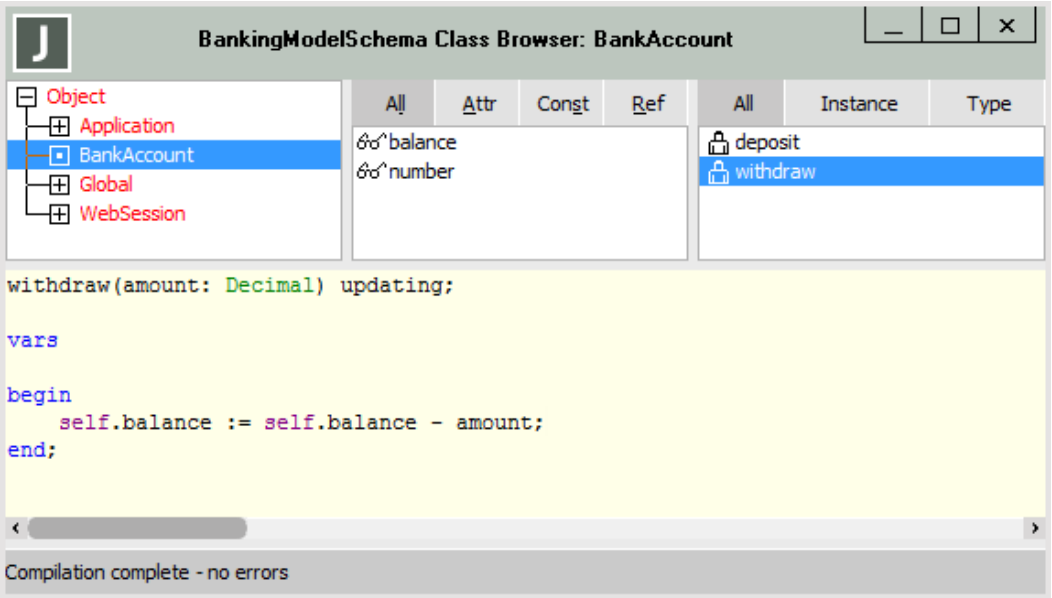
self Object

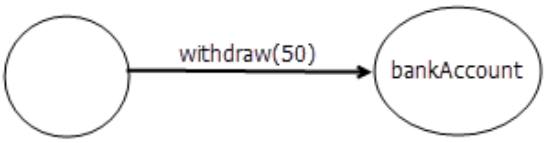
In the previous exercise, the **start** JadeScript method called the **constructMessage** JadeScript method (that is, a method in the same class), by sending a message to the **self** object.



In a later module, you will learn that a method being executed by an object can refer to properties of the object by using the **self** variable.

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.

```
withdraw(amount: Decimal) updating;

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

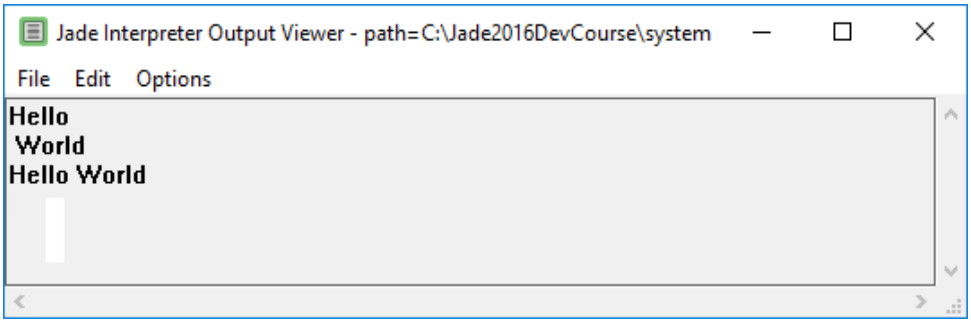
2. Add a JadeScript method called **threeHellos** that calls **threeWorlds**.

```
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"**.

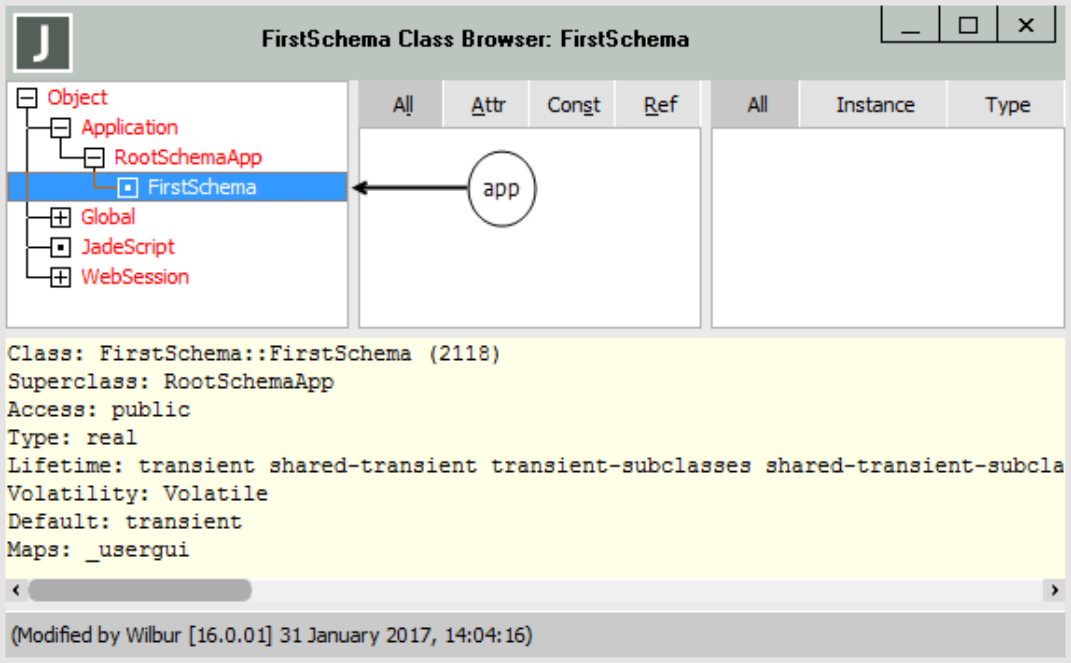
Module 4 Application Object

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.

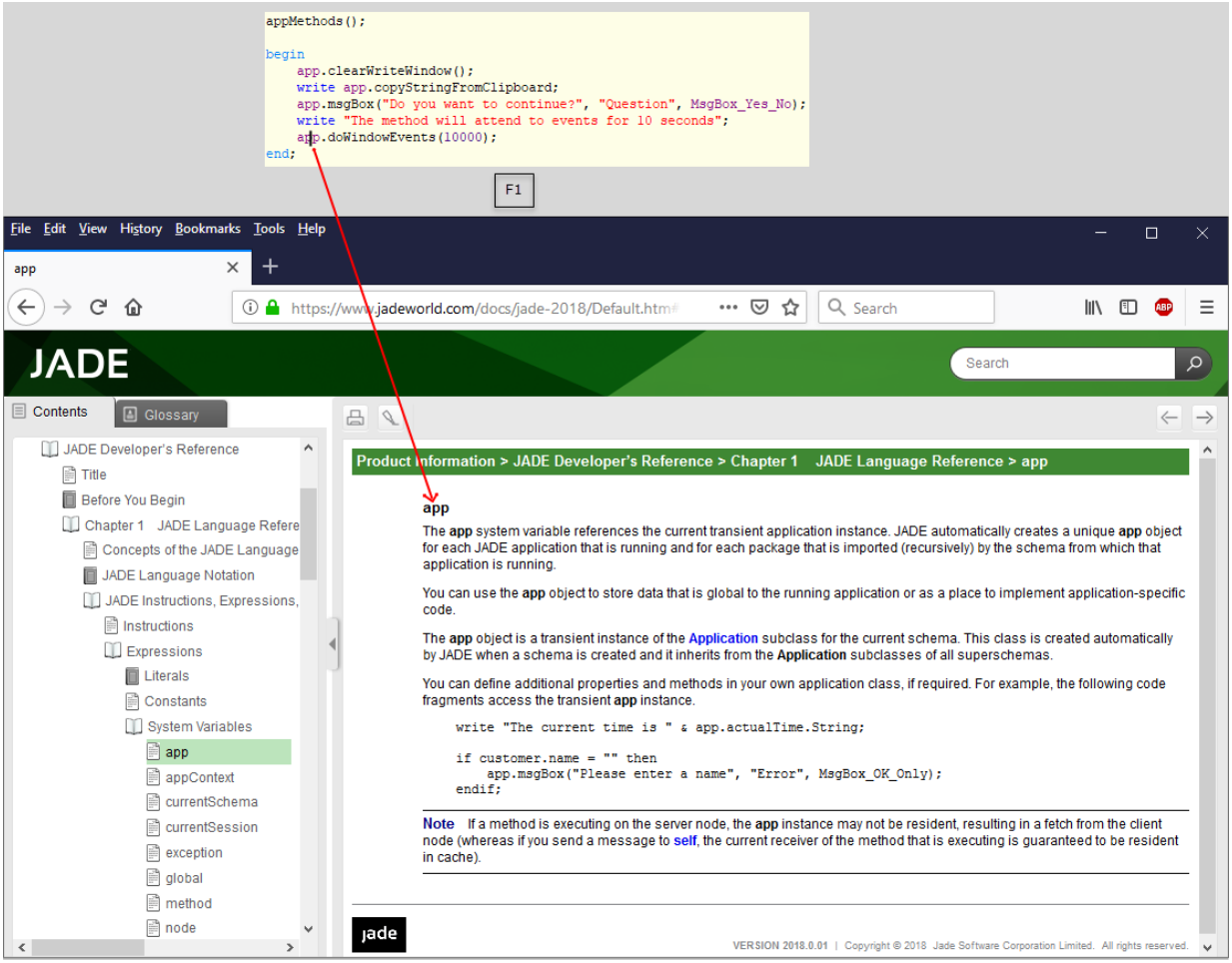
With the provision of the full product information library in both HTML5 (web) and PDF (print) format, by default, context-sensitive help is obtained from **.htm** topics in the HTML5 web format of the product information.

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://www.jadeworld.com/docs/jade-2018/Default.htm
# Where the .htm extension is used, .html is also valid.
```

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, position the cursor inside the word (for example, **app**) and then press F1 to open the web help or the relevant section of a Portable Document Format (PDF) file in Adobe Reader, as shown in the following diagram that accesses the topic in a web browser.



Troubleshooting PDF Context-Sensitive Help

If context-sensitive help to a PDF file is not working, check that Adobe Reader is installed. If it is installed but it is not working, the issue could be as described in a JADE forum (see https://forums.jadeworld.com/).

Adobe Reader XI and Context Help

POSTREPLY Search this topic... Search

Adobe Reader XI and Context Help

by **BeeJay** » Mon Dec 03, 2012 2:58 pm

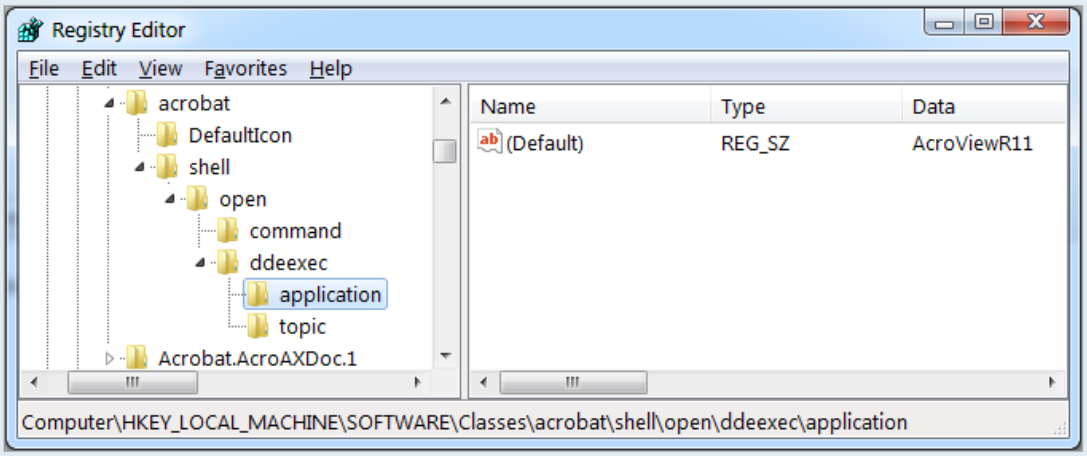
It would appear that Adobe have mucked up the DDE Server Name for Adobe Reader XI. According to Adobe's own documentation, the DDE Server Name should be AcroViewR11 but their Adobe Reader XI installer, whether it's upgrading a prior release of Adobe Reader or being installed on a machine that doesn't currently have Adobe Reader installed, results in a DDE Server Name of AcroViewR10. As their internal code only accepts a DDE Server Name of AcroViewR11, this means any application, including Jade, that tries to "link" to their DDE Server using the registry value to obtain the DDE Server name will fail.

Note: This issue will still affect Jade systems running on 6.3.09+ and 7.0.04+ releases, where Jade was changed to handle Adobe's new DDE Server Name naming convention that started with Adobe Reader X.

If you're happy with using the Windows Registry Editor, you can manually patch the value in the HKLM tree as follows:

CODE: SELECT ALL

```
HKEY_LOCAL_MACHINE
 |- SOFTWARE
   |-   Classes
      |-   acrobat
         |-   shell
            |-   open
               |-   ddeexec
                  |-   application
```



Adobe Reader registry patch.png (30.45 KiB) Viewed 1967 times

Hope this helps anyone else that may encounter this issue, until such time as Adobe fix their Installer.

Cheers,
BeeJay.

Note Before making changes in the Registry Editor (**regedit.exe**), you should select the File menu **Export** command, to extract all registry entries.

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 method names in the **appMethods** JadeScript method.
   - clearWriteWindow
   - clock
   - copyStringFromClipboard
   - dbPath
   - doWindowEvents
   - isValidObject
   - msgBox
   - random
   - userName

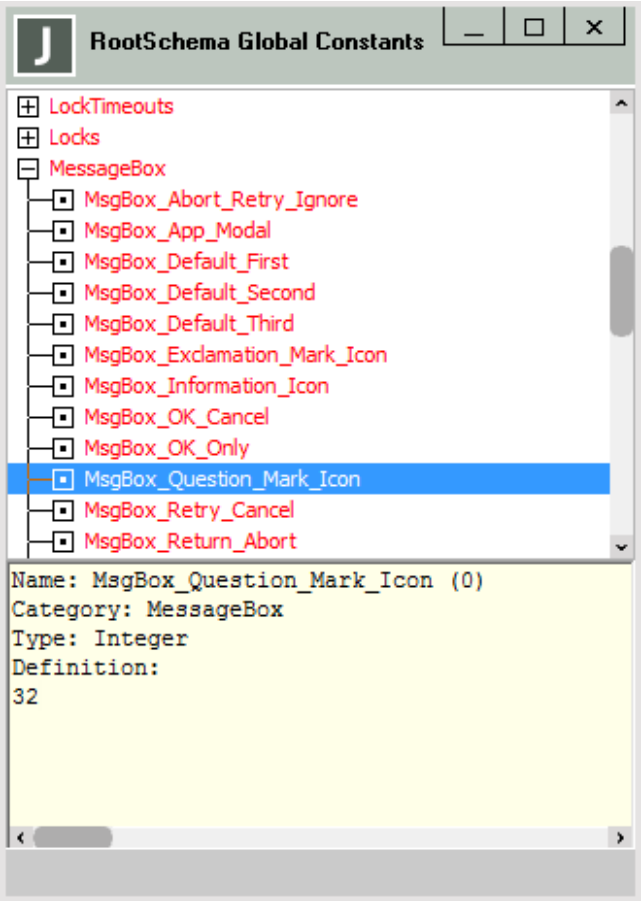
In this **appMethods** method:

- Single-line comments begin with two forward slash characters (*//*).
- Multiple-line comments are enclosed between */** and ***/*.

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.

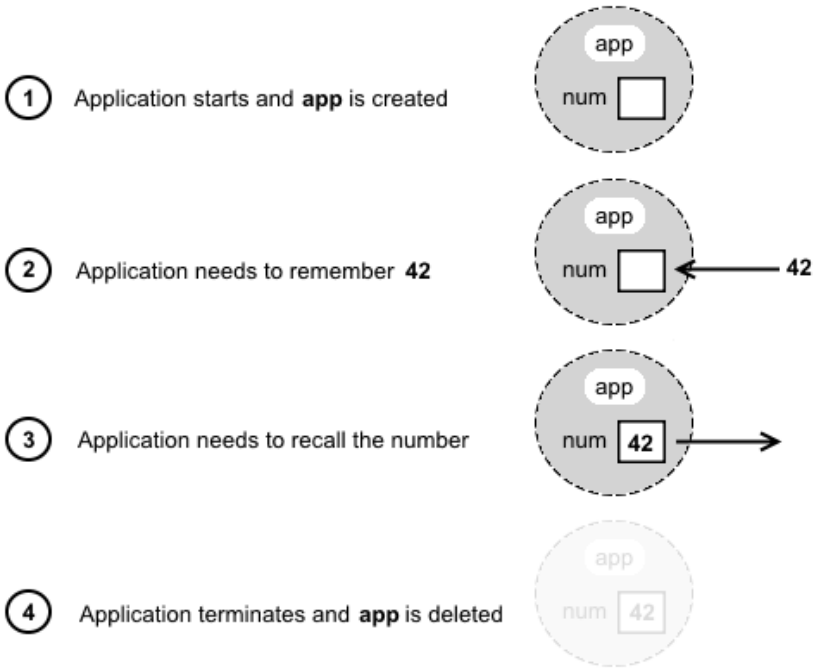
Access the list of categories and the global constants they contain, by using the Browse menu **Global Constants** command. The following diagram shows the global constants and categories in **RootSchema**.



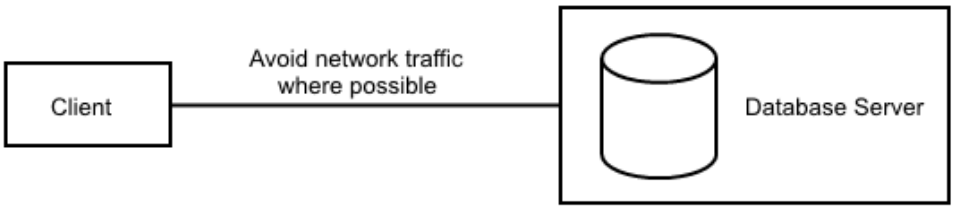
Another Use of the Application Object

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.

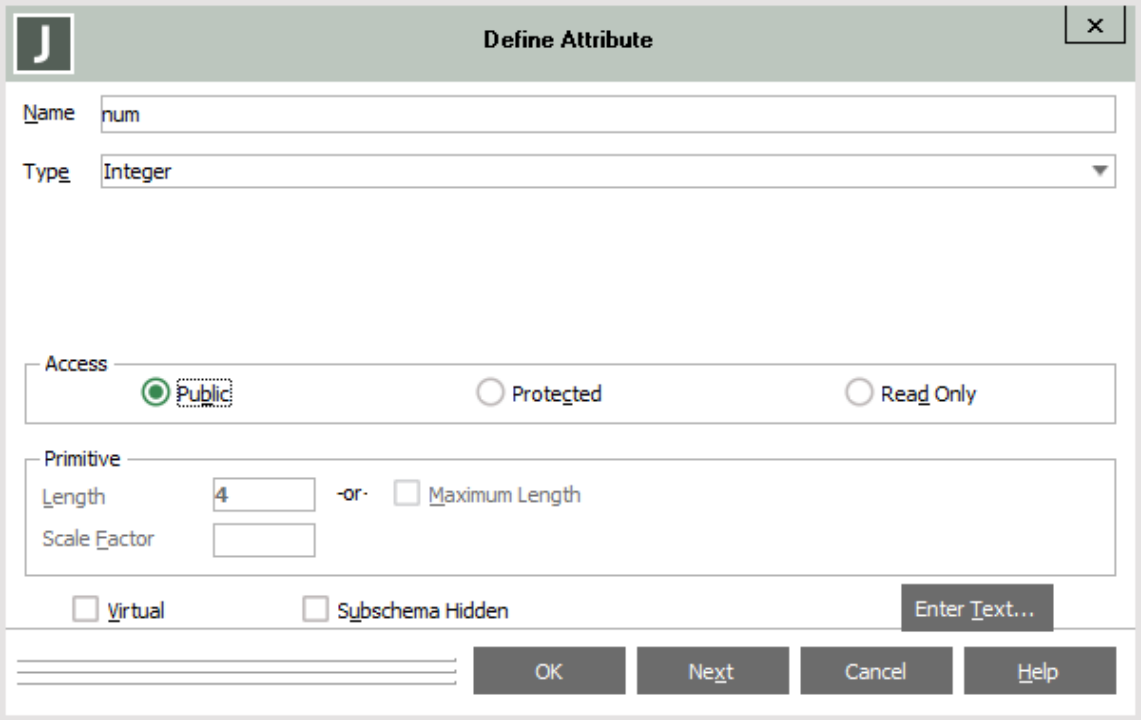


Exercise 4.2 – Adding an Attribute

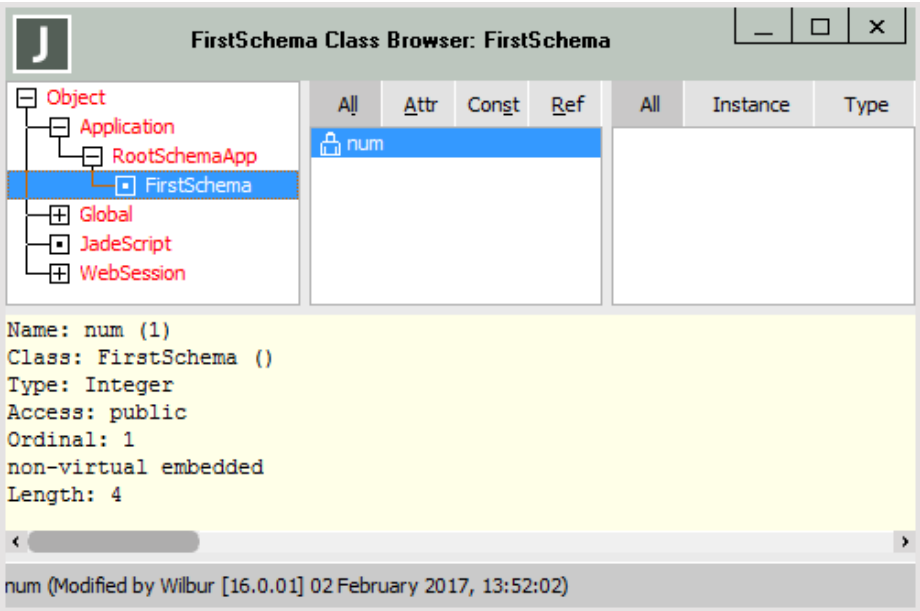
In this exercise, you will add a **num** attribute to your **Application** subclass.

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 and 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;
```

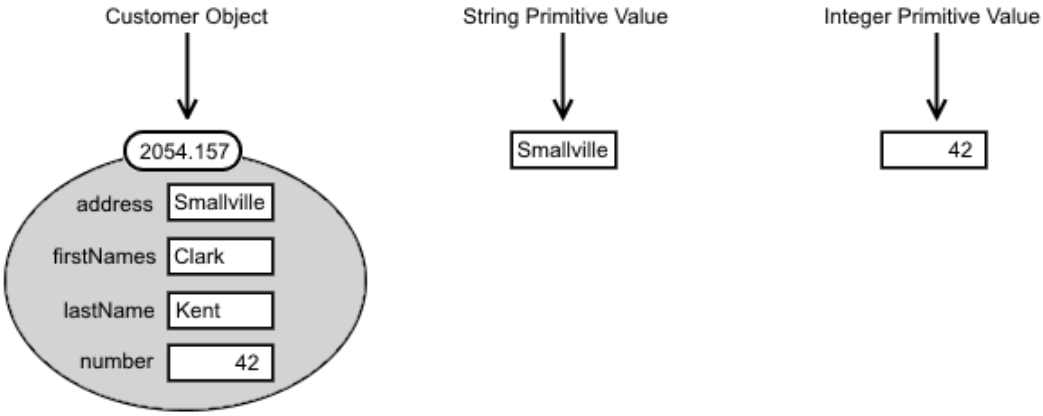
2. Execute the JadeScript method.

Module 5 Primitive Types

This module contains the following topics.

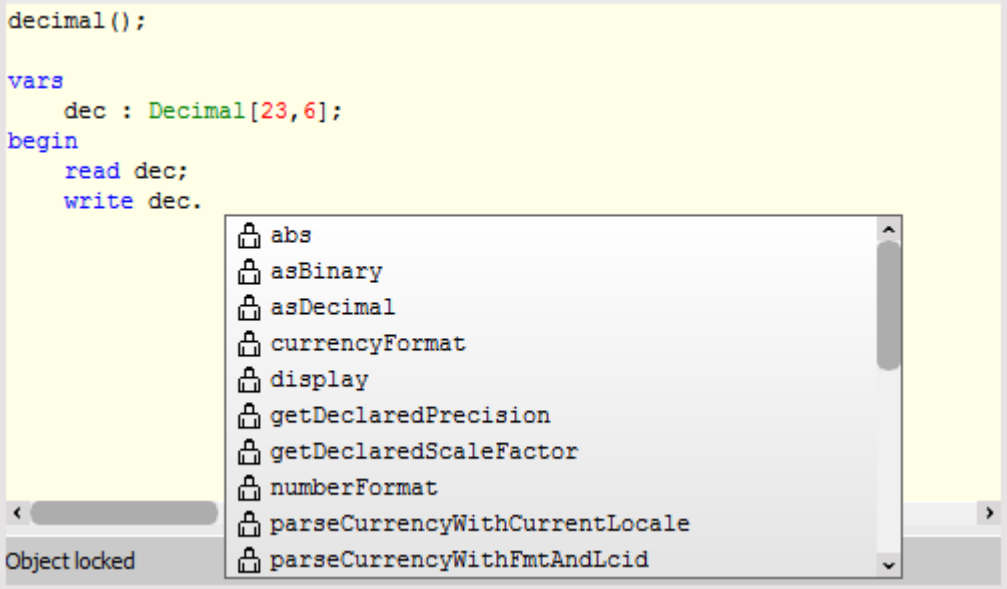
Introduction

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.



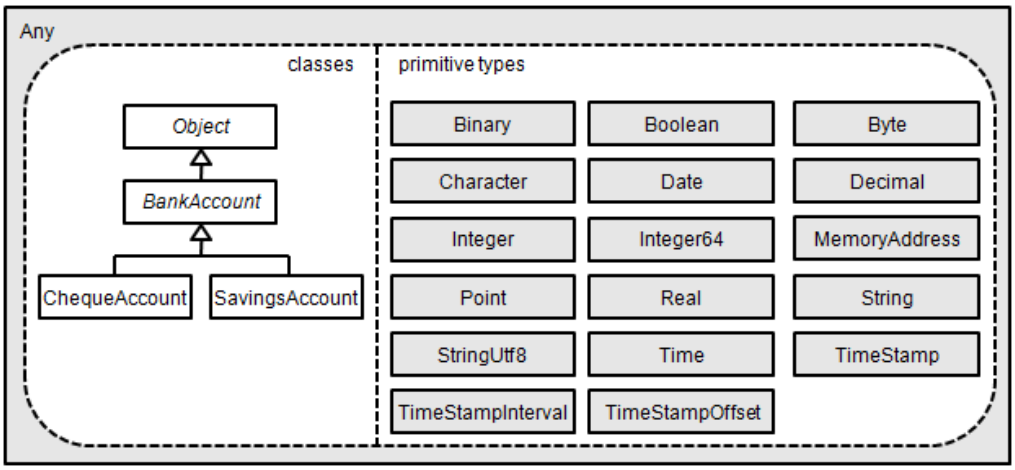
Primitive Types

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 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).

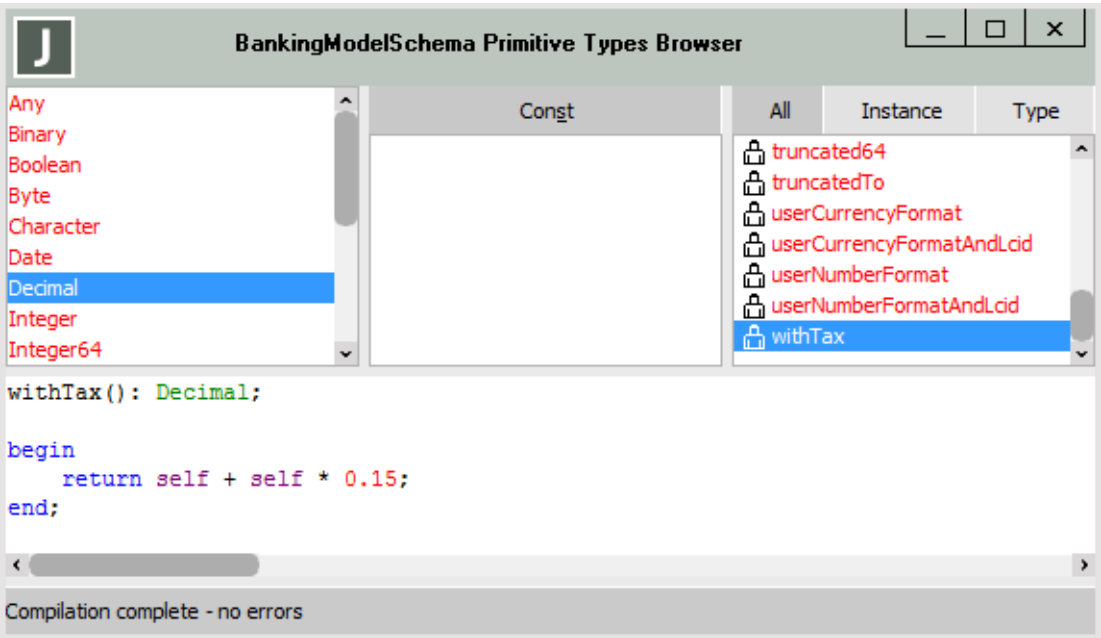
```
vars
    dec: Decimal[6, 2];     // 6 digits altogether
                            // 2 are after the decimal point (so 4 are in front)
                            // Maximum value would be 9999.99
```

- **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 the price or an article, the price with tax included is often required. You could add a **withTax** method, as shown in the following diagram.



To open a Primitive Types Browser, click the **P** button from the JADE 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.

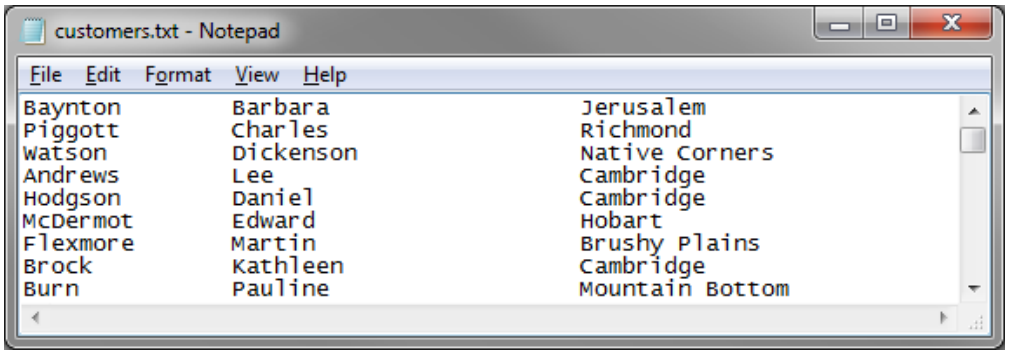
```
testTax();

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 diagram.



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.

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

```
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;
```

4. 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, xmas: Date;
begin
    xmas := "25 December 2015".Date;
    write xmas - today;
end;
```

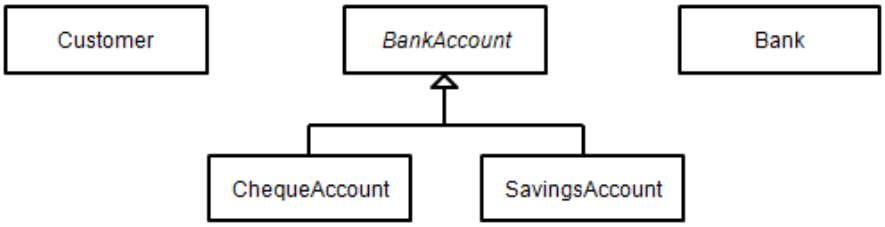
2. Execute the method.

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.



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

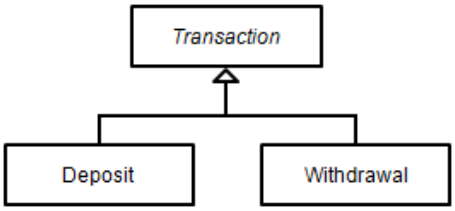
The **BankAccount** class is the abstract superclass for the hierarchy of bank account classes.

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

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.

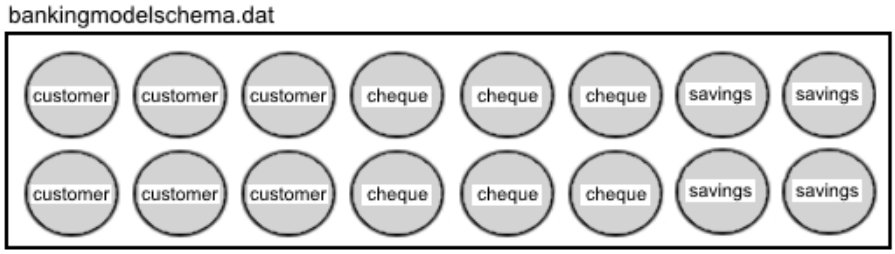
The **Bank** class is the *root object* class for the system. (The purpose of a root object will be explained in a later module.)

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

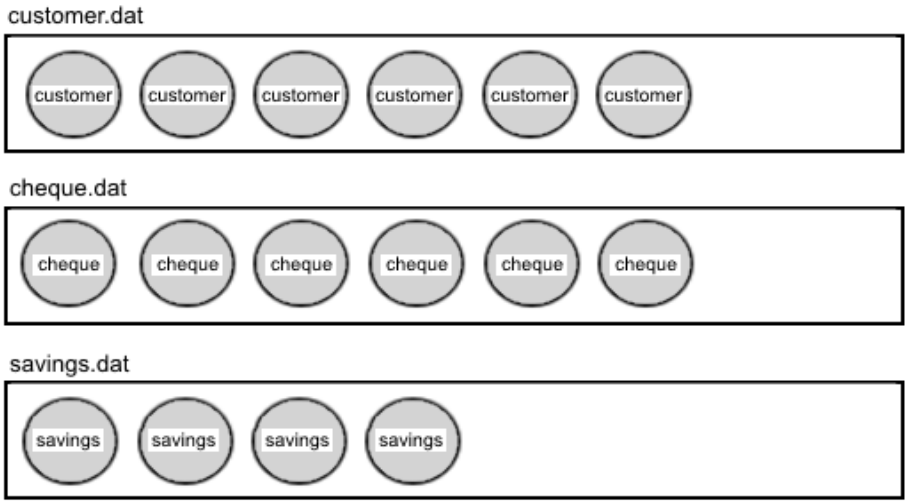


Database Files

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.

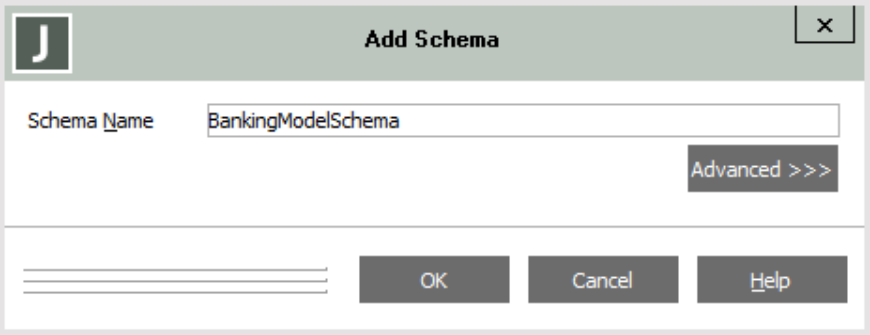


When classes are mapped to separate map files, the impact of a database reorganization can be limited, resulting in saving time because only the affected files need to be reorganized.

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 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.



Exercise 6.2 – 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 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.dat** and **savings.dat** map files.

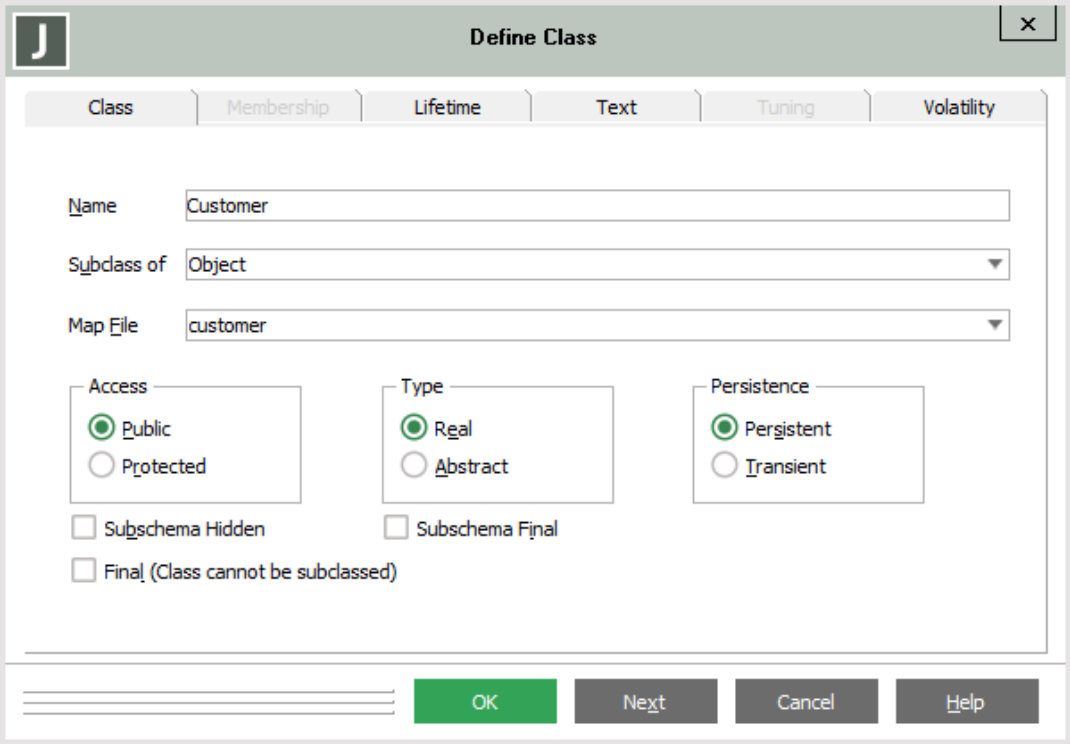
Exercise 6.3 – Adding a Class

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

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

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

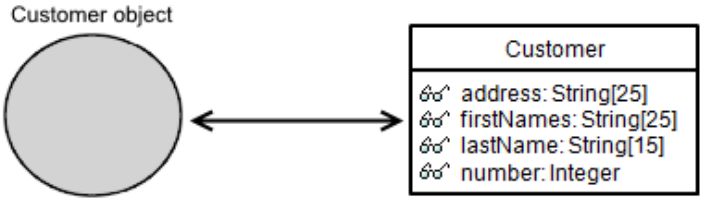
   a. Enter **Customer** as the name of the class and then click the **OK** button.



Instances of a Class

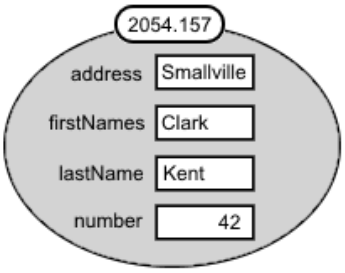
The main component of any JADE application is an object. These objects represent real-world entities. When building a JADE application, you merely mirror reality by creating the components that make up the real-world business system.

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.

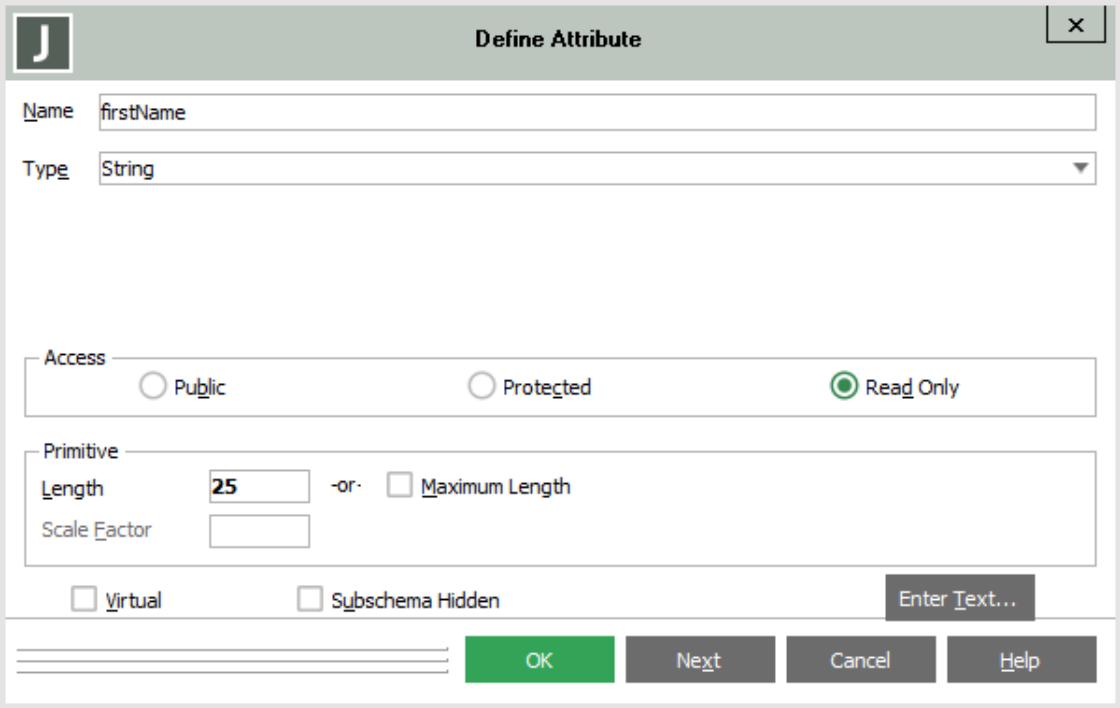
In this course, the read-only option (a pragmatic compromise between public and protected) is used for most properties.

Exercise 6.4 – Adding Attributes

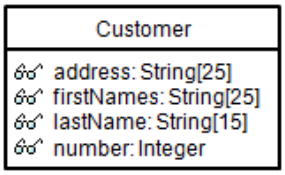
In this exercise, you will add attributes to the **Customer** class.

1. Select the **Customer** class in the Class Browser.
2. Add an attribute by selecting the Properties menu **Add Attribute** command.
3. Enter **firstNames** as the name of the class.

   Select **String** as the type and then set the length to **25** characters. Set the access mode to read-only, and then click the **OK** button.



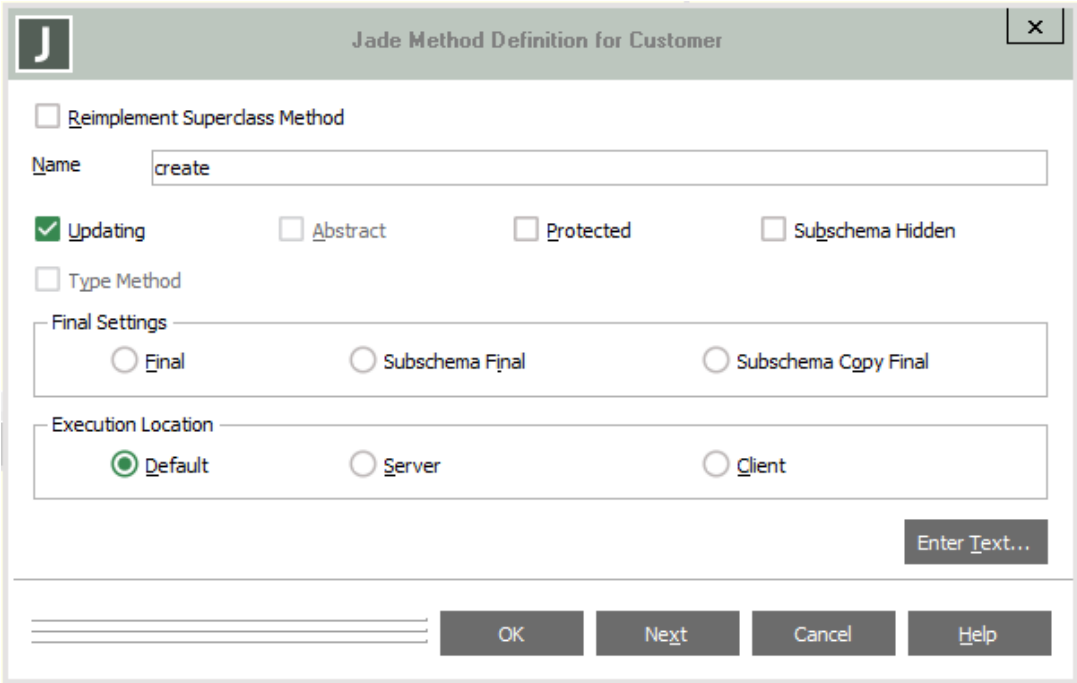
4. Add the read-only attributes specified in the following UML class diagram, making sure that you set the lengths to the values specified in the following diagram.



Exercise 6.5 – Adding a Method

In this exercise, you will add a **create** method with parameters (also known as a *constructor with parameters*) to the **Customer** class.

1. Select the **Customer** class in the Class Browser.
2. Add a method by selecting the Methods menu **New Jade Method** command.
3. Enter **create** as the name of the method. The **Updating** check box will be checked automatically.
4. Click the **OK** button.



5. Code the method as follows.

```
create(addr, first, last: String) updating;

begin
    self.address := addr.trimBlanks();
    self.firstNames := first.trimBlanks();
    self.lastName := last.trimBlanks();
end;
```

6. Compile the method by pressing F8.

In this method:

- The **number** property is not being set. (In a later module, you will code a mechanism to generate a unique value.)
- The **updating** method option is automatically included in the method signature, because the **create** method is called whenever the class is instantiated.
- The variable **self** refers to the receiver; that is, the object for which the method is executing, which is a **Customer** object.
- The **trimBlanks** method removes any trailing or leading spaces in the data supplied for the new customer.

Exercise 6.6 – 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. Code the method as follows.

```
createCustomer();

vars
    cust: Customer;
begin
    app.initialize();
    beginTransaction;
    cust := create Customer("Gotham City", "Bruce", "Wayne") persistent;
    commitTransaction;
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.

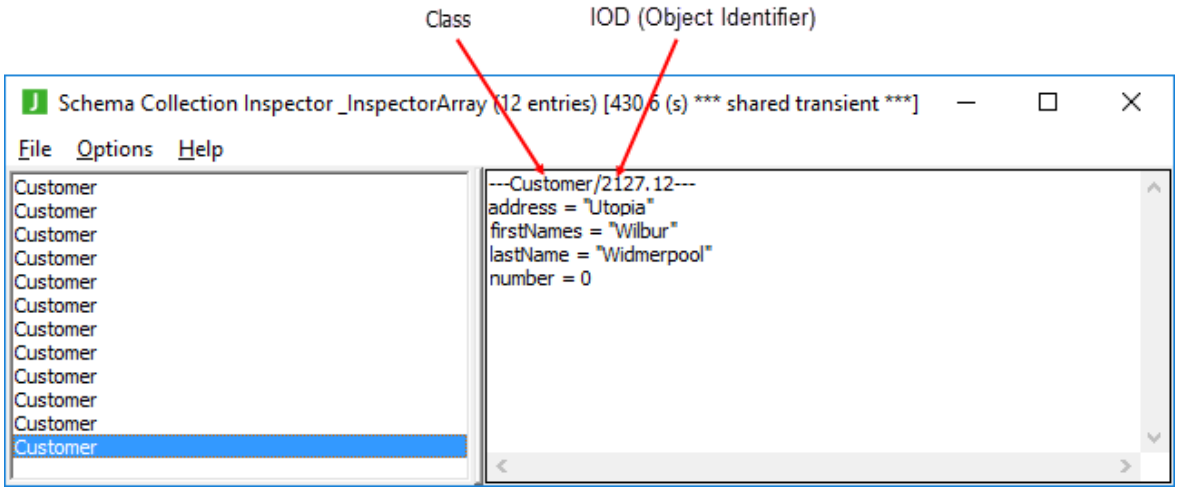
   There should be two customers in the database.

In this method:

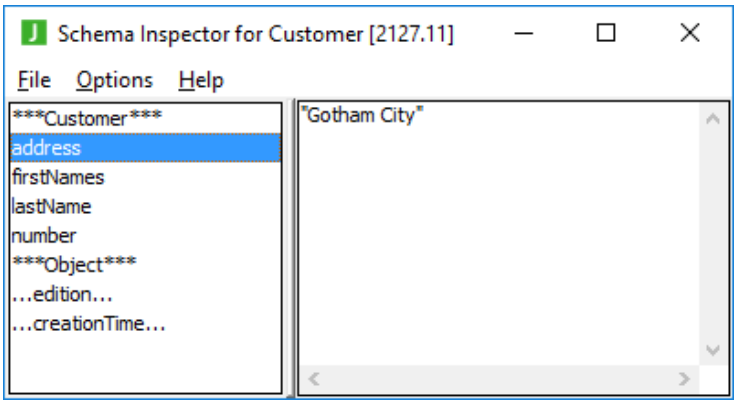
- The **create** instruction is used to create an object and to initialize the newly created object.
- The instructions creating the customer (that is, an address, a first name, and a last name) are contained within the **beginTransaction** and **commitTransaction** instructions.

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.

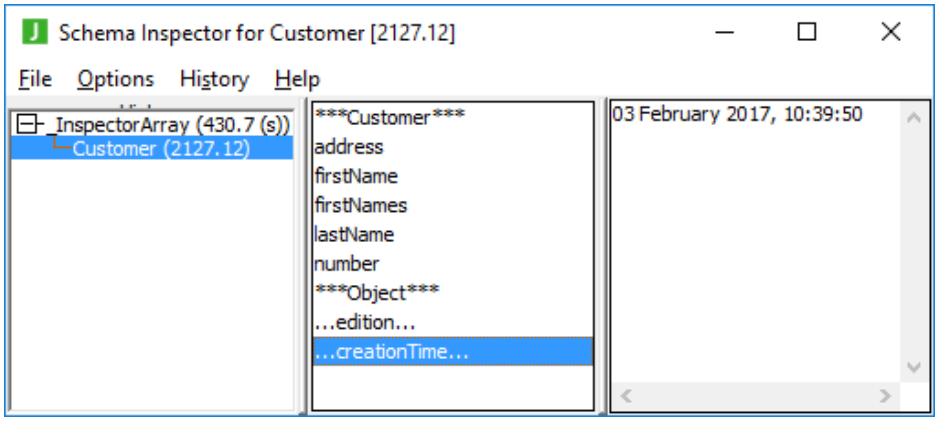


If you double-click an object in the left-hand window, a new Object Inspector window is opened to display the object in detail.



If you single-click a property in the left-hand window, the value of the property is displayed in the right-hand window. 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 diagram.



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.

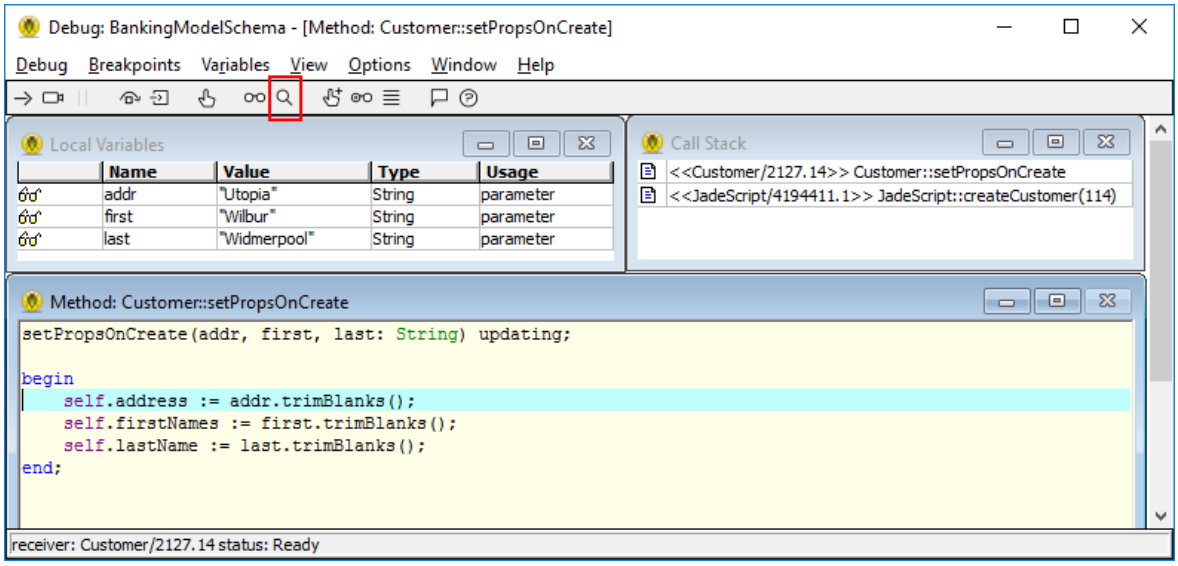
The ways in which you can invoke the Object Inspector are as follows.

- 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.

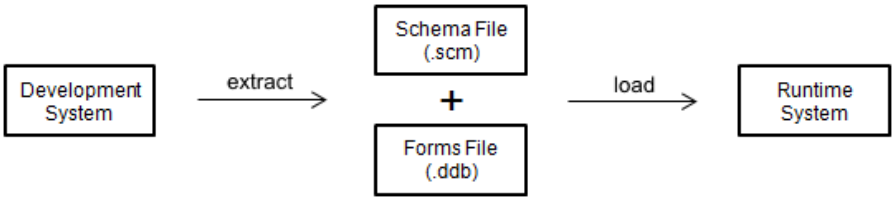
```
cust.inspect();
cust.inspectModal();
```

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



Extracting and Loading Schemas

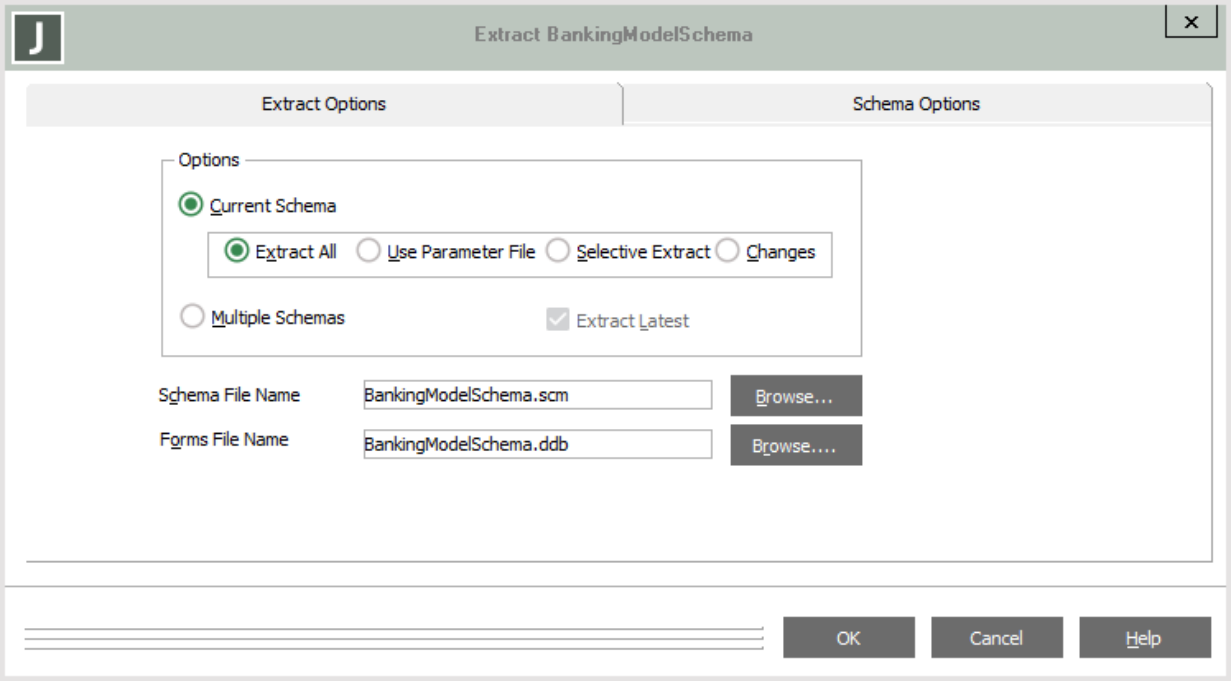
You can extract a complete schema, parts of a schema, or multiple schemas; for example, as a backup before you reorganize your database or you install a new release of JADE. You can load the extract files into another JADE system. The deployment mechanism for a JADE system is shown in the following diagram.



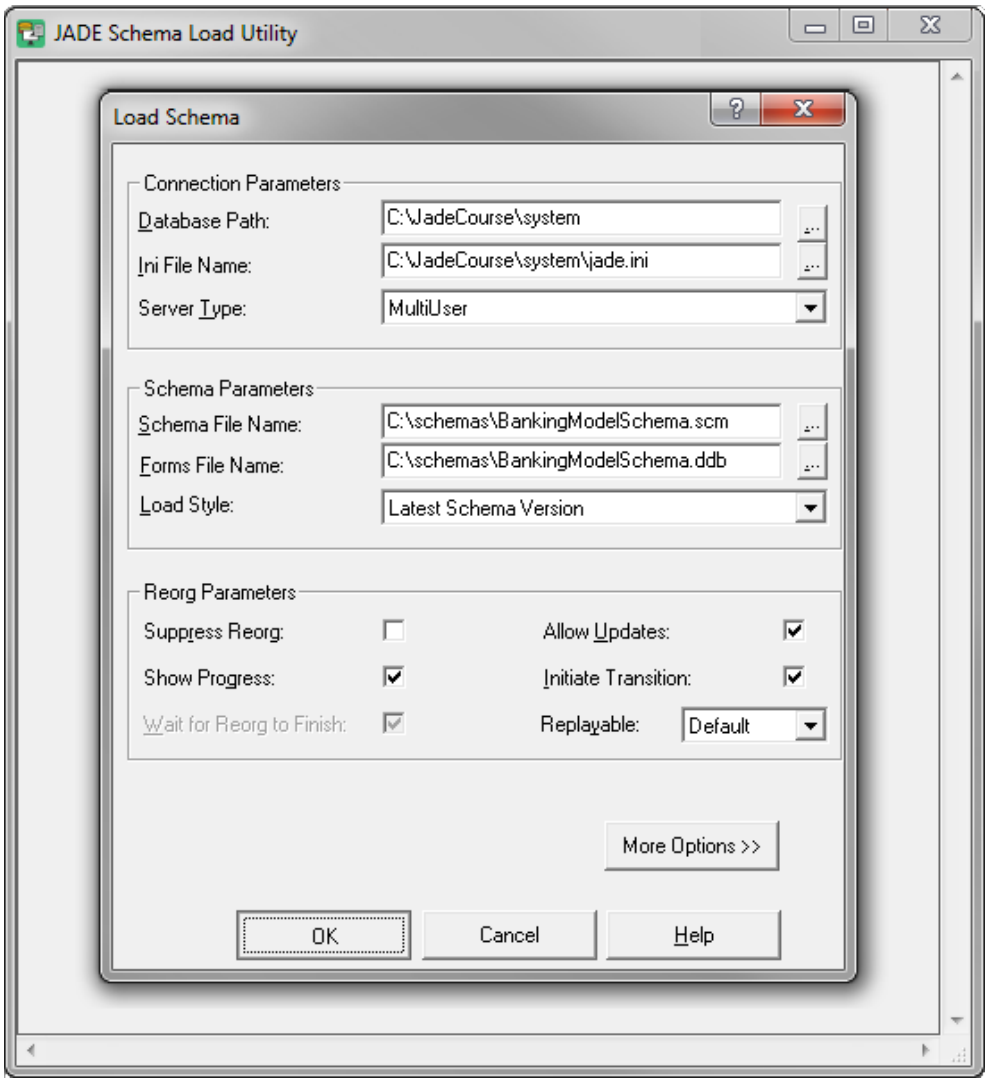
The extract process creates two files.

- 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.



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



Exercise 6.7 – 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 6.8 – 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;
```

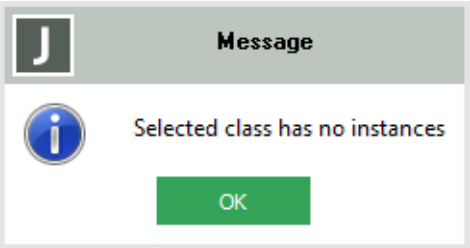
   In this method:

   - 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.

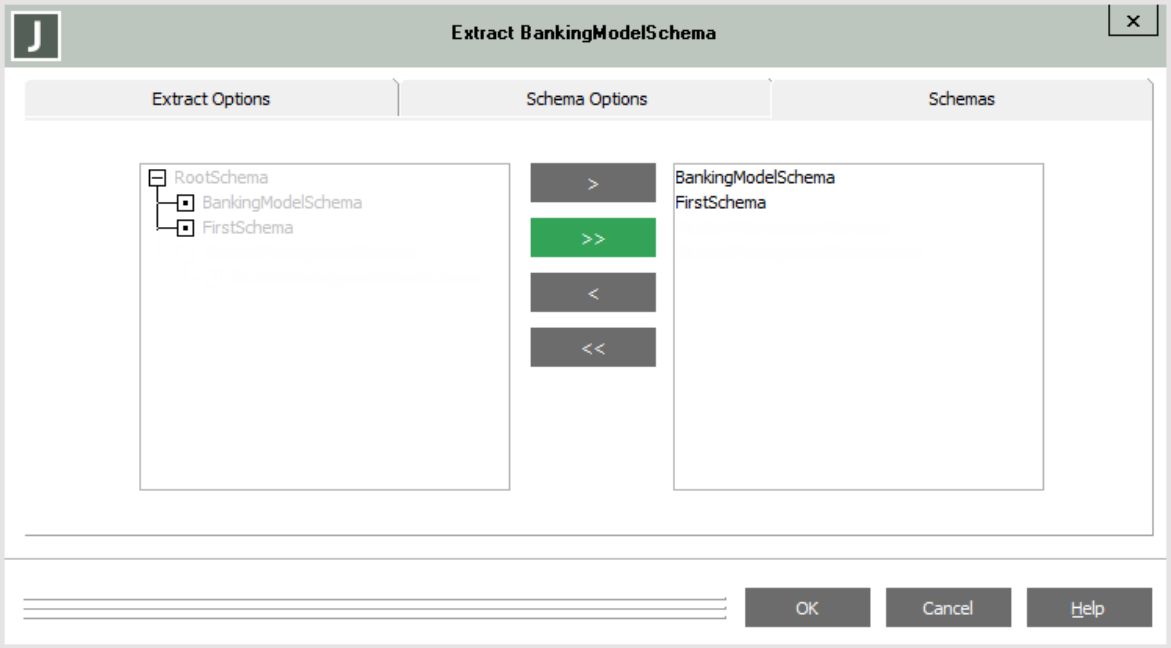


Exercise 6.9 – Extracting Multiple Schemas

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 **Schema File Name** text box to **Banking.mul** and then click the **Browse** button to specify where the extract files should be located.

5. Select the **Schemas** tab and then click the arrow key to select both schemas.



6. Click the **OK** button.

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

```
#MULTIPLE_SCHEMA_EXTRACT
BankingModelSchema.scm BankingModelSchema.ddb
FirstSchema.scm FirstSchema.ddb
```

Module 7 Root Object

This module contains the following topics.

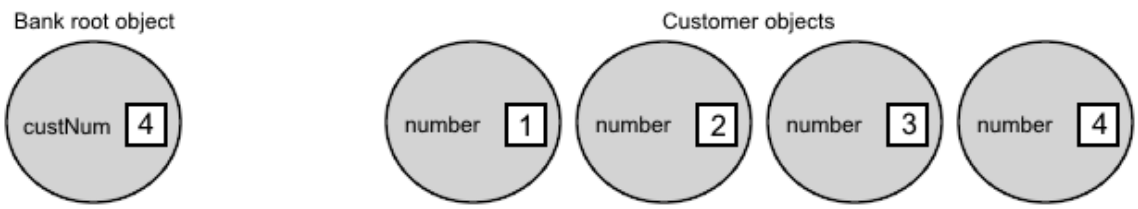
Introduction

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.

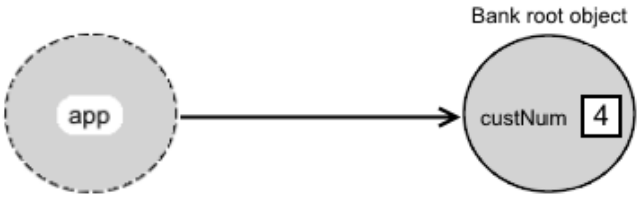


Initializing the Root Object

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.

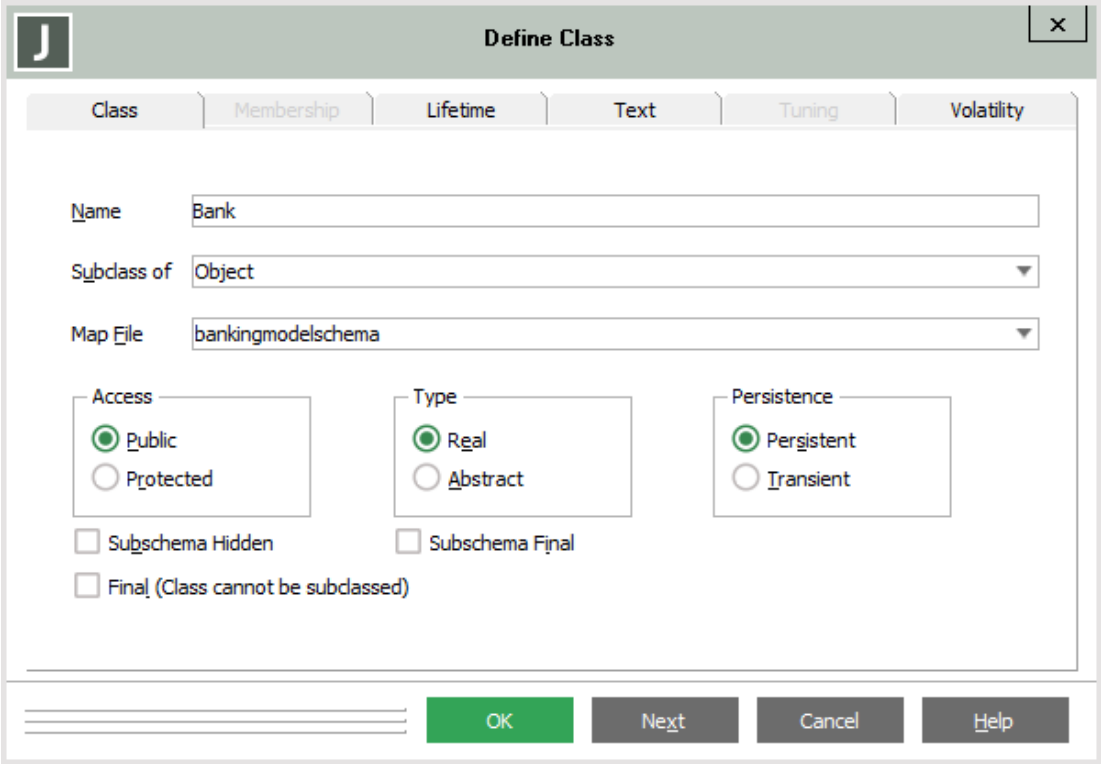
When a **Customer** object is created, you will use a constructor to set the value of the **number** attribute to the value returned by the **nextCustNum** method of the root object.

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.

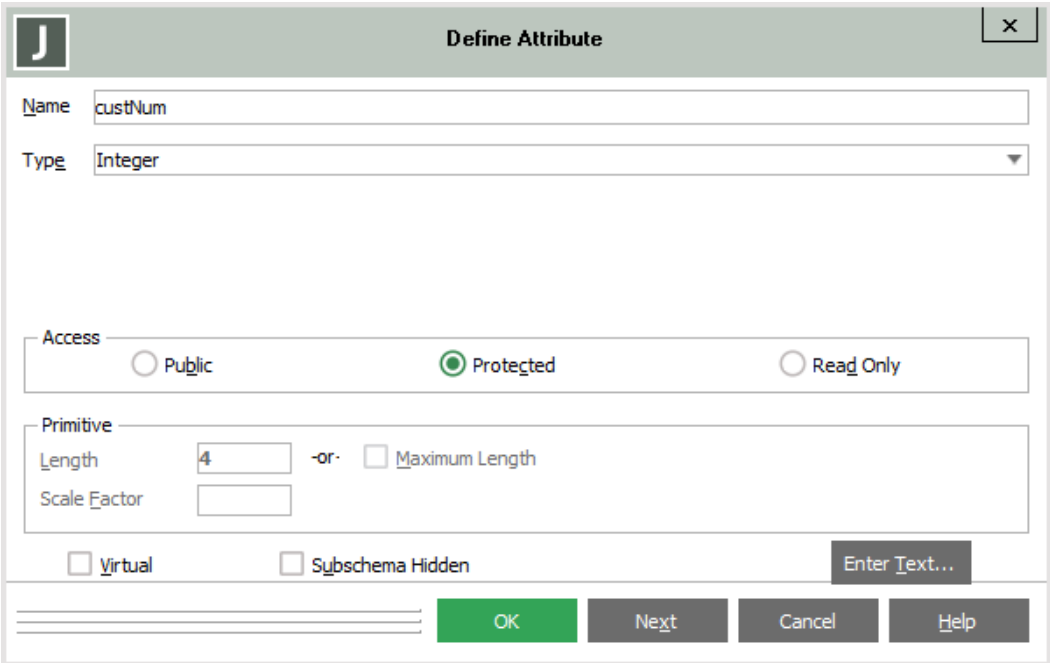
1. Select the **Object** class in the Class Browser.
2. Add a class by selecting the Classes menu **Add** command.

3. Enter **Bank** as the name of the class and then click the **OK** button.



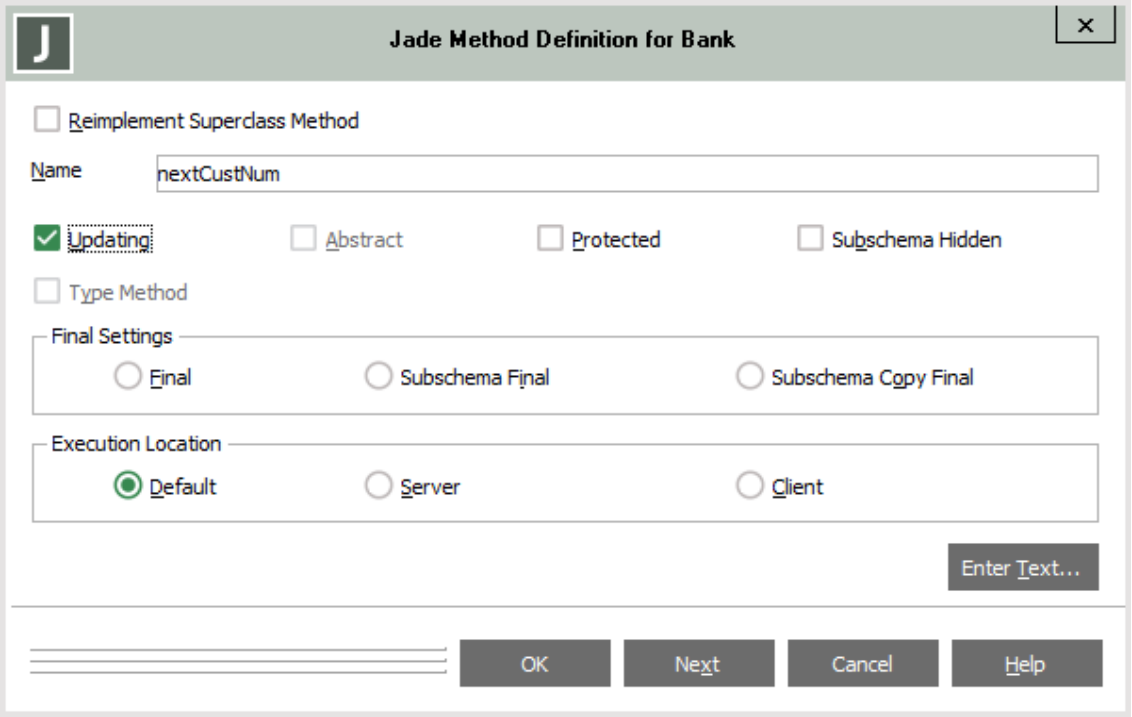
4. Add an attribute called **custNum**, by selecting the Properties menu **Add Attribute** command.

   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 **nextNum** attribute.



6. Code the method as follows.

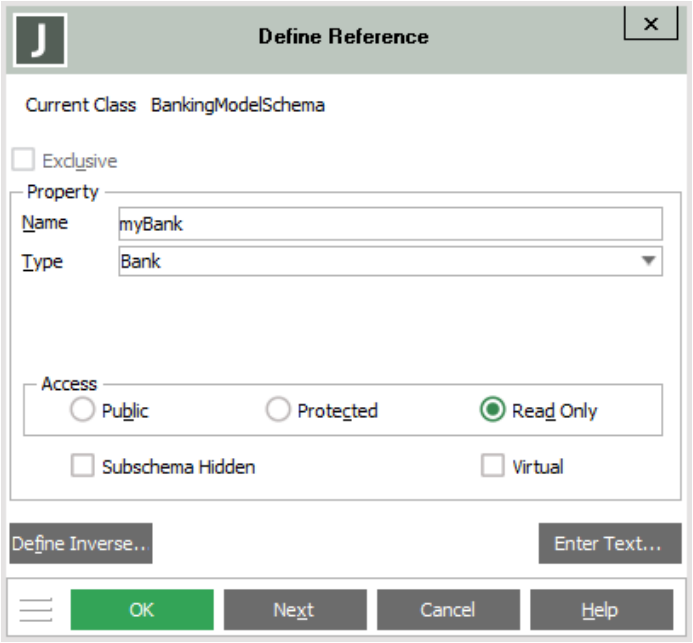
```
nextCustNum(): Integer updating;

begin
    self.custNum := self.custNum + 1;
    return self.custNum;
end;
```

Exercise 7.2 – Adding *myBank* and *initialize* Method

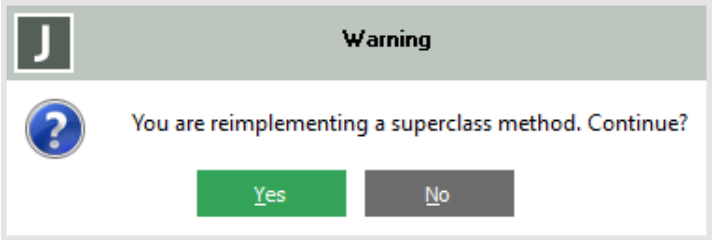
In this exercise, you will add a reference to the root object in your **Application** subclass.

1. Select your **Application** subclass in the Class Browser.
2. Add a reference by selecting the Properties menu **Add Reference** command.

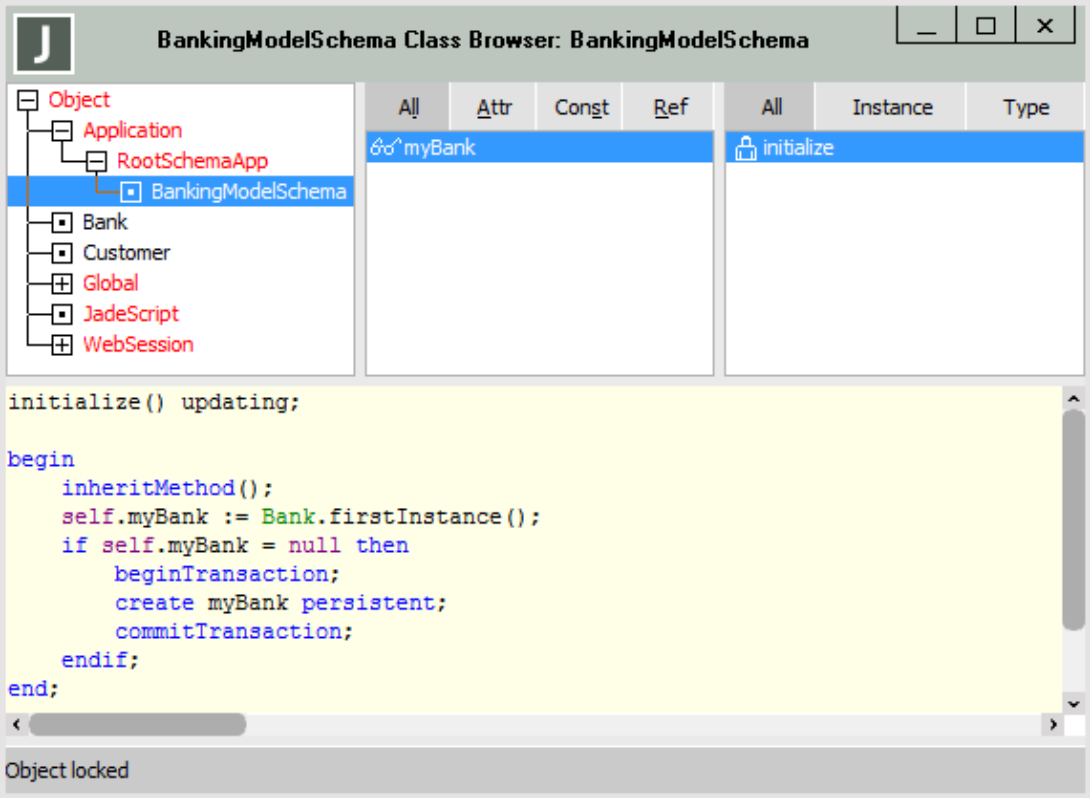
3. Enter **myBank** as the name, select **Bank** as the type, set the access mode to read-only, and then click the **OK** button.



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 diagram.



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

Exercise 7.3 – Adding a *Customer* Constructor

In this exercise, you will modify the constructor of 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 the following to the **create** method.

```
create(addr, first, last: String) updating;

begin
    self.number := app.myBank.nextCustNum();
    self.address := addr.trimBlanks();
    self.firstNames := first.trimBlanks();
    self.lastName := last.trimBlanks();
end;
```

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

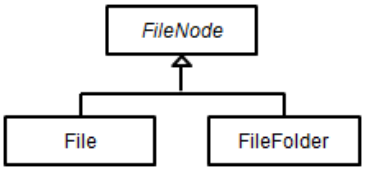
```
createCustomer();

vars
    cust: Customer;
begin
    beginTransaction;
    app.initialize();
    cust := create Customer("Gotham City", "Bruce", "Wayne") persistent;
    commitTransaction;
end;
```

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

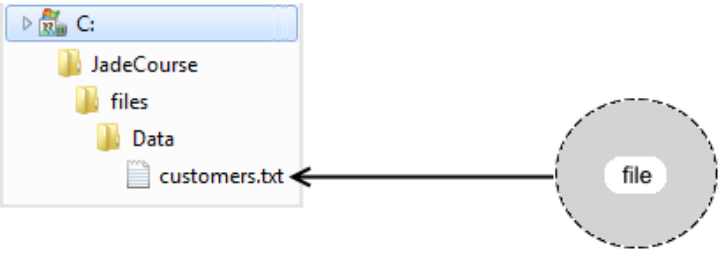
5. Inspect the two new customers. The value of the **number** attribute should be **1** for the first customer and **2** for the second customer.

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 on disk, you create a transient instance of the **File** class and set its **codename** property to the full path name of the file.



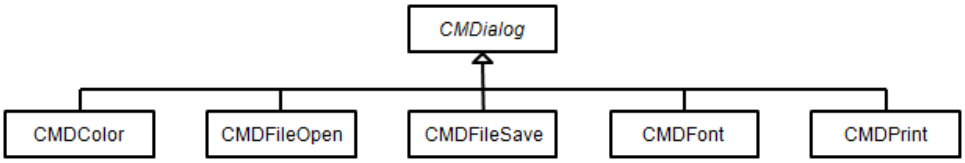
The following methods of the **File** class are used to read the information in a file.

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
    file: File;
    str:  String;
    cust: Customer;
begin
    app.initialize();
    create file transient;
    file.fileName := "C:\JadeCourse\Files\Customers.txt";
    while not file.endOfFile() do
        str := file.readLine();
        beginTransaction;
        cust := create Customer(str[41:end], str[16:25], str[1:15]);
        commitTransaction;
    endwhile;
epilog
    delete file;
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 file 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 file.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
    dlg: CMDFileOpen;
    file: File;
    str: String;
    cust: Customer;
begin
    app.initialize();
    create dlg transient;
    if not dlg.open() = 0 then
        // Exit as user did not select a file
        return;
    endif;
    create file transient;
//  file.fileName := "C:\JadeCourse\Files\Customers.txt";
    file.fileName := dlg.fileName;
    while not file.endOfFile() do
        str := file.readLine();
        beginTransaction;
        cust := create Customer(str[41:end], str[16:25], str[1:15]);
        commitTransaction;
    endwhile;
epilog
    delete dlg;
    delete file;
end;
```

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

In this method:

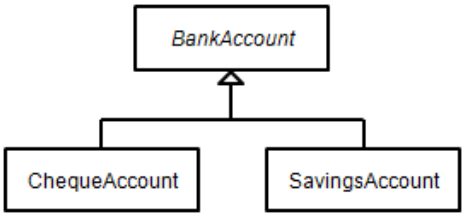
- **app.initialize** is executed as the first instruction, so that the method can access the root object.
- A transient **CMDFileOpen** object is created and it is deleted in the **epilog** section.
- The method is exited from early if the user fails to open a file successfully.

Module 8 Inheritance and Polymorphism

This module contains the following topics.

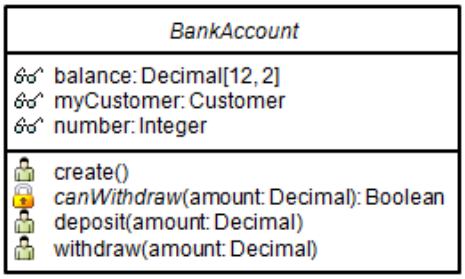
Introduction

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;
```

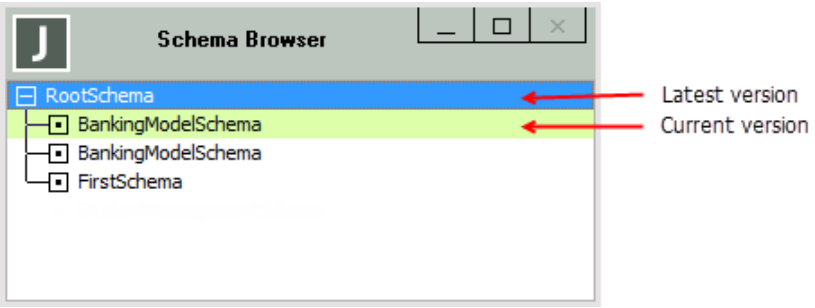
An abstract method specifies the parameters and return type that the implementation of the method inherits.

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

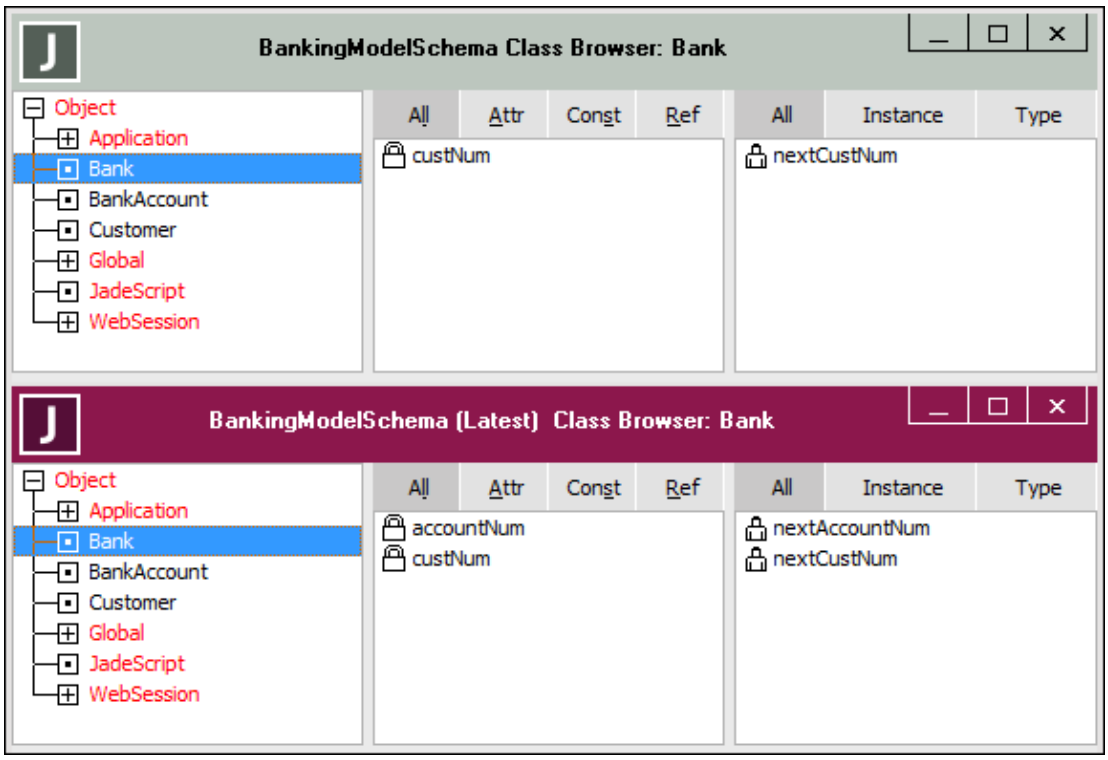
From the schema browser, you can create another version of your schema.



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 diagram 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.

Caution If the reorganization fails, you may need to restore a backup copy of the database.

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.

You can use the Schema menu **Unversion** command to discard the latest version.

The advantages of making changes in the latest schema are:

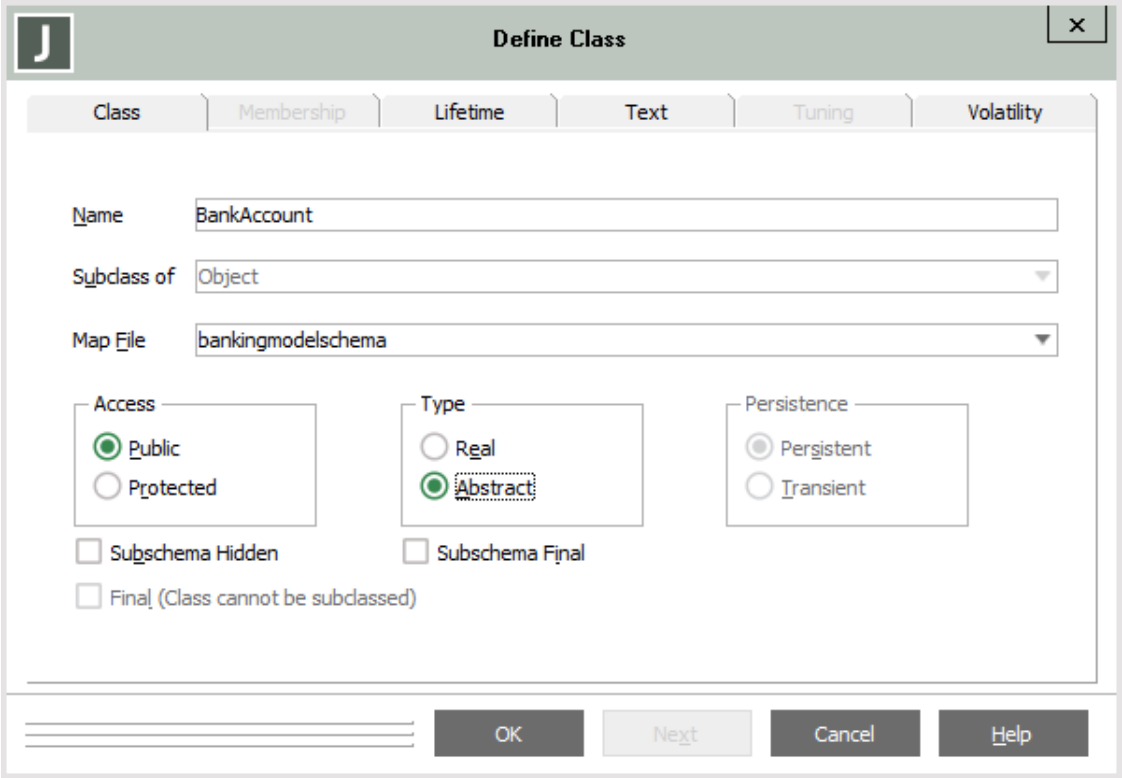
- Implementation of changes can be deferred until the most-convenient time.
- The current version is available while the latest version is reorganized. Only the final transition step requires the system to be offline.

Exercise 8.1 – Adding an Abstract Class

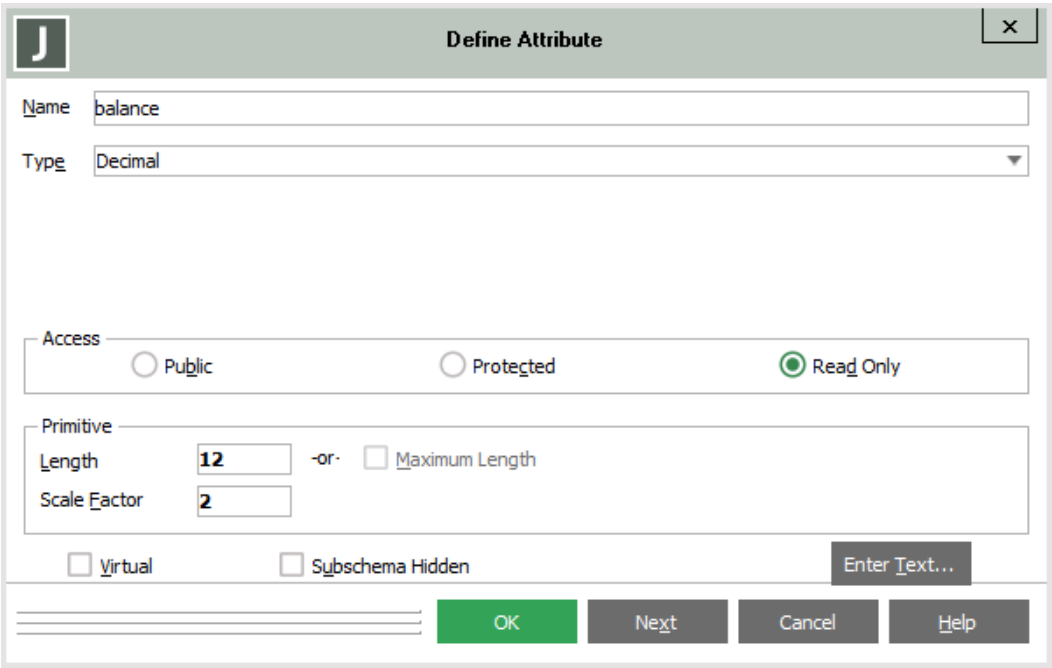
In this exercise, you will add an abstract **BankAccount** class in the **BankingModelSchema**. The properties and methods will be those specified in the UML class diagram under "Introduction", earlier in this module.

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

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



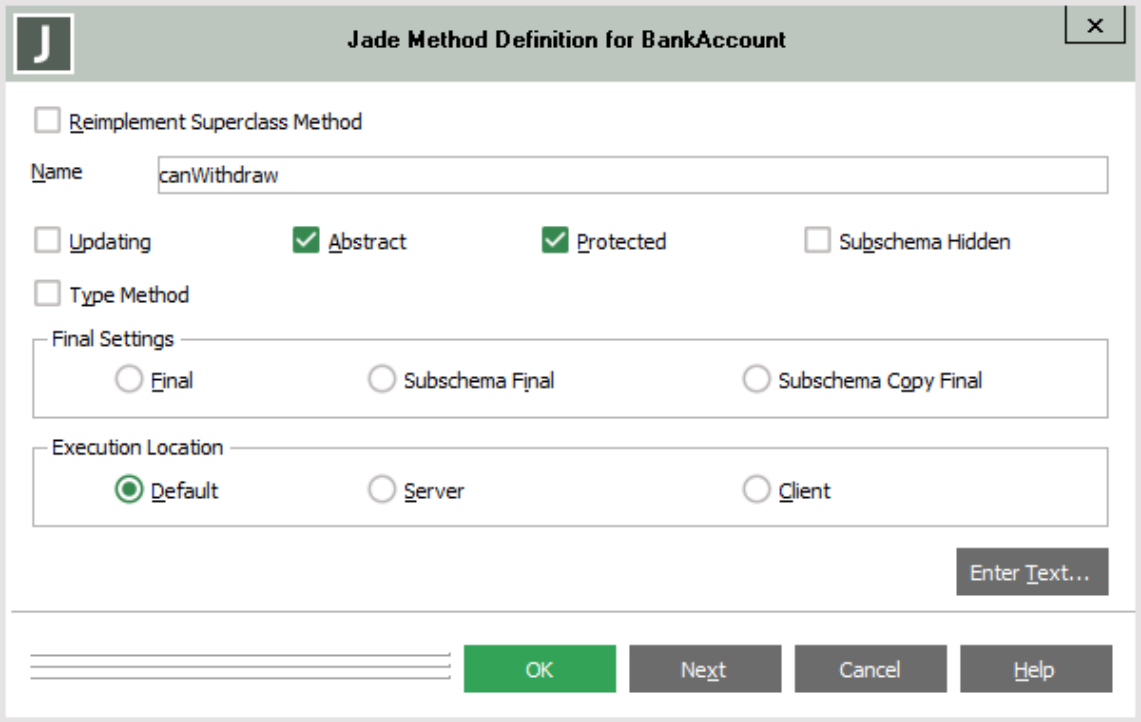
4. Add a read-only **balance** attribute of type **Decimal** with a length (precision) of **12** and a scale factor (number of decimal places) of **2**.



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

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

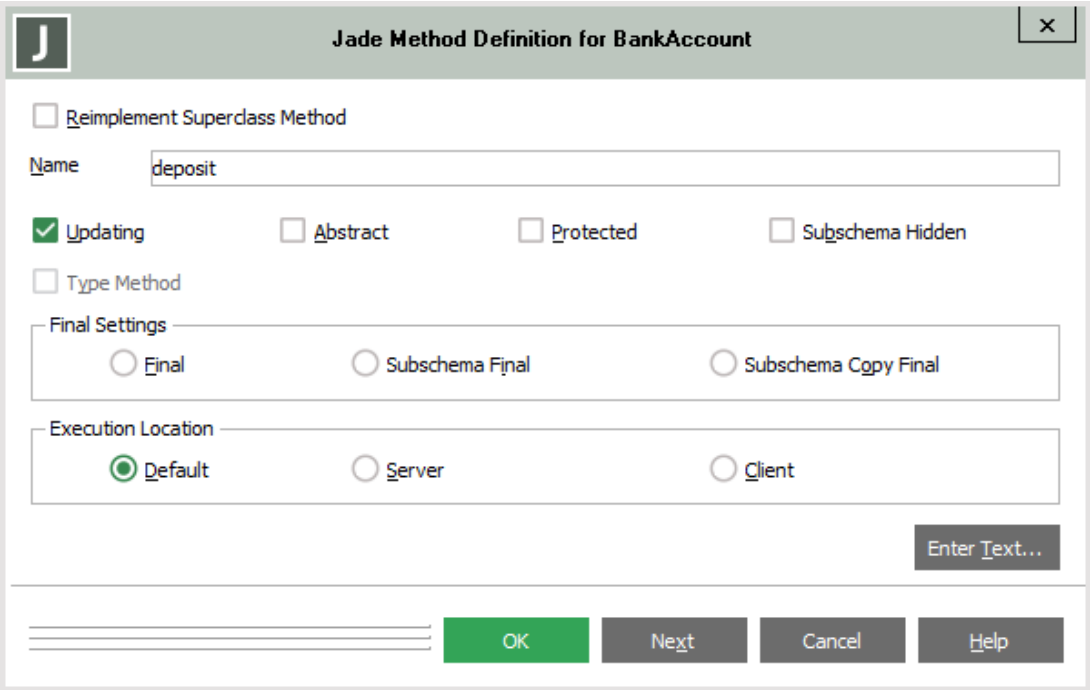
7. Add a **canWithdraw** method that is **abstract** and **protected**.



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

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

9. Add a **deposit** method. Make the method **updating**, because it will change the **balance** attribute.



10. Code the method as follows.

```
deposit(amount: Decimal) updating;

begin
    self.balance := self.balance + amount;
end;
```

11. Add a **withdraw** method. Make the method **updating**, because it will change the **balance** attribute.

12. Code the method as follows.

```
withdraw(amount: Decimal) updating;

begin
    if self.canWithdraw(amount) = true then
        self.balance := self.balance - amount;
    endif;
end;
```

Exercise 8.2 – 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.

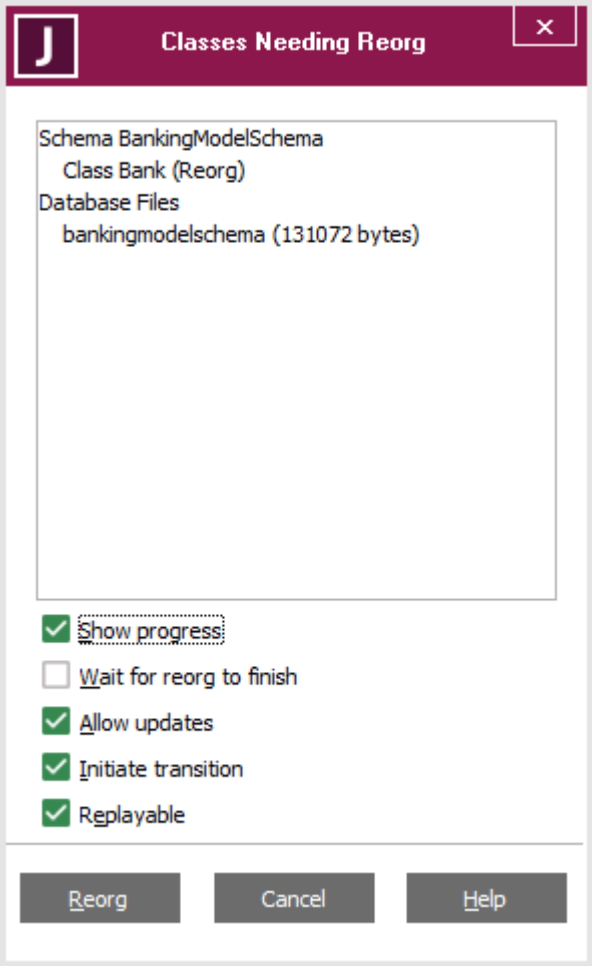
2. Add an attribute called **accountNum** by selecting the Properties menu **Add Attribute** command.

   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 toolbar button, and then clicking the **Reorg** button in the Classes Needing Reorg dialog.



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

7. Code the method as follows.

```
nextAccountNum(): Integer updating;

begin
    self.accountNum := self.accountNum + 1;
    return self.accountNum;
end;
```

8. 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 8.3 – Adding a *BankAccount* Constructor

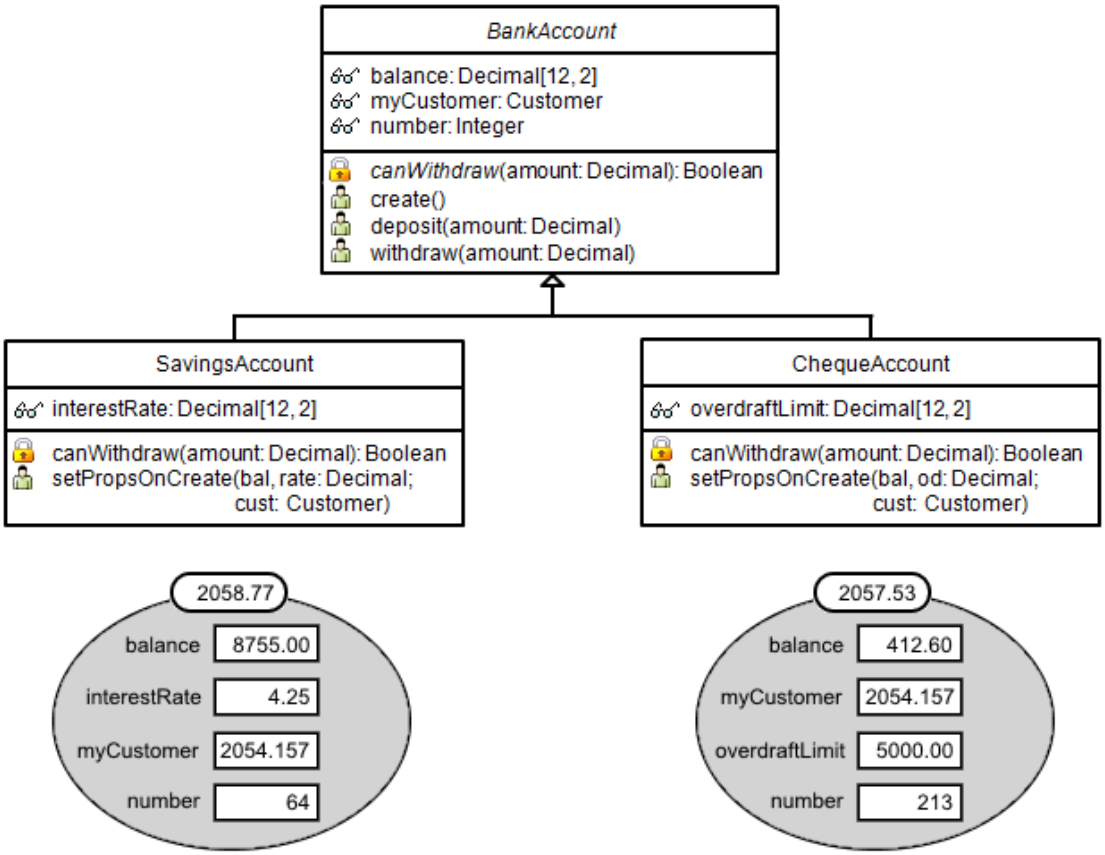
In this exercise, you will add a constructor to the **BankAccount** class, which will assign a new value to the **number** attribute.

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

```
create() updating;
begin
    self.number := app.myBank.nextAccountNum();
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);
```

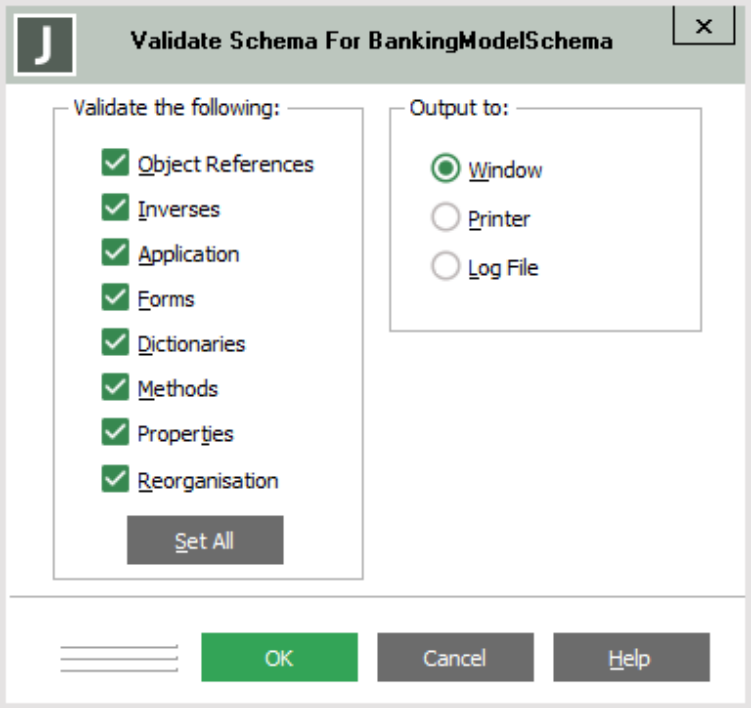
The **ba** variable is of generic type **BankAccount**. At run time, the ATM user selects a cheque account, a savings account, or some other type of bank account and then enters a value for the **amount** parameter.

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.

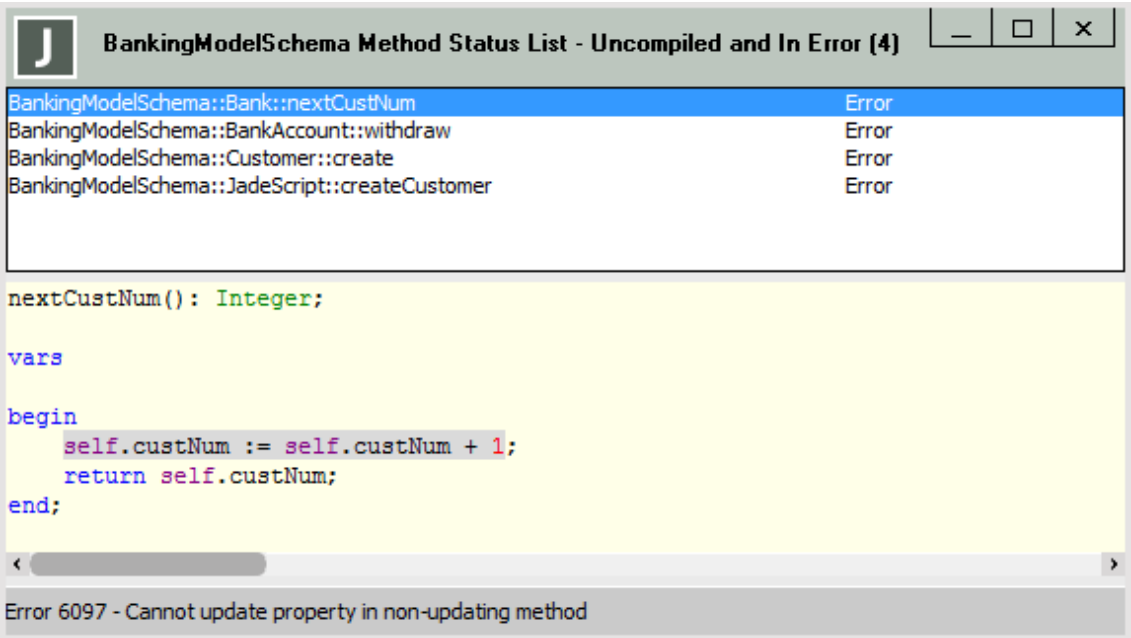
```
// Non-polymorphic coding
if ba.isKindOf(ChequeAccount) then
    // Code for a cheque account
elseif ba.isKindOf(SavingsAccount) then
    // Code for a savings account
endif;
```

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.

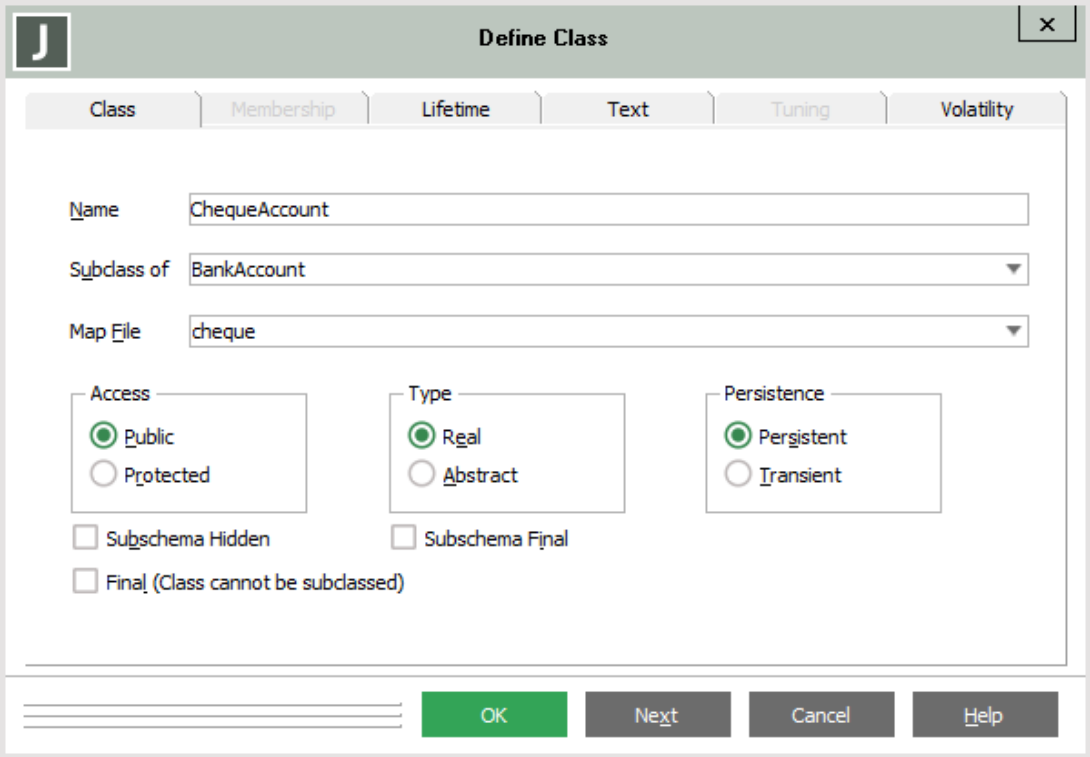


Exercise 8.4 – Adding a ChequeAccount Class

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 a **create** method to initialize the read-only properties.

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 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. Add an updating method called **create**, by selecting the Methods menu **New Jade Method** command.
7. Code the method as follows.

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

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

8. 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.

9. Code the method as follows.

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

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

10. Compile the method.

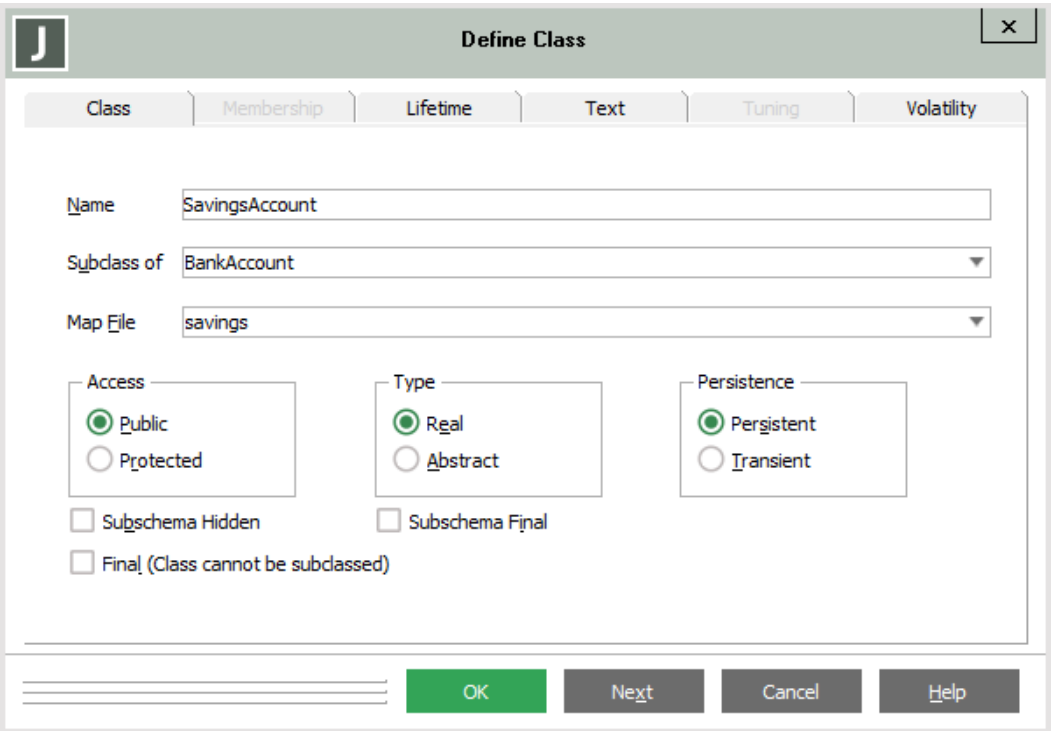
Exercise 8.5 – Adding a *SavingsAccount* Class

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 class by selecting the Classes menu **Add** command.

   a. Enter **SavingsAccount** as the name of the class, select the **savings** map file, and then click the **OK** button.



3. Add a read-only **interestRate** attribute of type **Decimal** with a length (precision) of **12** and a scale factor of **2**.

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

5. Code the method as follows.

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

begin
    self.balance := bal;
    self.interestRate := rate;
    self.myCustomer := cust;
end;
```

6. 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.

7. Code the method as follows.

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

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

8. Compile the method.

Exercise 8.6 – 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
    cheque: ChequeAccount;
    savings: SavingsAccount;
begin
    app.initialize();
    beginTransaction;
    cheque.:= create ChequeAccount(0, 500, null);
    savings := create SavingsAccount(100, 4.5, null);
    commitTransaction;
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 8.7 – ATM Simulation

In this exercise, you will simulate a withdrawal from an ATM.

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

```
simulateATM();

vars
    accountType: String;
    ba: BankAccount;
    amount: Decimal[12,2];
begin
    // Select account
    write 'Enter "cheque" or "savings"';
    read accountType;
    if accountType = "cheque" then
        ba := ChequeAccount.firstInstance();
        write "Balance of cheque account = " & ba.balance.String;
    elseif accountType = "savings" then
        ba := SavingsAccount.firstInstance();
        write "Balance of savings account = " & ba.balance.String;
    endif;
    // Enter amount
    write "Enter amount to withdraw";
    read amount;
    // Process withdrawal
    beginTransaction;
    ba.withdraw(amount);
    commitTransaction;
    write "New balance of account = " & ba.balance.String;
end;
```

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

Module 9 Collections

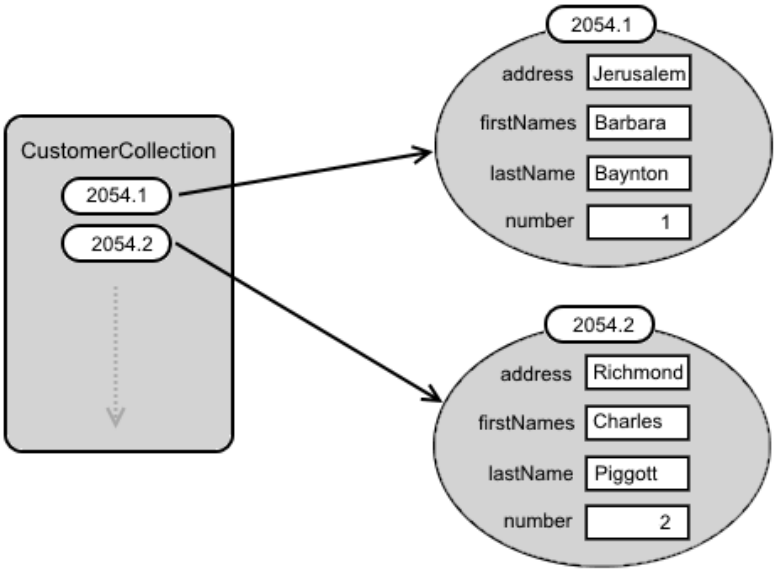
This module contains the following topics.

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:

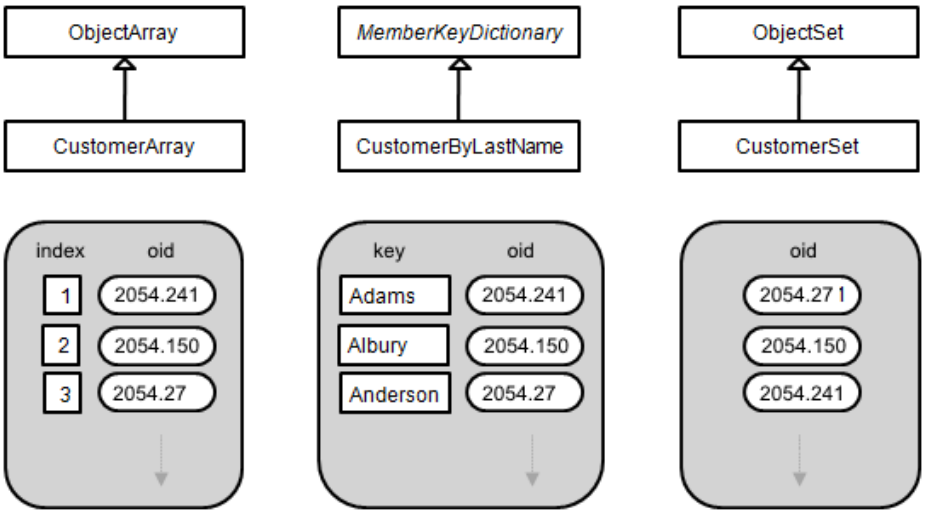
- **Array**, which is a collection of objects or primitive values, ordered by index number. An array can hold the same object or primitive value more than once.
- **Dictionary**, which is a collection of objects ordered by keys that you specify.

  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);`

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

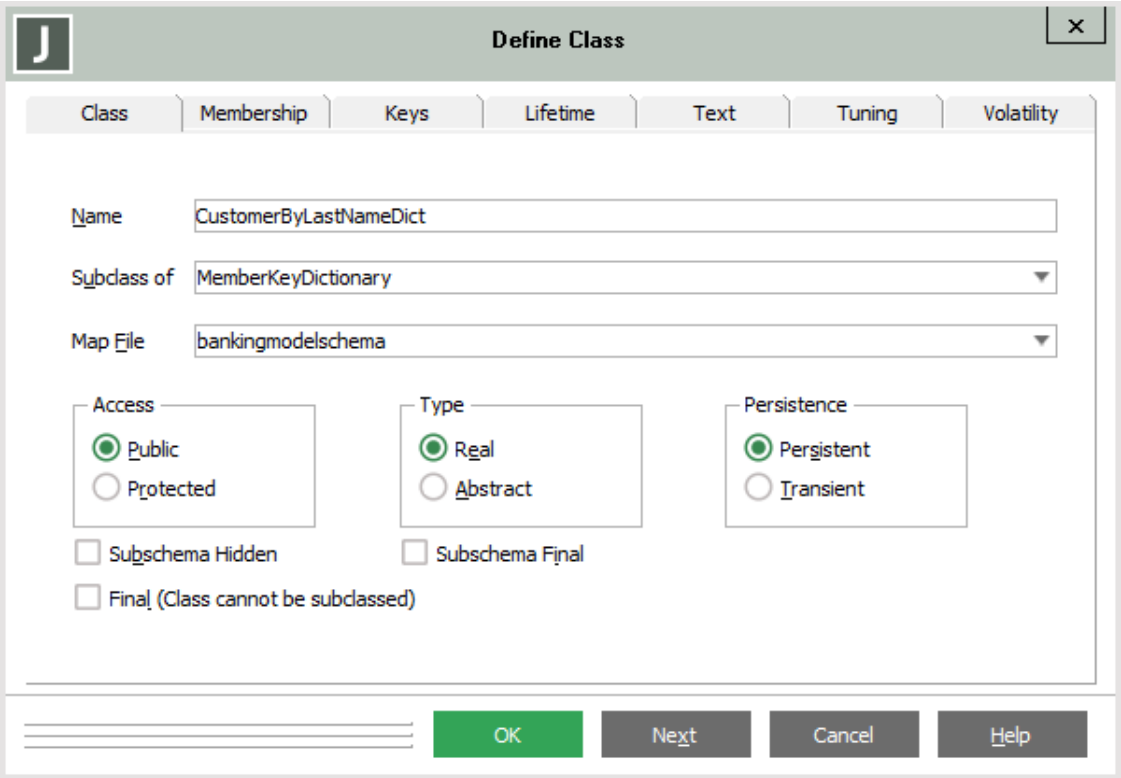
Methods are available for inserting and removing objects into an array. When these methods are executed, the other entries in the array are moved up or down automatically.

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.

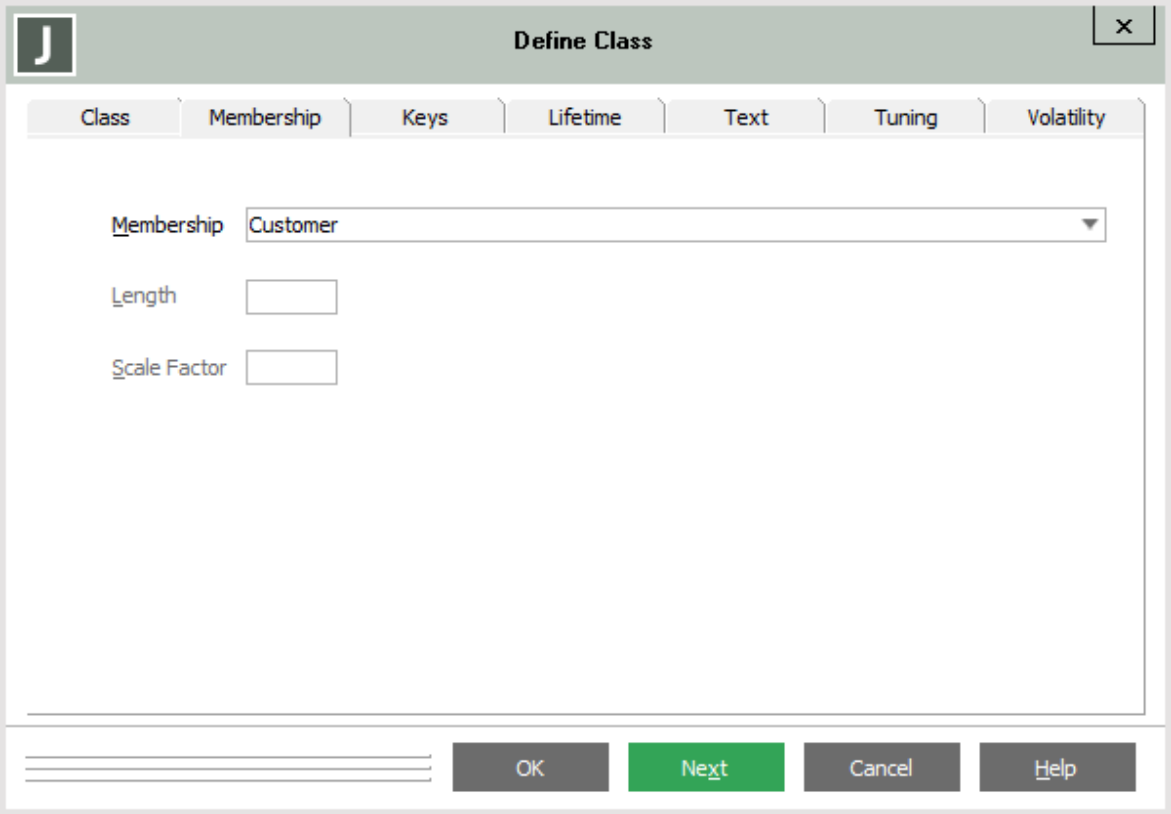
Exercise 9.1 – Adding a Customer Dictionary

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

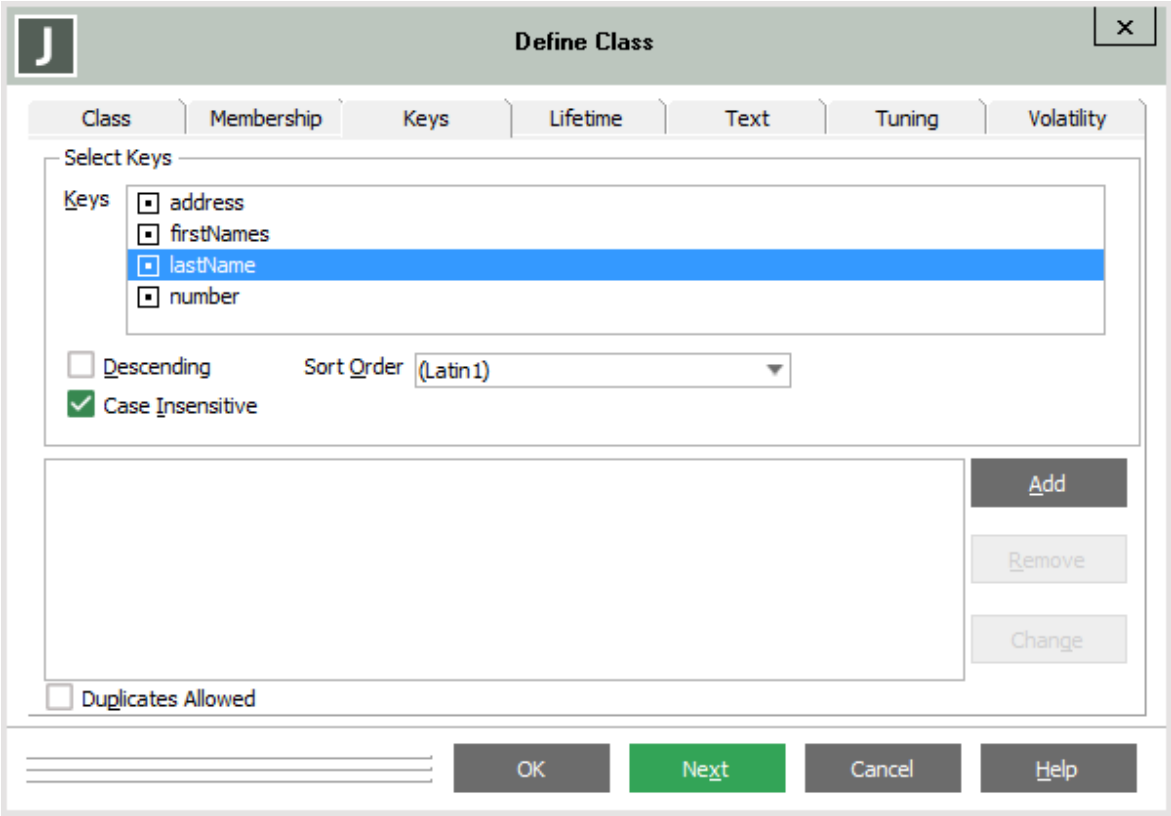
1. Find the **MemberKeyDictionary** class.
2. Add a class by selecting the Classes menu **Add Class** command.
   a. On the **Class** sheet, enter **CustomerByLastNameDict** as the name of the class, and then select the **Membership** sheet.



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



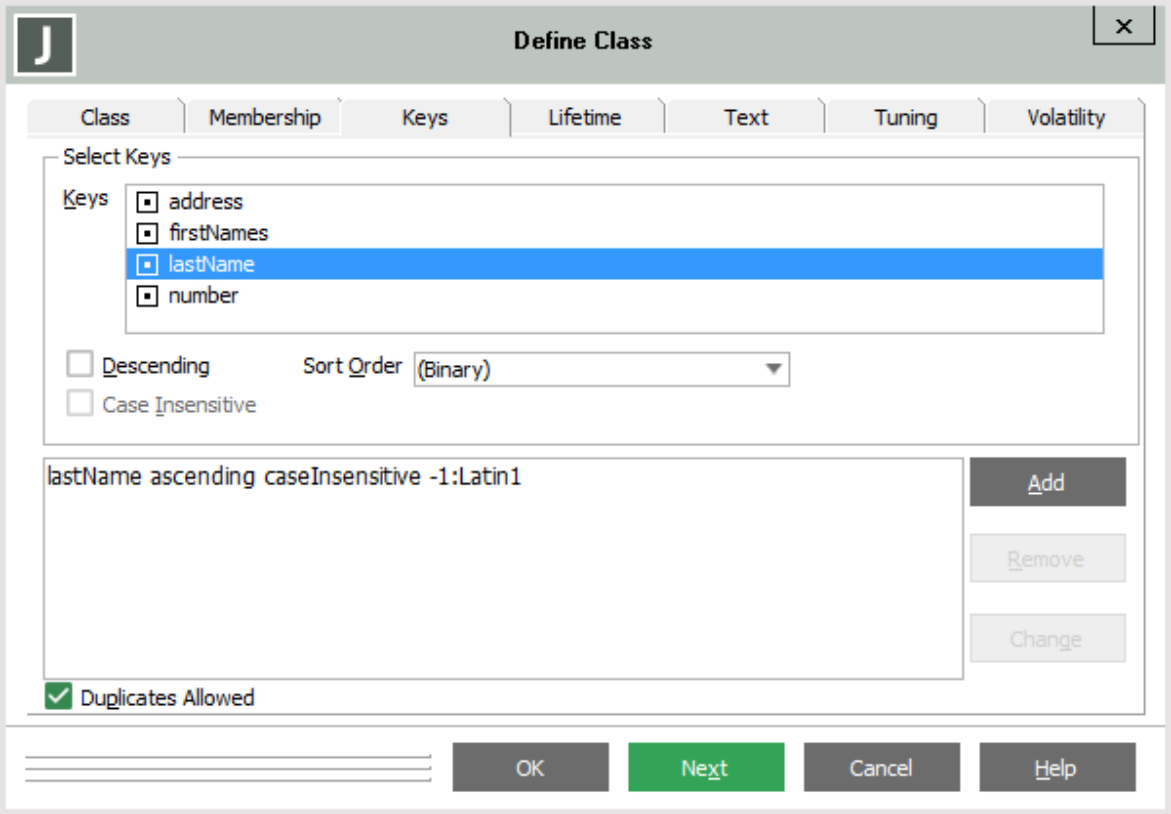
4. On the **Keys** sheet, select **lastName** as the key, select **Latin1** as the sort order, check the **Case Insensitive** check box, and then click the **Add** button.



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

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

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



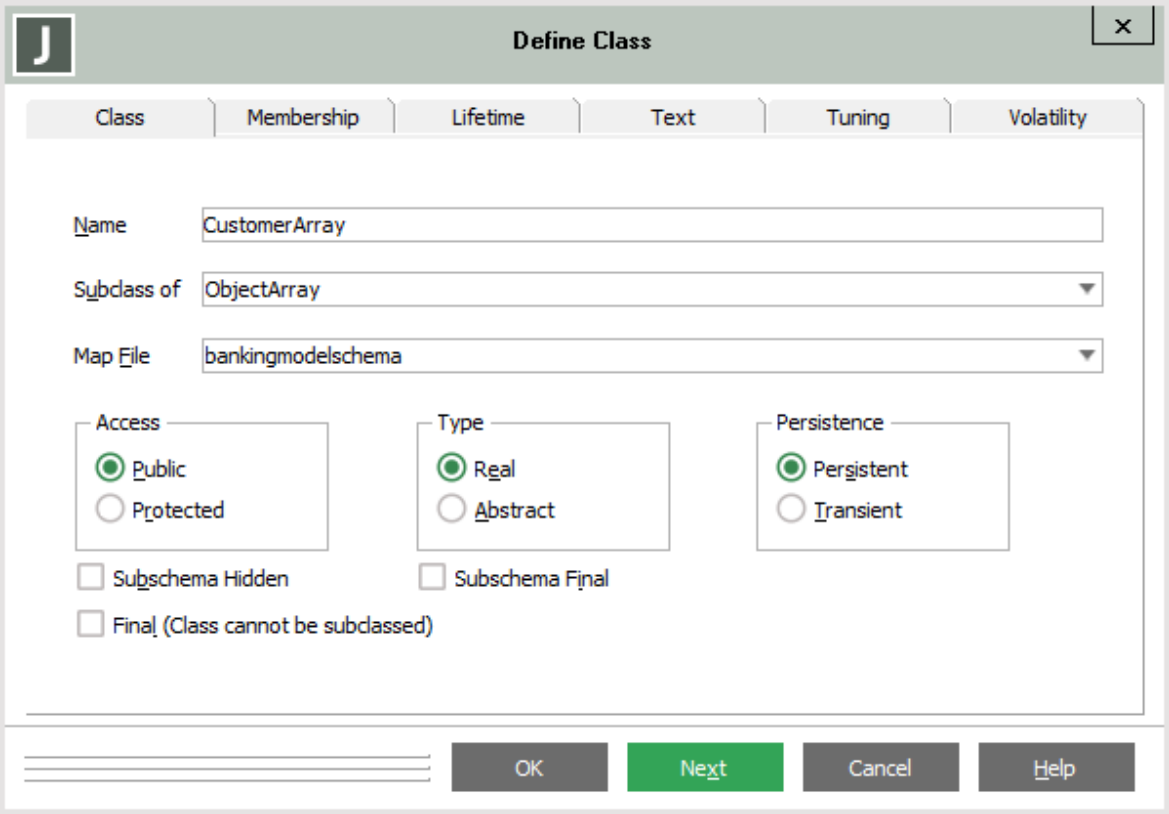
Tip Check the **Duplicates Allowed** check box if the selected keys are likely not to be unique.

Exercise 9.2 – Adding a Customer Array

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

1. Find the **ObjectArray** class.
2. Add a class by selecting the Classes menu **Add Class** 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** 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 – Populating a Collection

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

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

```
createCustomersFromFile();

vars
    dlg: CMDFileOpen;
    file: File;
    str: String;
    cust: Customer;
    dict: CustomerByLastNameDict;
begin
    app.initialize();
    create dlg transient;
    if not dlg.open() = 0 then
        return; // Exit as user did not select a file
    endif;
    beginTransaction;
    create dict persistent;
    commitTransaction;
    create file transient;
    file.fileName := dlg.fileName;
    while not file.endOfFile() do
        str := file.readLine();
        beginTransaction;
        cust := create Customer(str[41:end], str[16:25], str[1:15]);
        dict.add(cust);
        commitTransaction;
    endwhile;
epilog
    delete dlg;
    delete file;
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;
```

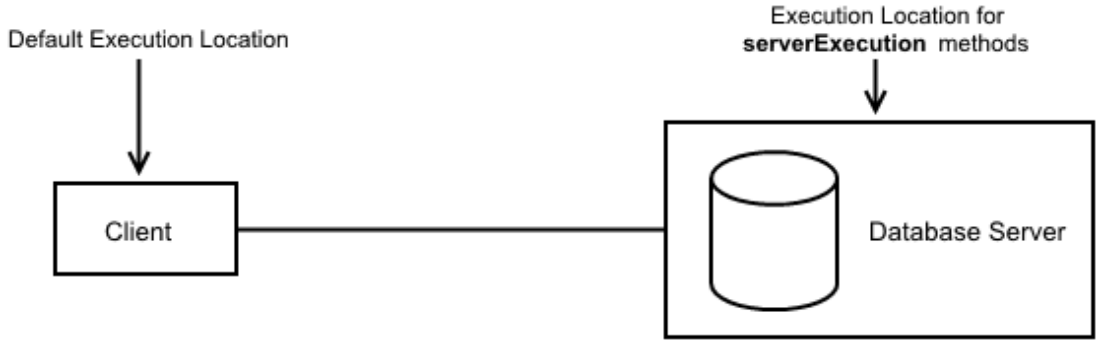
For a dictionary, you can set the start position for iteration by using one of the **startKey** family of methods.

```
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.

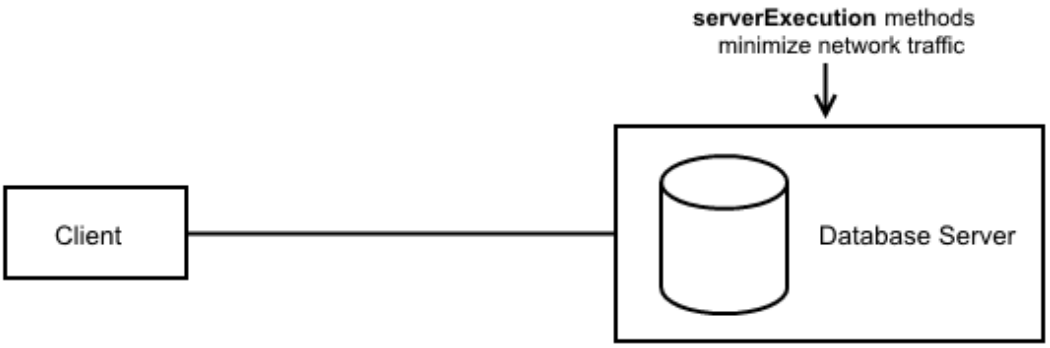
```
calledMethod01(parameters): returnType serverExecution;
```

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.

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 **clientExection** options have no effect.

Exercise 9.5 – 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.

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 := i + 1;
    endforeach;
    commitTransaction;
    write i.String & " customers deleted";
end;
```

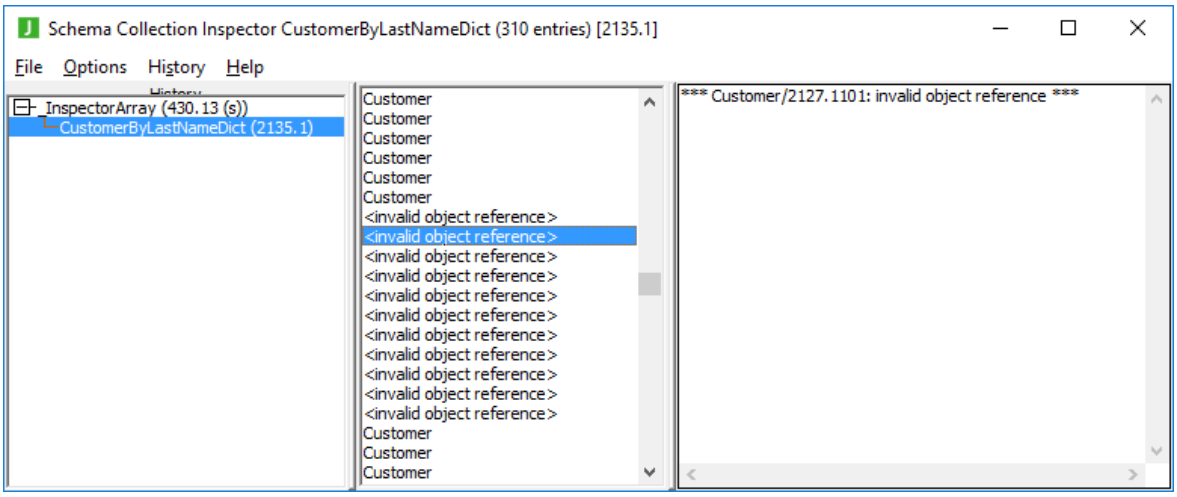
   In this method:

   - 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 9.6 – 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(): CustomerArray serverExecution;

vars
    dict: CustomerByLastNameDict;
    array: CustomerArray;
    cust: Customer;
begin
    dict := CustomerByLastNameDict.firstInstance();
    create array transient;
    foreach cust in dict where app.isValidObject(cust)
                          and cust.address = "Richmond" do
        array.add(cust);
    endforeach;
    write dict.size();
    write array.size();
    return array;
end;
```

3. Execute the method.

In this 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.)
- The **size** method demonstrates the reduced subset of objects that are to be processed on the client.

You could write a JadeScript method to call the **filter_Richmond_customers** method and process the collection that is returned.

Module 10 Relationships

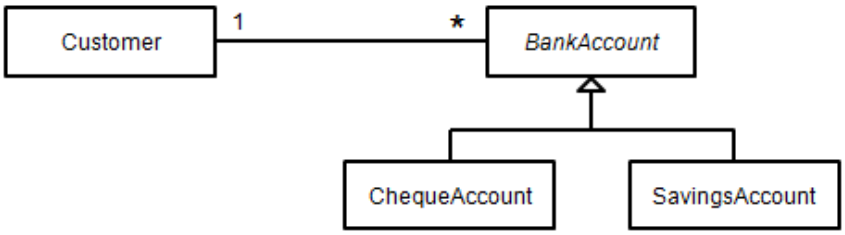
This module contains the following topics.

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.

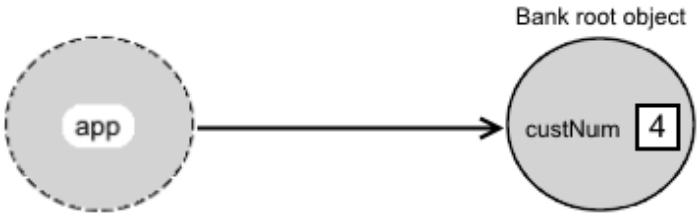
  The accounts can be cheque accounts, savings accounts, or other types that are added to the hierarchy later.



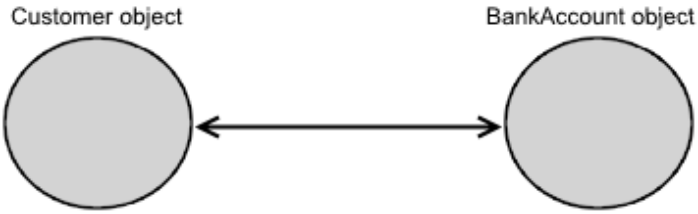
Relationships between classes are implemented using references. References enable you to:

- 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.

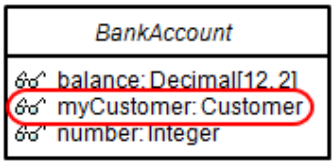


The one-to-many relationship enables navigation from a customer object to a bank account owned by the customer, and in the other direction.



myCustomer Reference

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

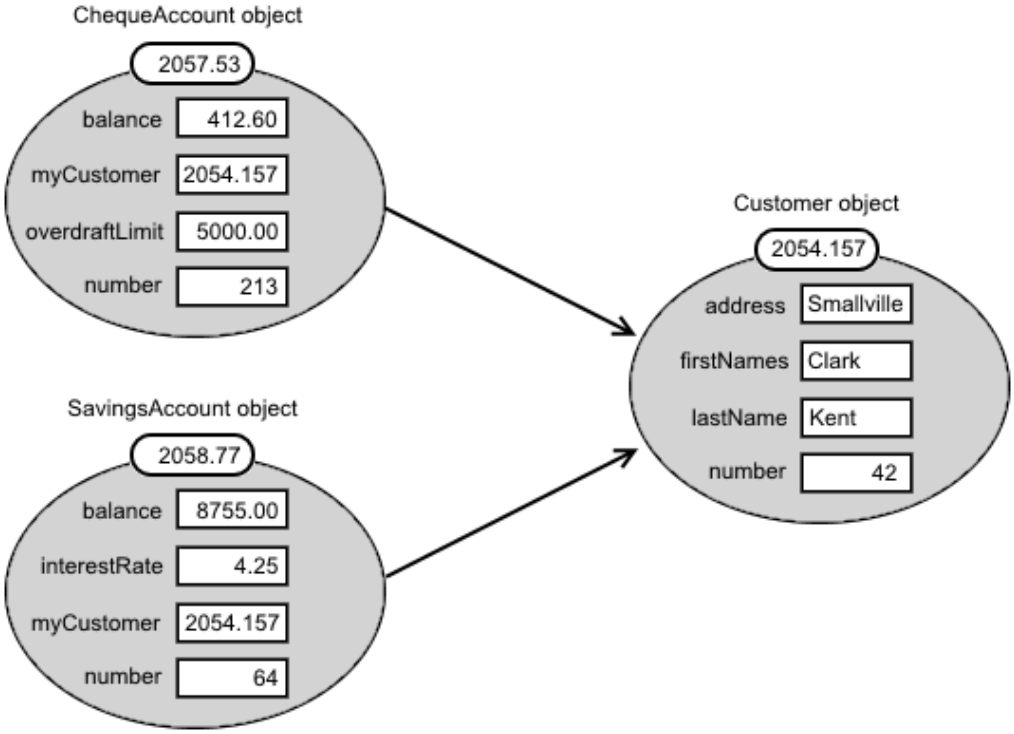


By convention, a reference name starting with **my** is a reference to a single object. In this case, the **BankAccount** object references the **Customer** object who owns the bank account. When a customer is created, the **myCustomer** reference is null.

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.

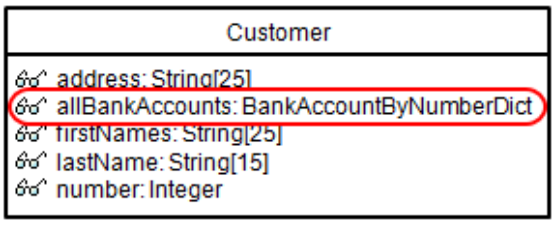


The **myCustomer** reference enables you to navigate from a bank account to the customer who owns the bank account.

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

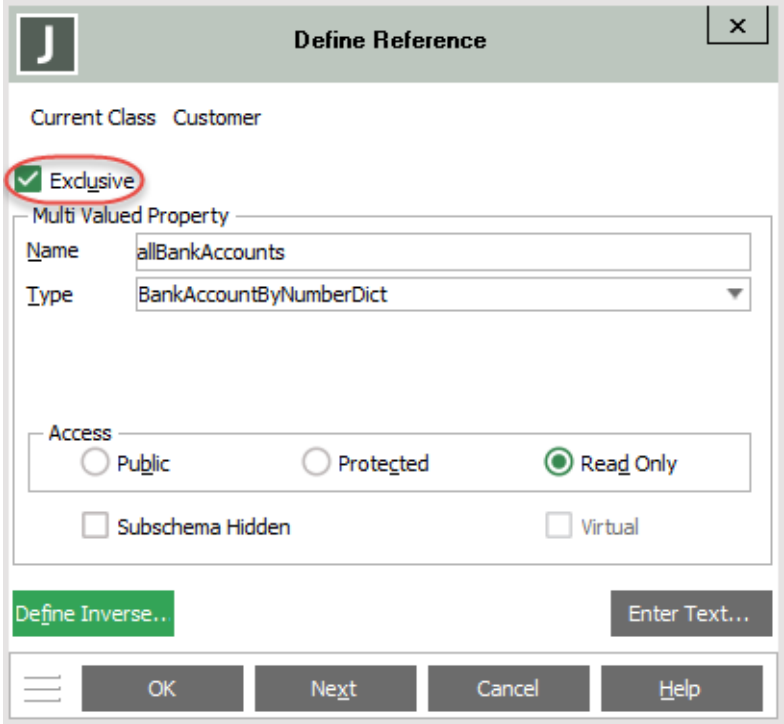
An exclusive collection is one that belongs exclusively to a parent object. Conceptually, the exclusive collection is created when the parent object is created, and deleted when the parent object is deleted. A customer can have any number of bank accounts of different types. This can be implemented by 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.



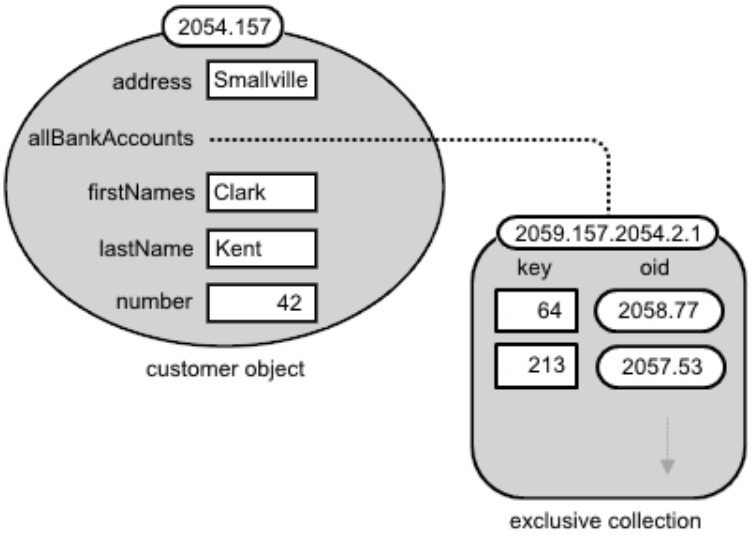
The naming convention used in this course is as follows.

- 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.

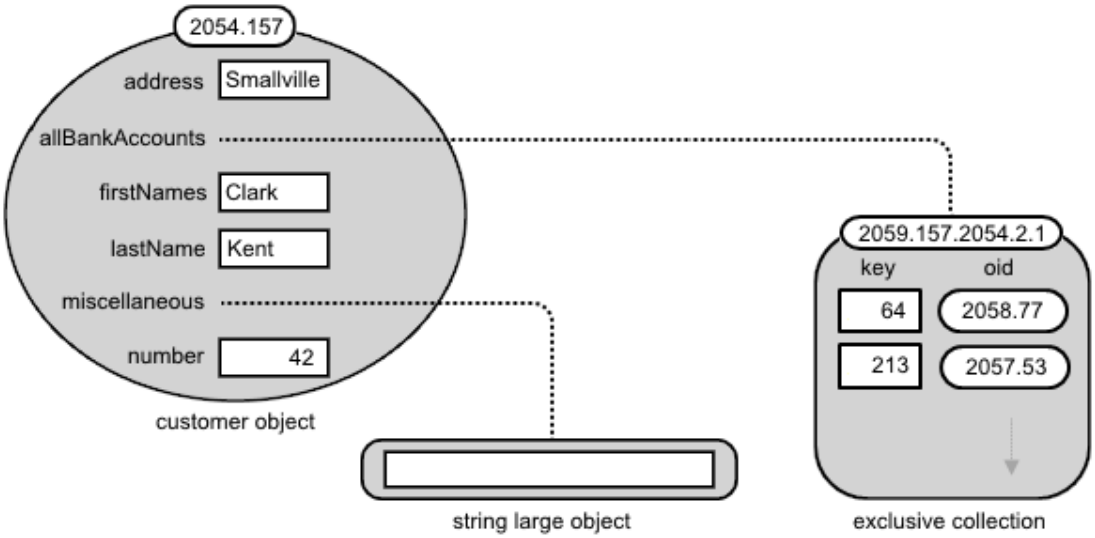


Other Subobjects

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.

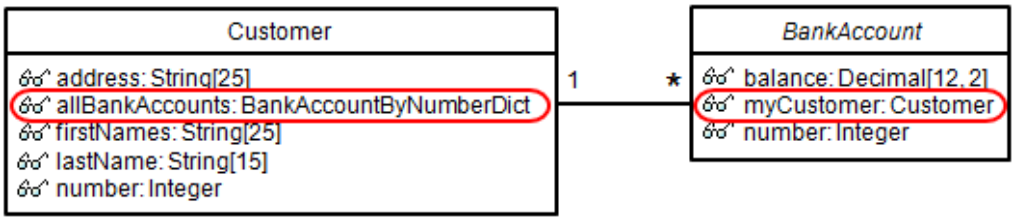


Although you can think of subobjects being created at the same time as the parent object, in reality they are not created until the first time they are used. In addition, subobjects are not fetched from the database unless they are being accessed in code.

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;
```

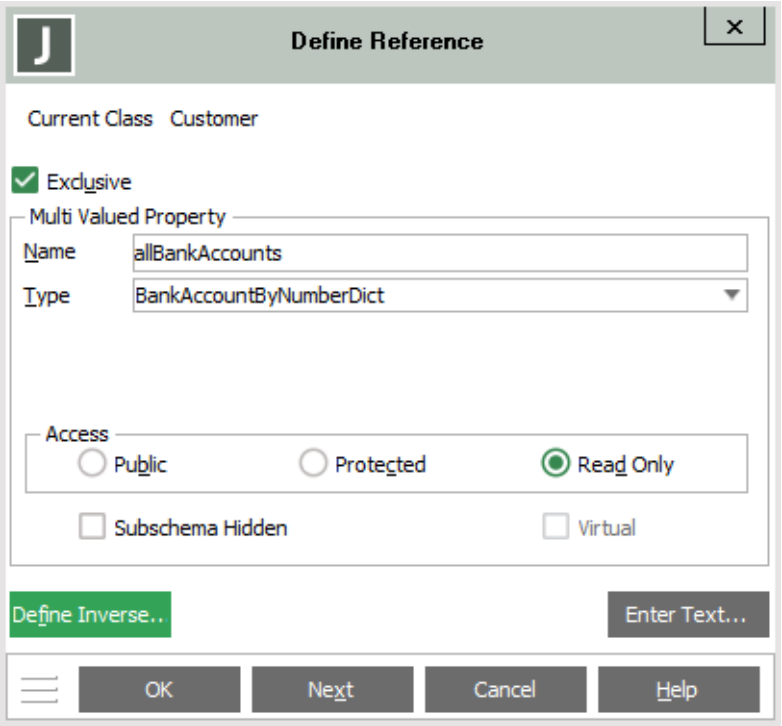
Note Deletions no longer result in collections with *invalid object references*, as they did before.

Adding Both Inverse References

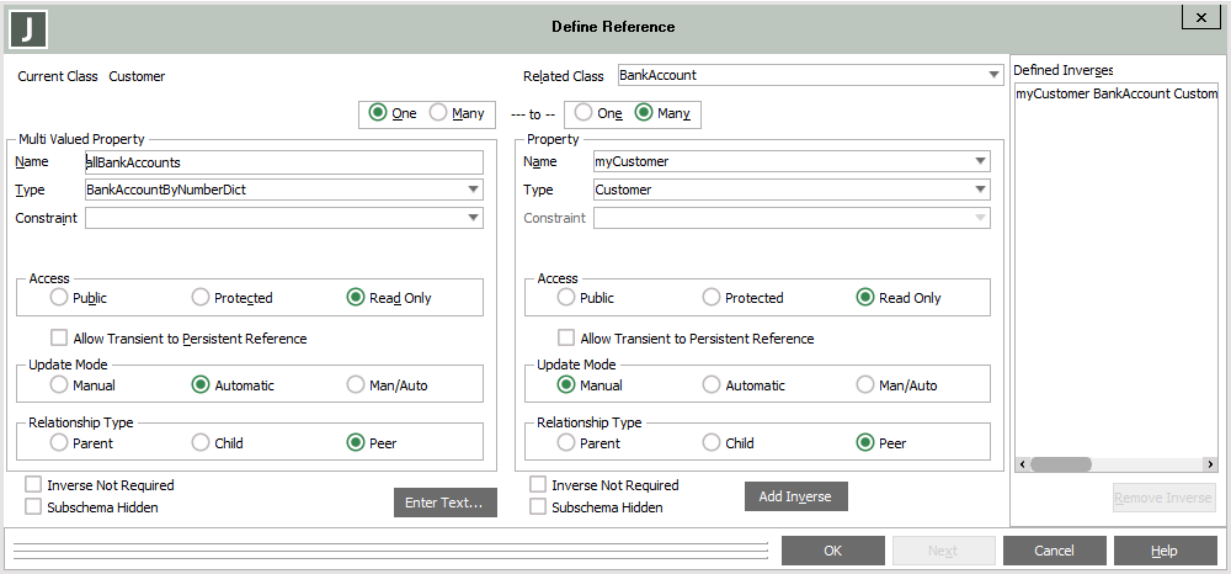
The one-to-many relationship between the **Customer** and **BankAccount** classes has been defined in the following three separate stages.

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.

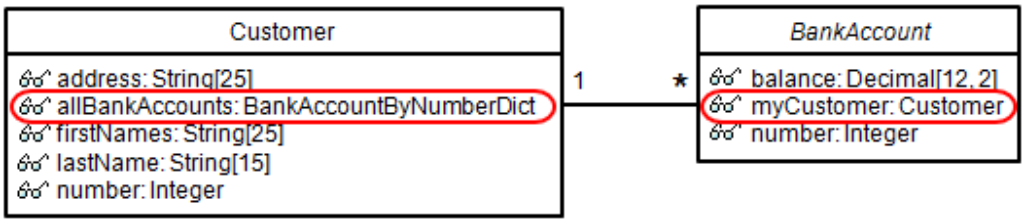


When the **Define Inverse** button is clicked, the dialog expands to show the related **BankAccount** class next to the **Customer** class. This enables you to add both inverse references at the same time.



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

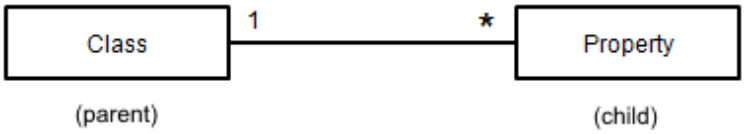
Peer-to-peer and parent-child relationships specify whether deleting one object causes related objects to be deleted.

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.



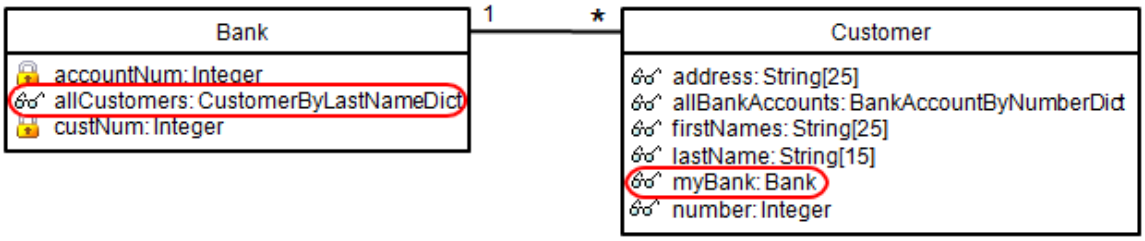
The **Customer** class object is the parent of the **address**, **firstNames**, and **lastName** property objects. If you were to remove the **Customer** class, the associated property and method objects would be deleted automatically.

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.



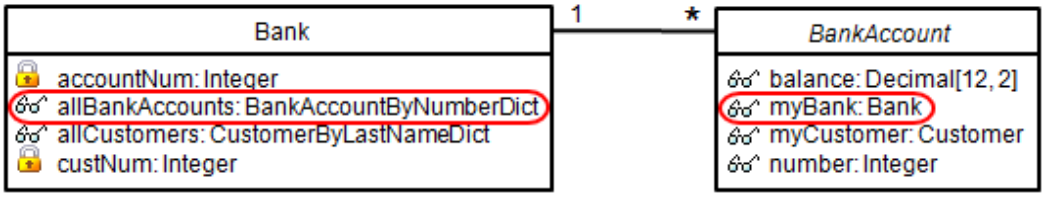
After defining the inverse references, a coding change is required to ensure that the **myBank** reference is set for a new customer. This can be done in the **create** method in the **Customer** class, as follows.

```
create(addr, first, last: String) updating;

begin
    self.number := app.myBank.nextCustNum();
    self.address := addr.trimBlanks();
    self.firstNames := first.trimBlanks();
    self.lastName := last.trimBlanks();
    self.myBank := app.myBank;
end;
```

Note There is a general rule to set references after setting attributes. In the **create** 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, 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 **create** methods in the **ChequeAccount** and **SavingsAccount** classes, as follows.

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

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

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

begin
    self.balance := bal;
    self.interestRate := rate;
    self.myCustomer := cust;
    self.myBank := app.myBank;
end;
```

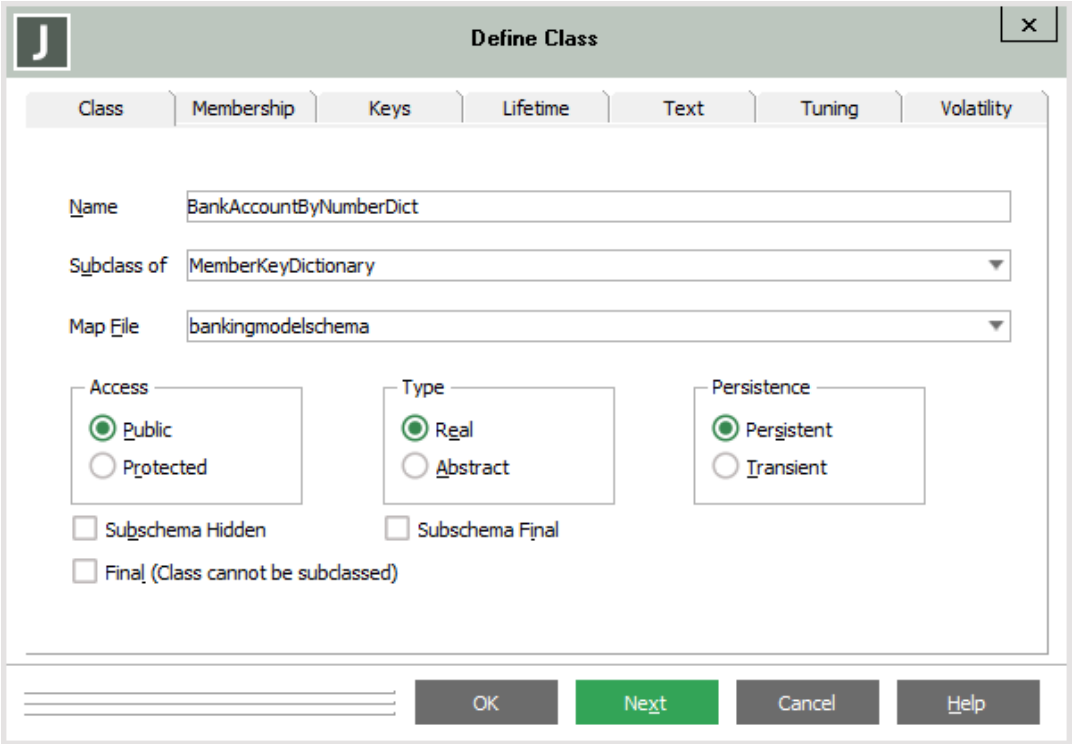
Exercise 10.1 – Adding a BankAccount Dictionary

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.

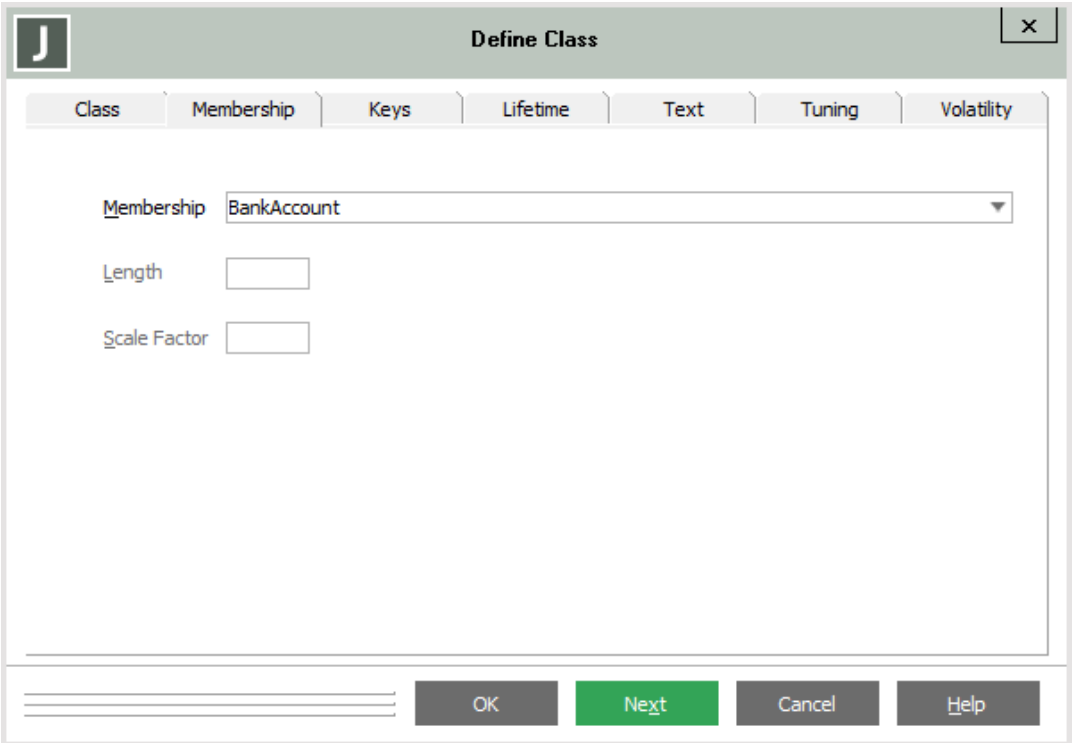
1. Find the **MemberKeyDictionary** class.

2. Add a class, by selecting the Classes menu **Add Class** command.

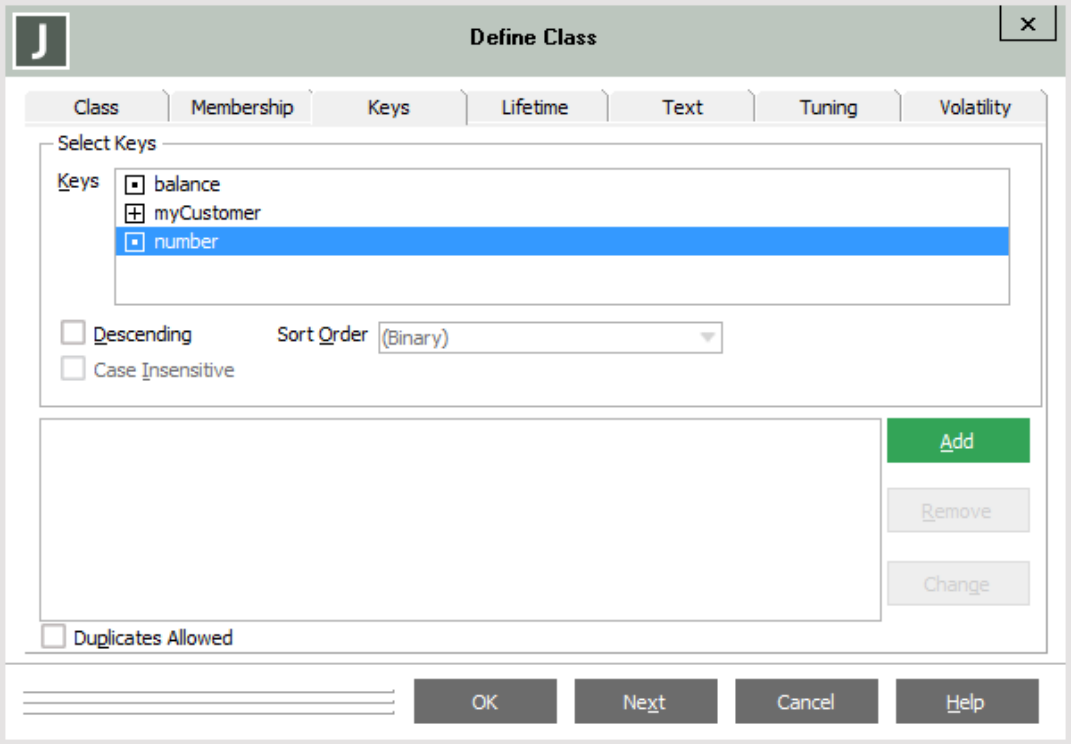
3. On the **Class** sheet, enter **BankAccountByNumberDict** as the name of the class and then select the **Membership** sheet.



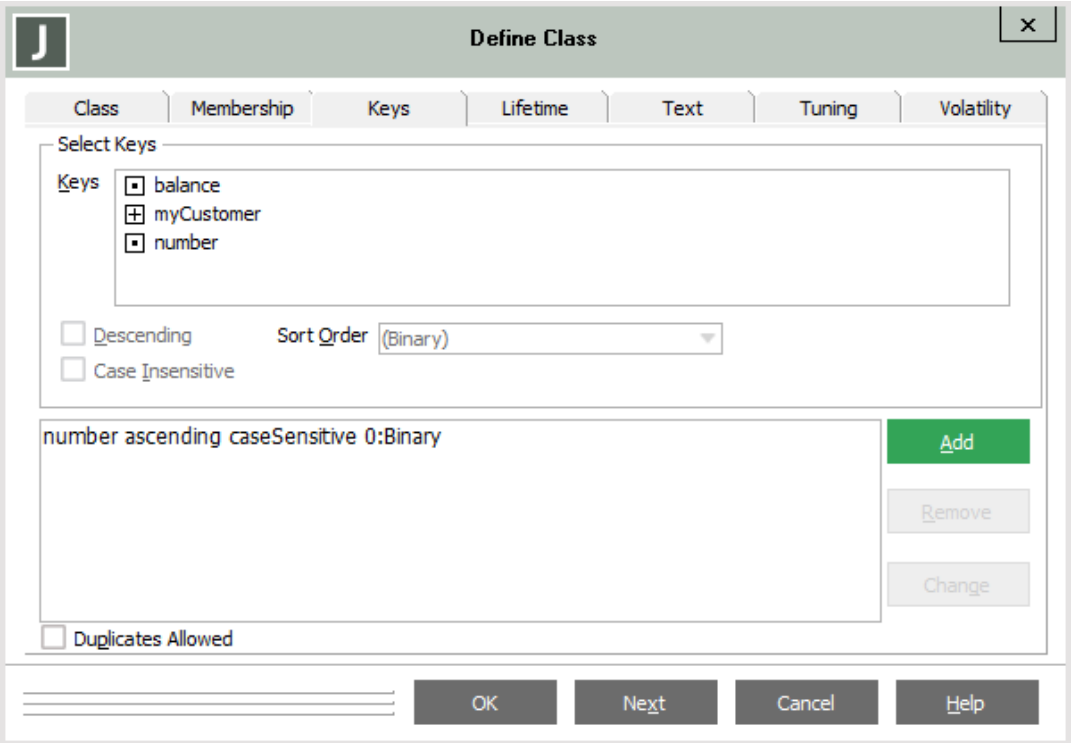
4. On the **Membership** sheet, select **BankAccount** as the **Membership** class and then select the **Keys** sheet.



5. On the **Keys** sheet, select **number** as the key and then click the **Add** button.



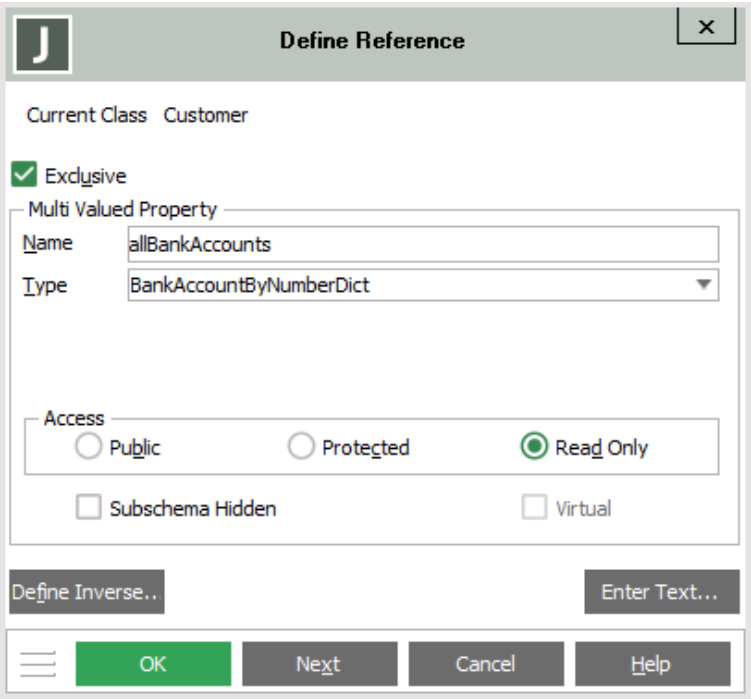
6. Click the **OK** button.



Exercise 10.2 – Adding an Exclusive Collection

In this exercise, you will add an **allBankAccounts** reference.

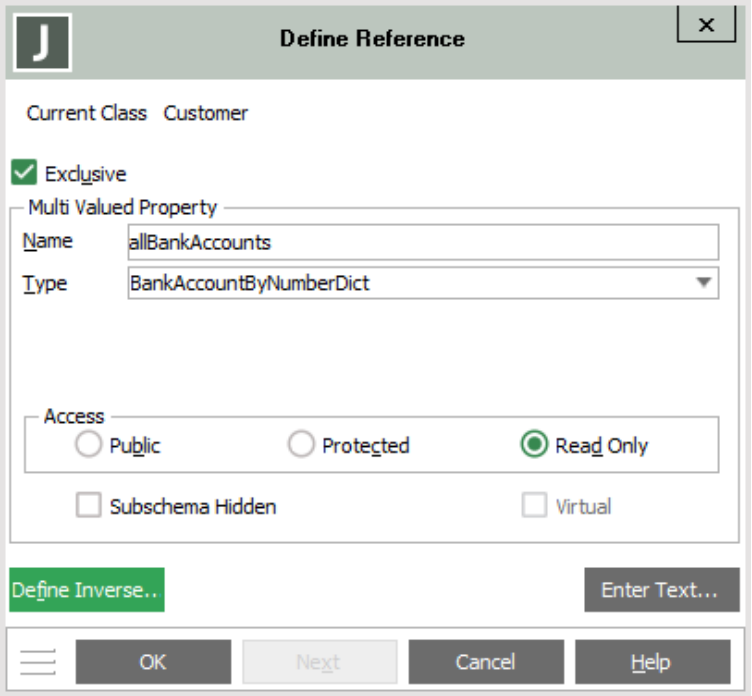
1. Select the **Customer** class.
2. Add a reference by selecting the Properties menu **Add Reference** command.
   a. Enter **allBankAccounts** as the name, make the reference read-only, and then click the **OK** button.



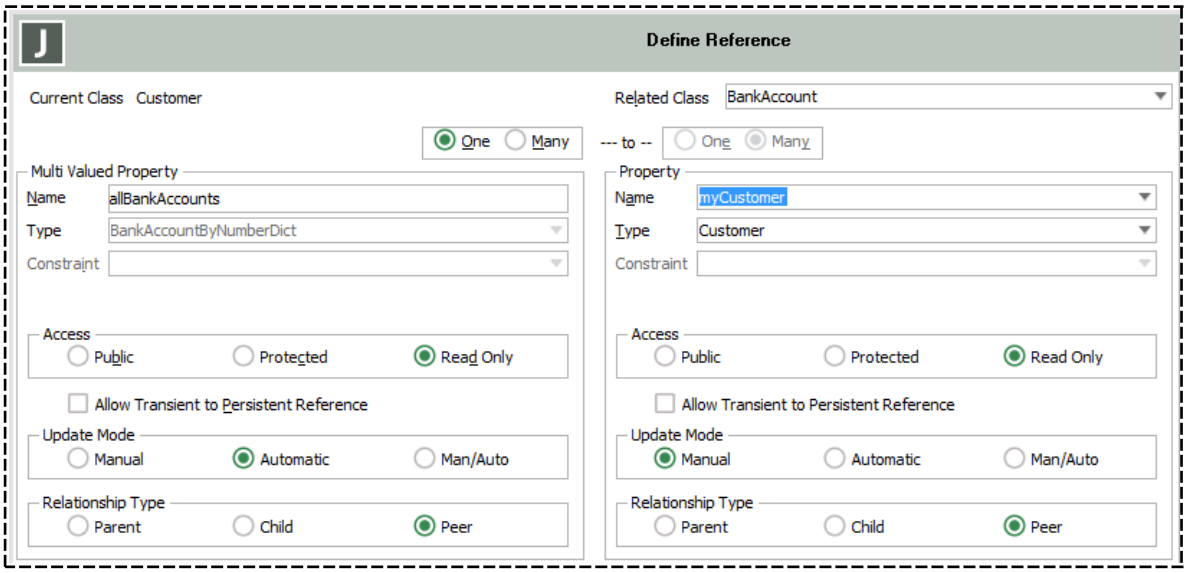
Exercise 10.3 – Adding Inverse References

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.

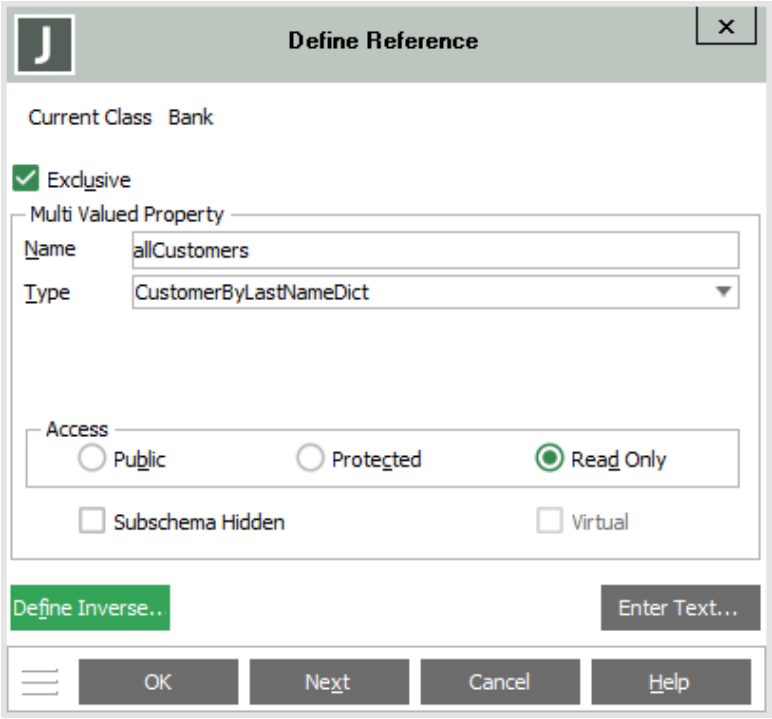


Exercise 10.4 – Adding Root Object Collections

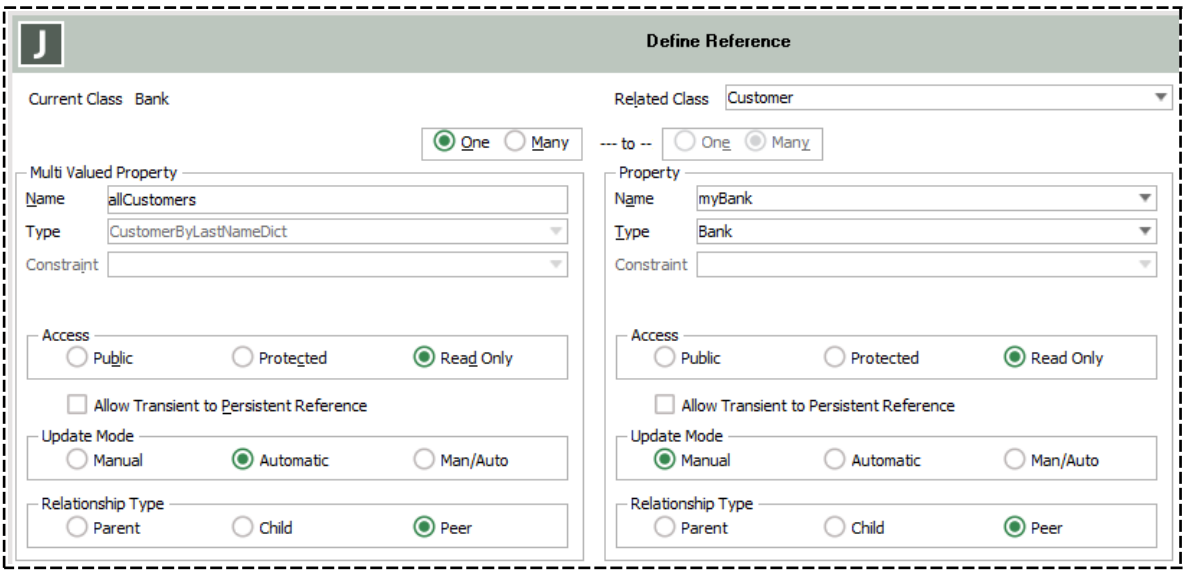
In this exercise, you will add the root object collections of **Customer** and **BankAccount** objects. You will also change the **create** methods for these classes so that new instances are automatically added to these collections.

1. Select the **Bank** class.

2. Add a reference called **allCustomers** of type **CustomerByLastNameDict** class, and then click the **Define Inverse** button.



3. In the **Customer** class, enter **myBank** as the reference name and then click the **OK** button.

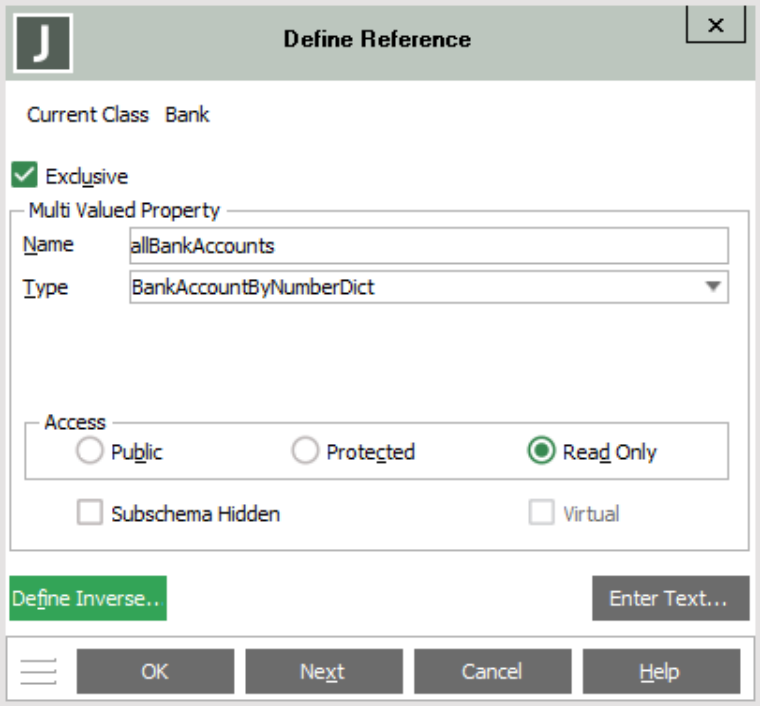


4. Select the **create** method in the **Customer** class. Add an instruction to set the **myBank** reference to the root object, as follows.

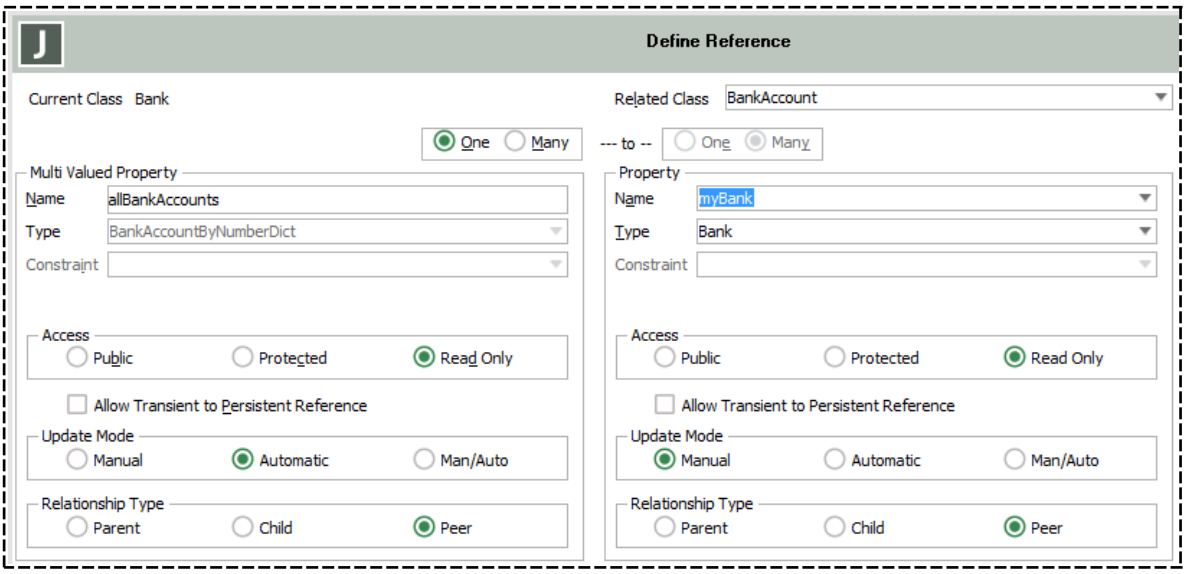
```
create(addr, first, last: String) updating;

begin
    self.address := addr.trimBlanks();
    self.firstNames := first.trimBlanks();
    self.lastName := last.trimBlanks();
    self.myBank := app.myBank;
end;
```

5. Select the **Bank** class.

6. Add a reference called **allBankAccounts** of type **BankAccountByNumberDict** class and then click the **Define Inverse** button.



7. In the **BankAccount** class, enter **myBank** as the reference name and then click the **OK** button.



8. Select the **create** method in the **ChequeAccount** class. Add an instruction to set the **myBank** reference to the root object, as follows.

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

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

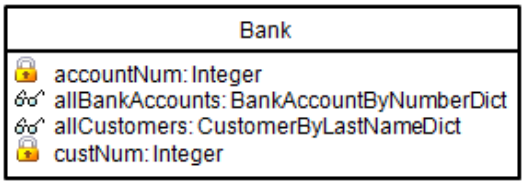
9. Select the **create** method in the **SavingsAccount** class. Add an instruction to set the **myBank** reference to the root object, as follows.

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

begin
    self.balance := bal;
    self.interestRate := rate;
    self.myCustomer := cust;
    self.myBank := app.myBank;
end;
```

Exercise 10.5 – Multiple Inverses

At this stage, the **Bank** root object has two collections, as follows.

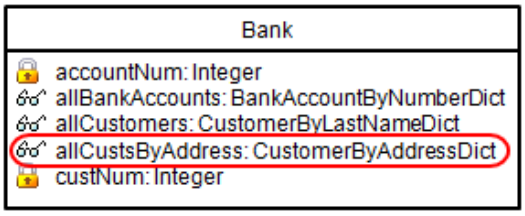
- A collection of bank accounts ordered by number
- A collection of customers ordered by last name



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 **allCustsByAddress**, 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.

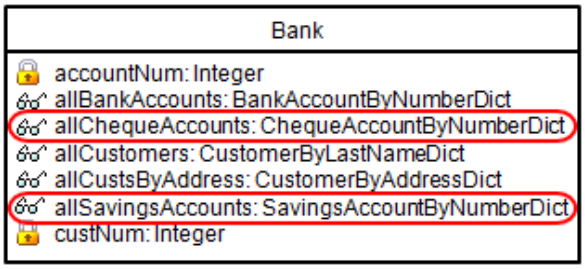


When the **myBank** reference is set for a new customer, the customer is added to the **allCustomers** collection and the **allCustsByAddress** collection.

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.

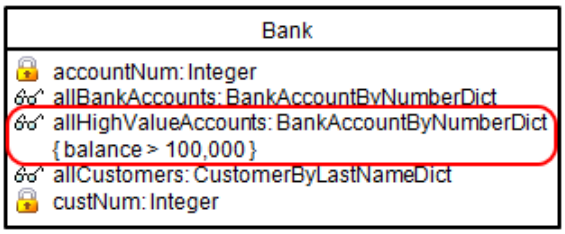
```
highValue(): Boolean condition;

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 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.

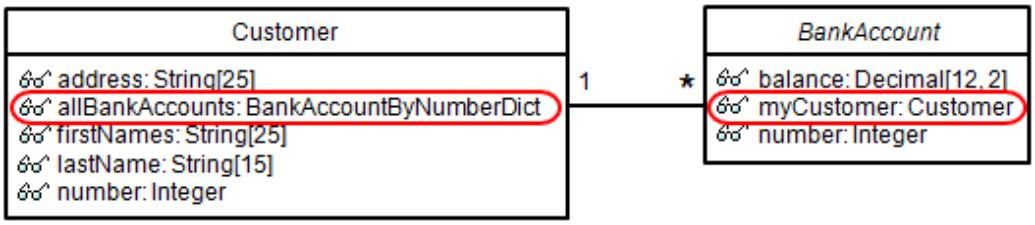


When an account is created, depending on the initial balance, inverse maintenance adds it to the **allHighValueAccounts** collection. Subsequently, as the balance changes through deposits and withdrawals, the bank account will be removed automatically from or added to the collection, depending on whether the condition is met.

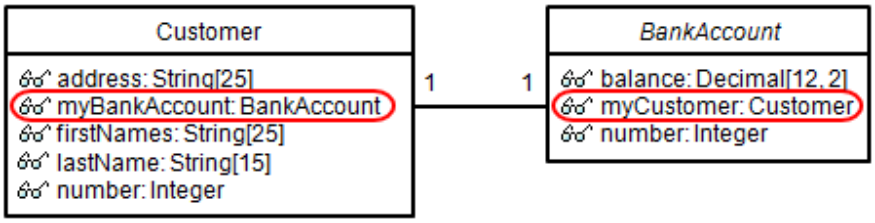
Cardinality

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.

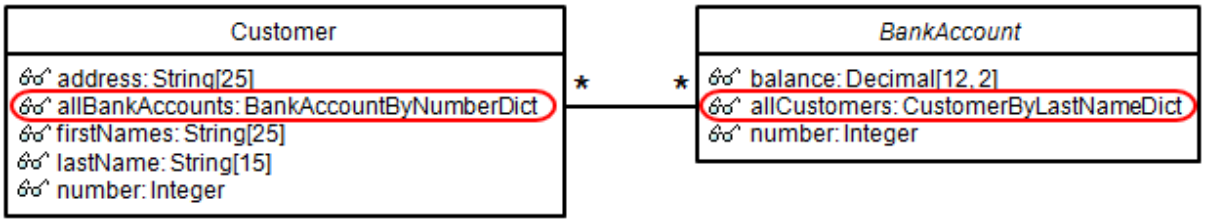


A one-to-one relationship has **my** references at both ends. No collections are required.



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.

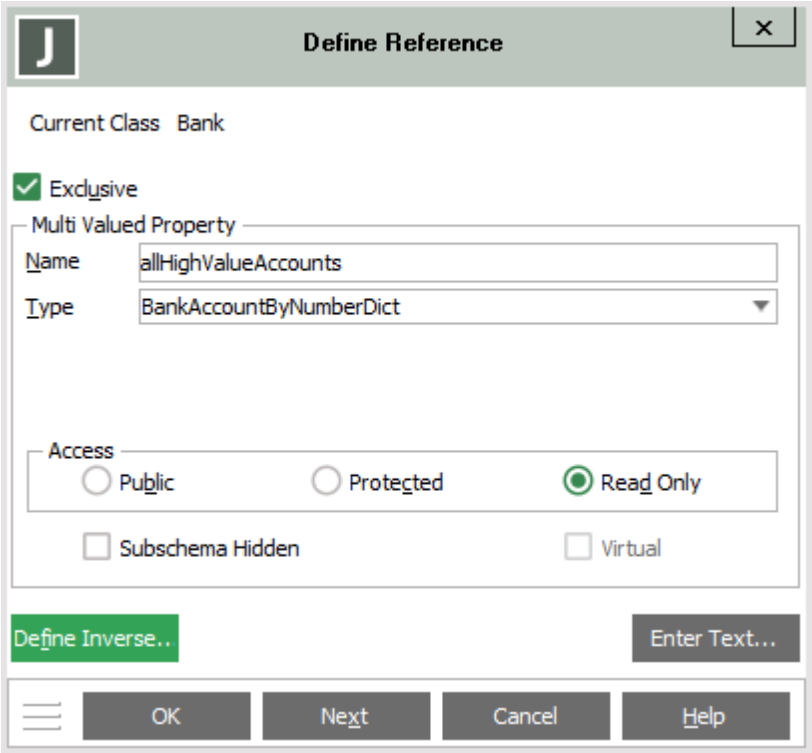
Exercise 10.6 – Adding an allHighValueAccounts Collection

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.

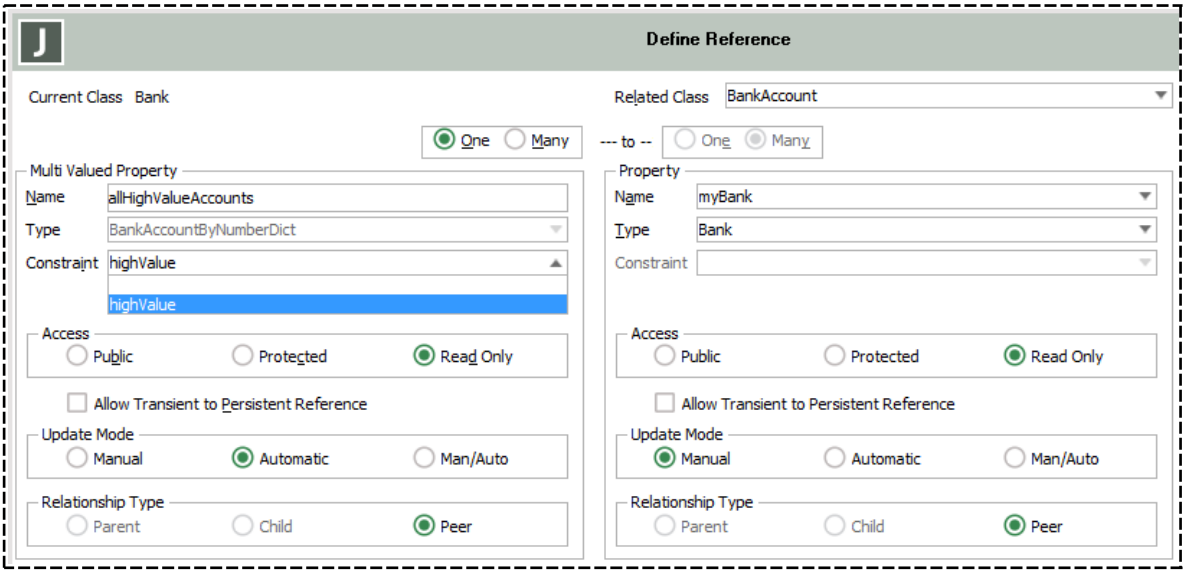
1. Select the **BankAcount** class.
2. Add a condition called **highValue**, by selecting the Methods menu **New Condition**command.
3. Code the method as follows.

```
highValue(): Boolean condition;

begin
    return self.balance >= 100000;
end;
```

4. Add a reference called **allHighValueAccounts** 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 diagram.



6. 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
    cheque: ChequeAccount;
begin
    app.initialize();
    beginTransaction;
    create cheque persistent;
    cheque.setPropsOnCreate(0, 0, null);
    commitTransaction;
    write app.myBank.allHighValueAccounts.size();   // Outputs 0

    beginTransaction;
    cheque.deposit(100000);
    commitTransaction;
    write app.myBank.allHighValueAccounts.size();   // Outputs 1

    beginTransaction;
    cheque.withdraw(1);
    commitTransaction;
    write app.myBank.allHighValueAccounts.size();   // Outputs 0
end;
```

7. Execute the JadeScript method.

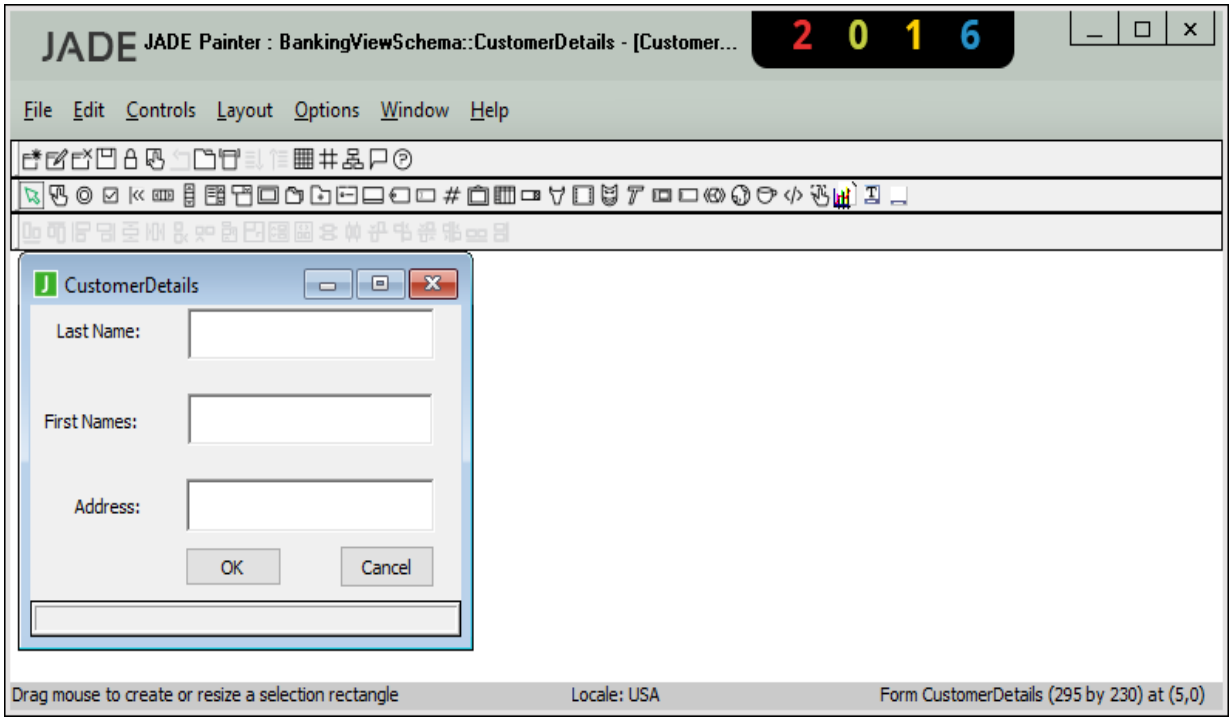
Module 11 Forms

This module contains the following topics.

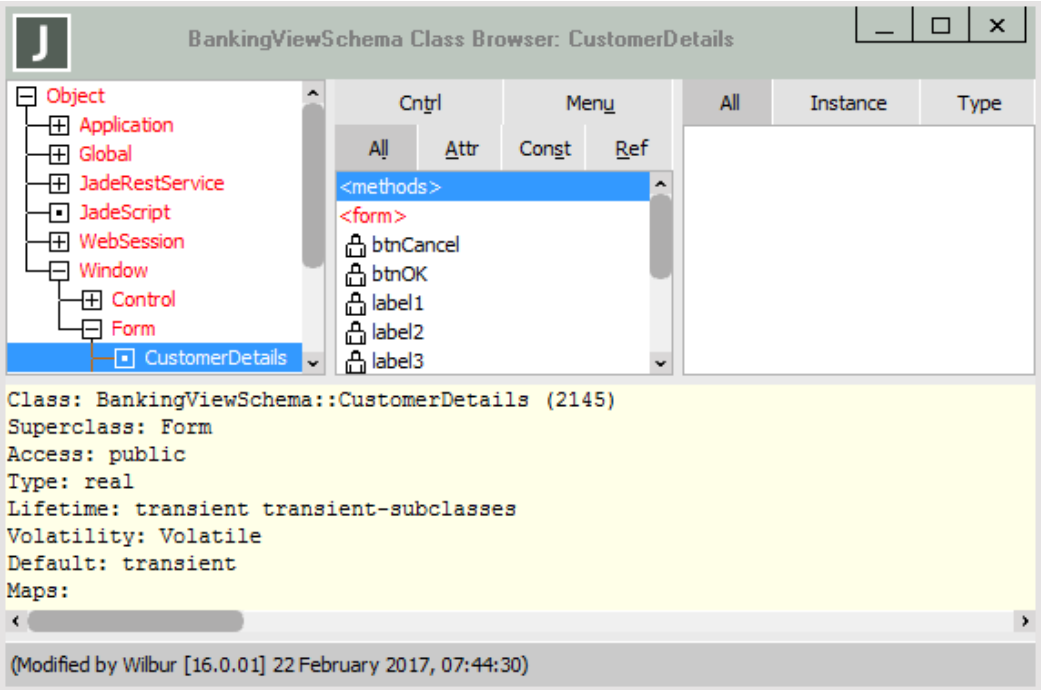
Introduction

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 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 install files contains runtime skins that you can load. For details about loading the **SampleSkins.ddb** 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]
```

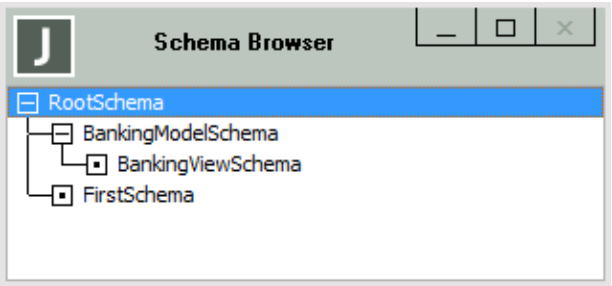
In addition, any subsequent forms opened in the JADE Painter are displayed using the selected runtime skin. The selected skin is saved in your user preferences when you close the JADE Painter and restored when you re-open the Painter.

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 environment.

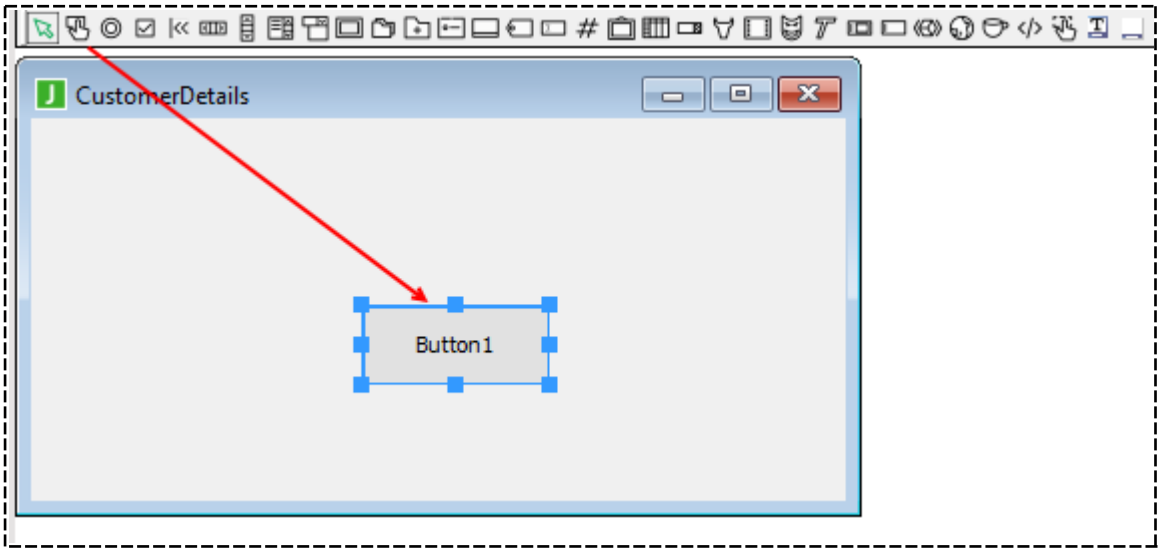
Separating the views from the model by packaging them in their own schemas prevents the model schema from becoming cluttered with user interface implementation, and means that the model schema can support many different views. It also makes it easier to identify the services provided by the model.

Create forms in a subschema (the **BankingViewSchema**, in this course).



Painter

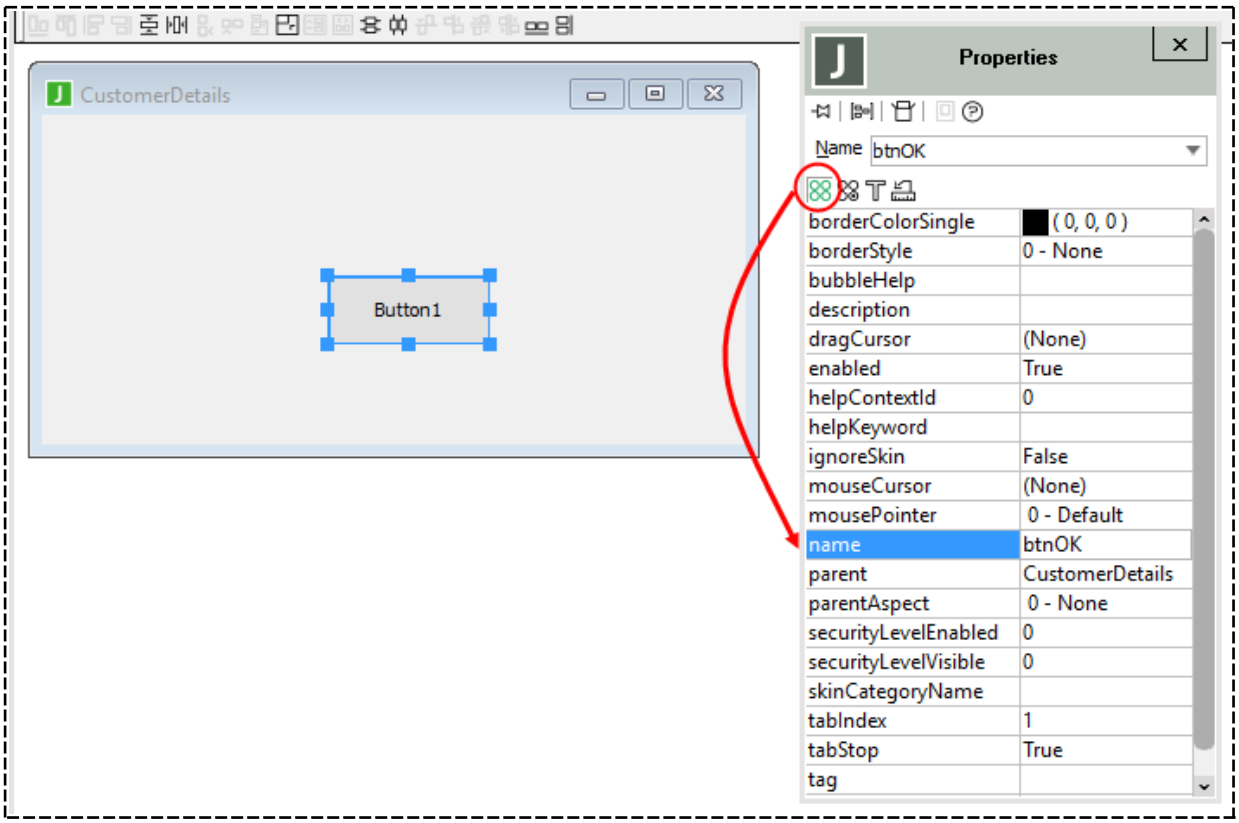
To add a control to a form, click on the control in the **Tools** palette and then click on the form. Alternatively, use the Ctrl+Insert shortcut keys to display a text-based list of the controls that are available to be added.



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

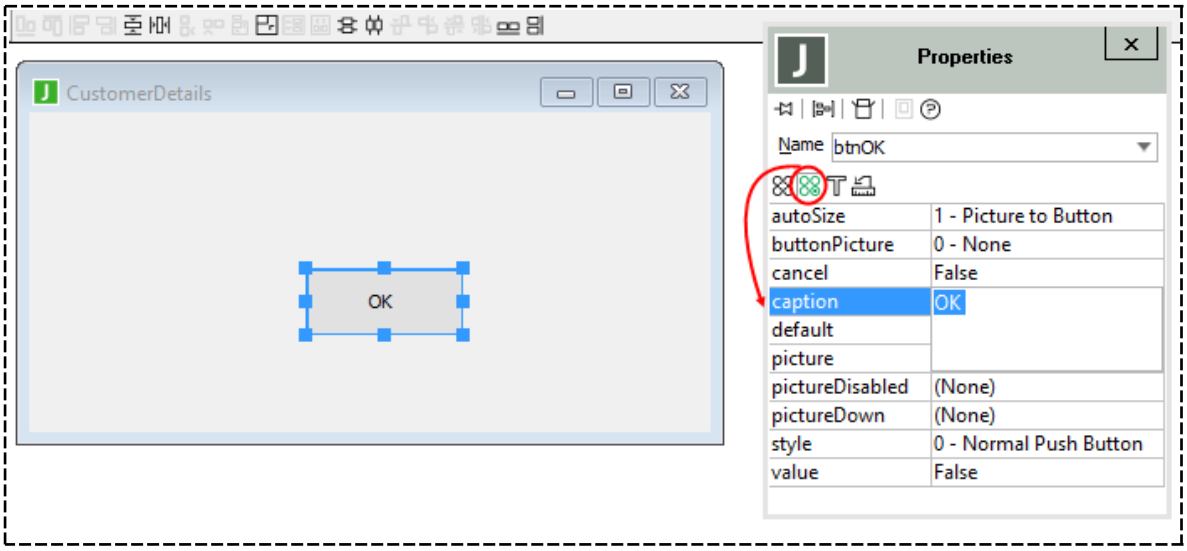
The **name** property is in the **Common** group of properties. The **Common** properties are those that every type of control has; for example, every control has a name. You use the **name** property when referring to the control in your code. You should change the default names **button1**, **button2**, and so on, to something more meaningful to a developer.



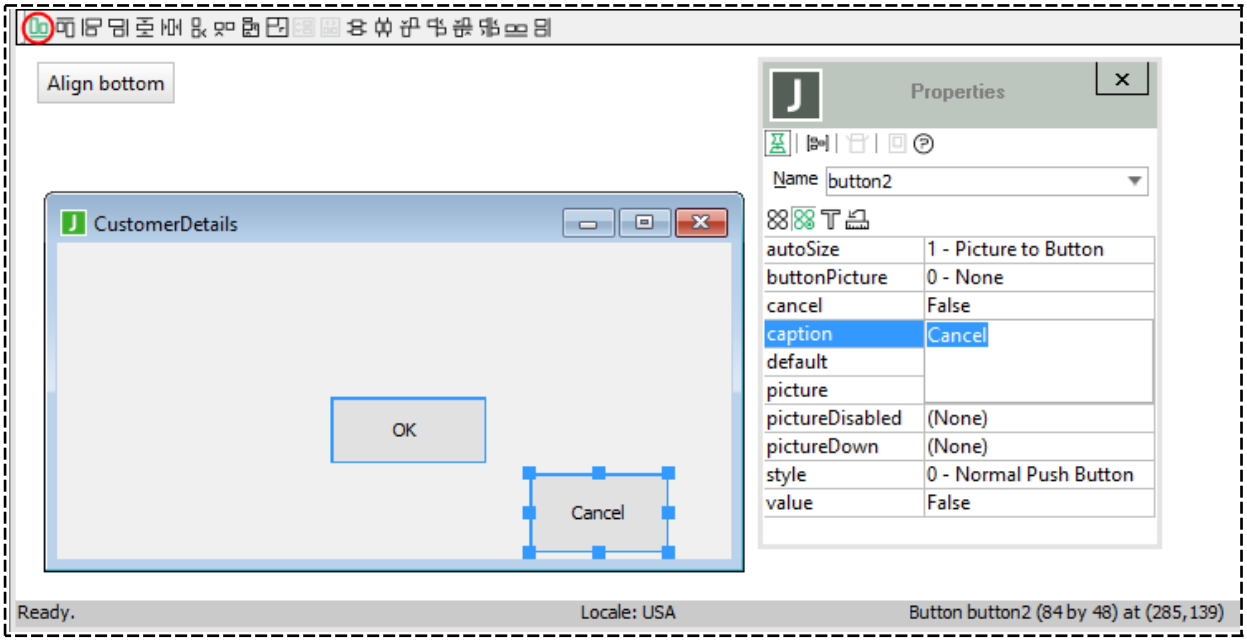
Tips Click the **Stay on top of Painter** icon at the left of the Properties dialog toolbar, to keep the Properties dialog positioned on top of the Painter. The icon then changes shape and is highlighted.

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 JADE 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 when you select the **Show 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.

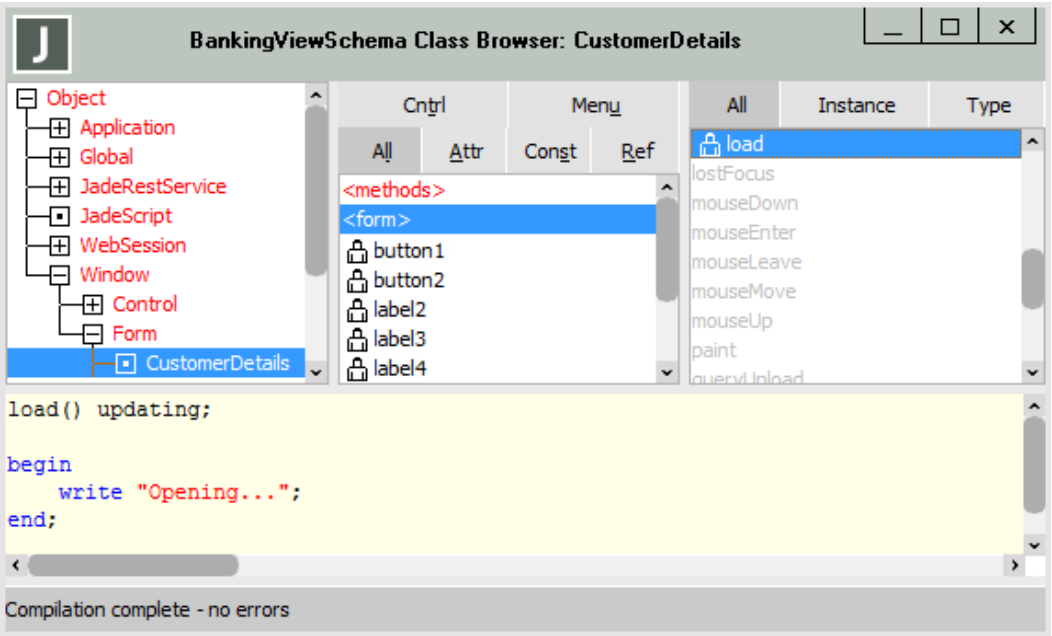
```
vars
    form: CustomerDetails;
begin
    create form transient;
    form.show();
    // Wait five seconds
    app.doWindowEvents(5000);
    form.unloadForm();
end;
```

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.

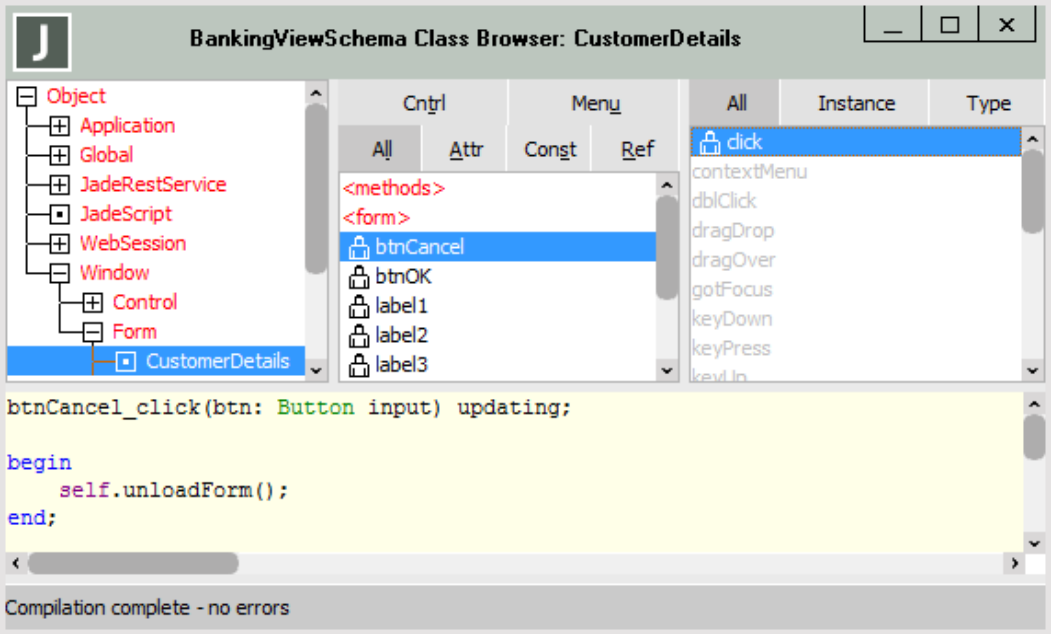
To code one of these event methods, select **<form>** in the central window (that is, the Properties List) and then select the appropriate event method from Methods List on the right.



Buttons

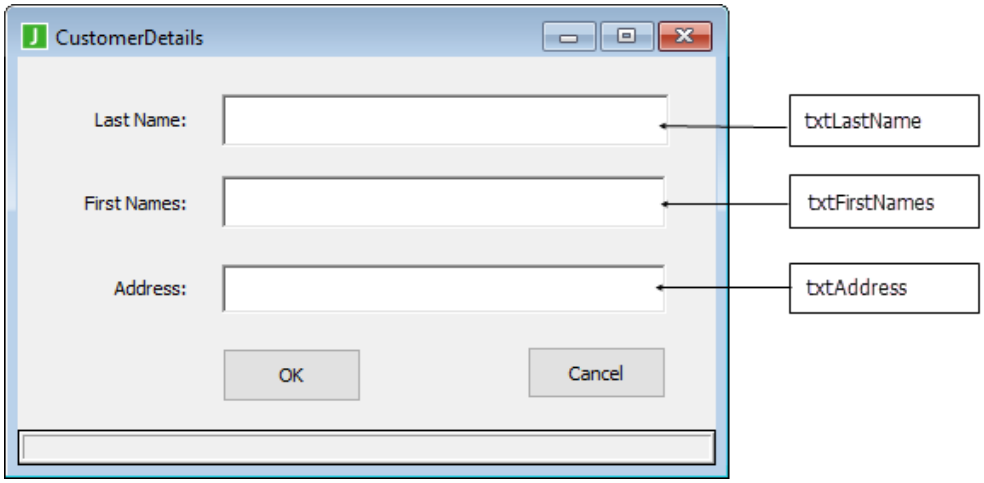
In a GUI application, most of the functionality is triggered when the application user clicks buttons on forms. To code a button **click** event method, select the button control in the central Properties List and then select the **click** event method from the Methods List on the right.

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() protected;

begin
    txtLastName.text := "";
    txtFirstNames.text := "";
    txtAddress.text := "";
    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 txtLastName.text = "" then
        txtLastName.setFocus();
        statusLine.caption := "Please enter a last name";
        return false;
    elseif txtFirstNames.text = "" then
        txtFirstNames.setFocus();
        statusLine.caption := "Please enter first names";
        return false;
    elseif txtAddress.text = "" then
        txtAddress.setFocus();
        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.

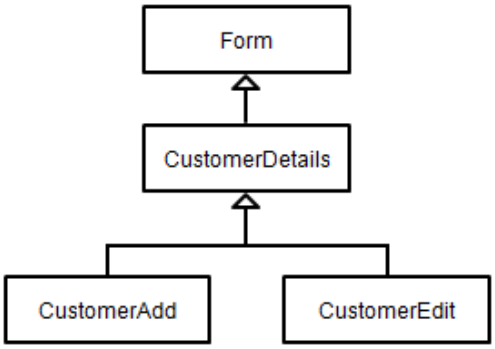
```
createCustomer() protected;

vars
    cust: Customer;
begin
    beginTransaction;
    cust.:= create Customer(txtAddress.text, txtFirstNames.text, txtLastName.text);
    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

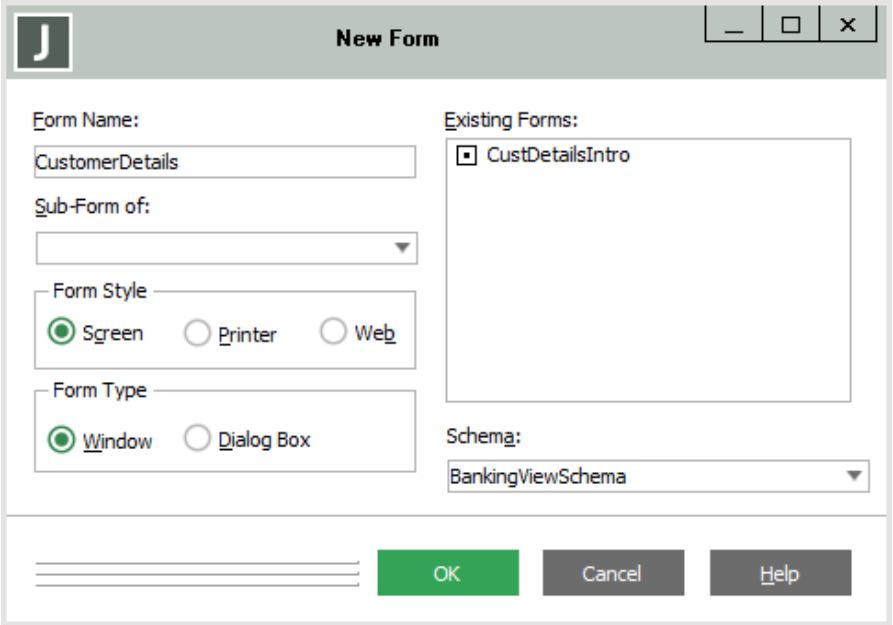
In this exercise, you will create the **BankingViewSchema**, in which you will create forms and applications for the banking system.

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.

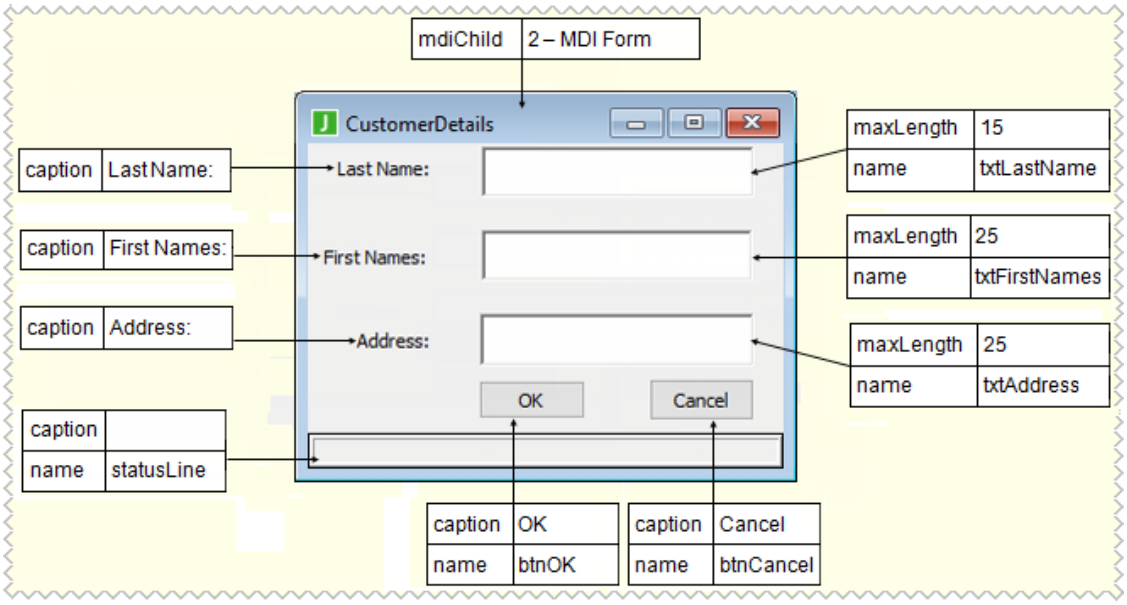
Exercise 11.2 – Adding a CustomerDetails Form

In this exercise, you will create a new form called **CustomerDetails** in the **BankingViewSchema**.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **CustomerDetails** as the name of the form.



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.

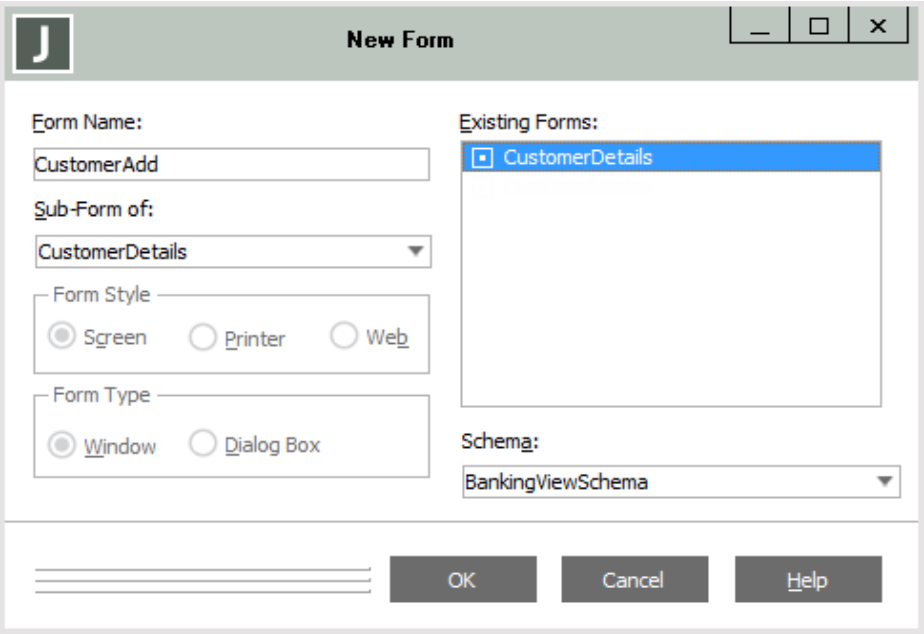
```
runForm();

vars
    form: CustomerDetails;
begin
    app.initialize();
    create form transient;
    form.show();
    // Wait five seconds then close
    app.doWindowEvents(5000);
    form.unloadForm();
end;
```

3. Execute the JadeScript method.

Exercise 11.4 – Adding a CustomerAdd Form

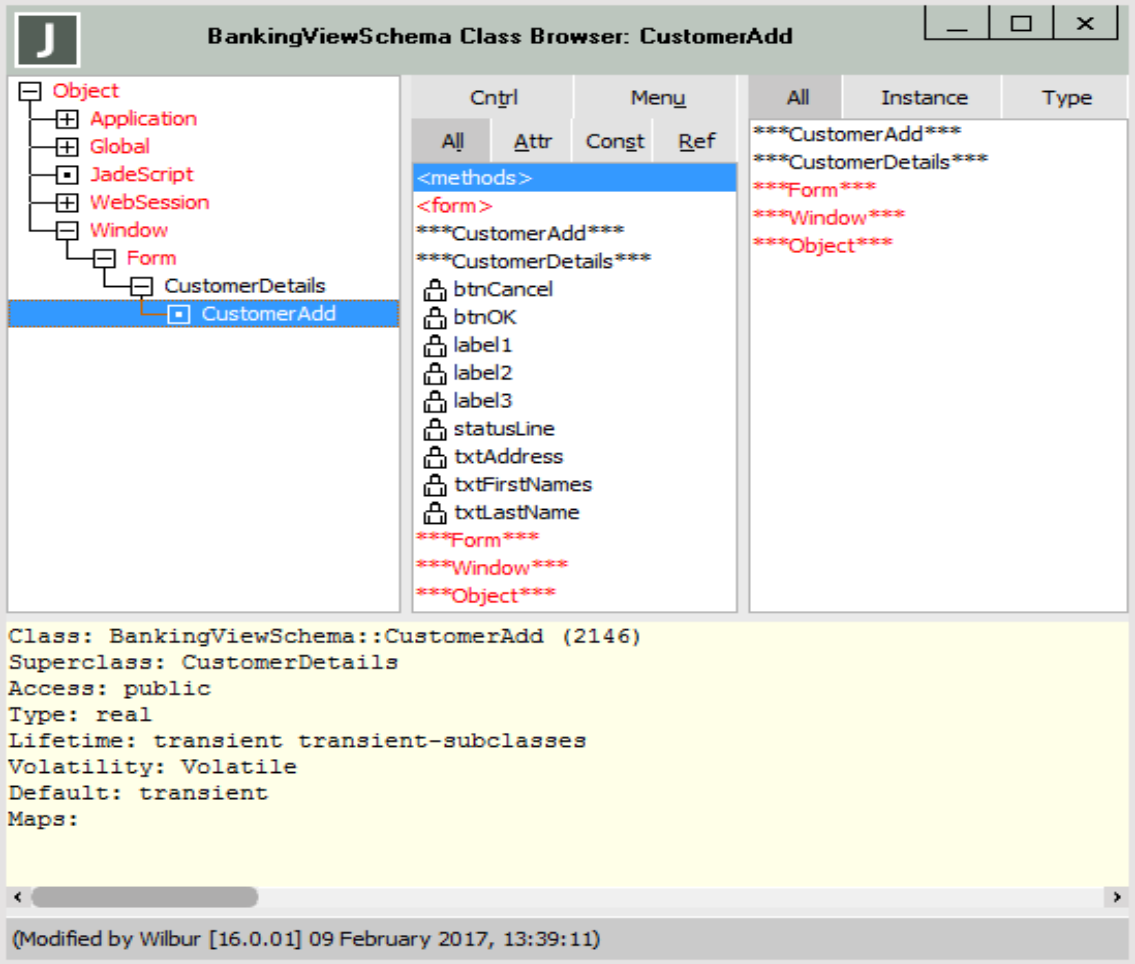
In this exercise, you will create a new subform of **CustomerDetails** called **CustomerAdd**.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **CustomerAdd** as the name of the form and then select **CustomerDetails** from the **Sub-Form of** combo box.



3. Change the form caption property to **Adding a Customer**.
4. Save the form.

5. Return to the Class Browser and then select the View menu **Show Inherited** command, so that inherited controls from **CustomerDetails** are displayed when you view the **CustomerAdd** form.



Exercise 11.5 – Coding the CustomerDetails Form

In this exercise, you will code the following methods in the **CustomerDetails** form that will apply to all subforms.

- An event method to close the form when the **btnCancel** button is clicked
- A protected method called **isDataValid** to check that the user has entered data in all of the text boxes
- A protected method called **clearTextBoxes** to empty text boxes and to position the cursor in the first text box

In subforms (for example, **CustomerAdd**), you will call the protected methods from event methods.

1. In the **CustomerDetails** 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. In the **CustomerDetails** form, select the Methods menu **New Jade Method** command, enter **clearTextBoxes** as the name, select the **Protected** option, and then click the **OK** button.

4. Code the method as follows.

```
clearTextBoxes() protected;

begin
    txtLastName.text := "";
    txtFirstNames.text := "";
    txtAddress.text := "";
    txtLastName.setFocus();
end;
```

5. Add another protected method called **isDataValid**, and code it as follows.

```
isDataValid(): Boolean protected;

begin
    if txtLastName.text = "" then
        txtLastName.setFocus();
        statusLine.caption := "Please enter a last name";
        return false;
    elseif txtFirstNames.text = "" then
        txtFirstNames.setFocus();
        statusLine.caption := "Please enter first names";
        return false;
    elseif txtAddress.text = "" then
        txtAddress.setFocus();
        statusLine.caption := "Please enter an address";
        return false;
    endif;
    return true;
end;
```

Exercise 11.6 – Coding the CustomerAdd Form

In this exercise, you will code the following methods in the **CustomerAdd** form that apply to that form.

- A protected method called **createCustomer**, to create a new customer and to set its properties from the text entered into the text boxes
- An event method, to create a new customer when the **btnOK** button is clicked

To add methods to the **CustomerAdd** form:

1. In the **CustomerAdd** form, select the Methods menu **New Jade Method** command, enter **createCustomer** as the name, select the **Protected** option, and then click the **OK** button.

2. Code the method as follows.

```
createCustomer() protected;

vars
    cust: Customer;
begin
    beginTransaction;
    create cust persistent;
    cust.setPropsOnCreate(txtAddress.text, txtFirstNames.text, txtLastName.text);
    commitTransaction;
end;
```

3. Select the **btnOK** button, and then select the **click** event. Code the method as follows.

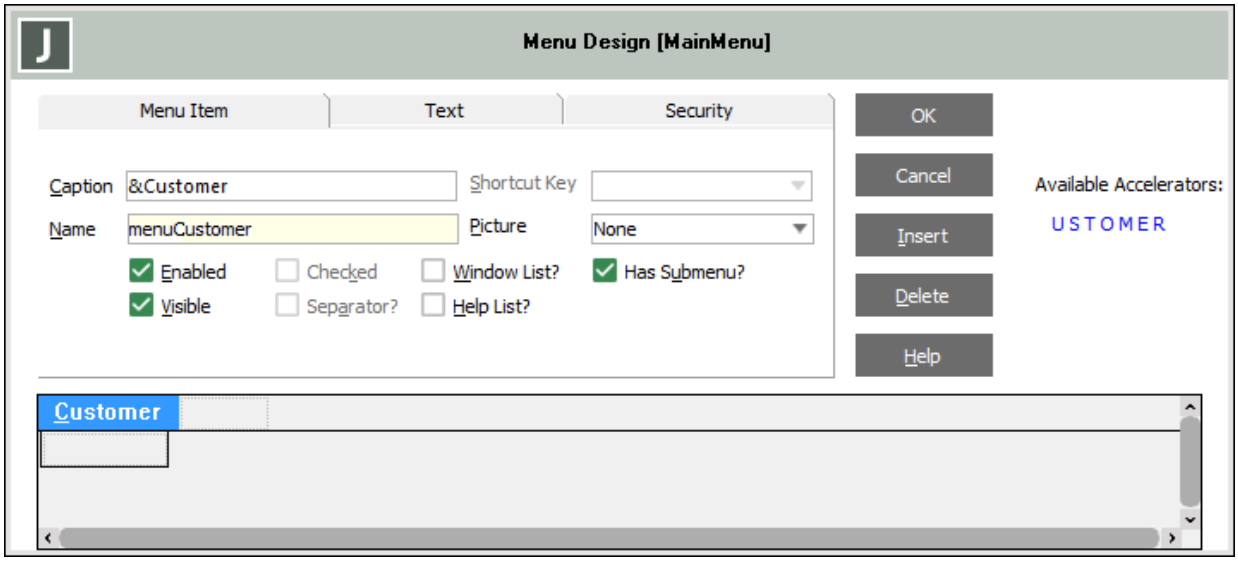
```
btnOK_click(btn: Button input) updating;

begin
  if self.isDataValid() then
      self.createCustomer();
      self.clearTextBoxes();
      statusLine.caption := "Customer successfully added";
  endif;
end;
```

4. Change the JadeScript **runForm** method to open **CustomerAdd** instead of **CustomerDetails**, and comment out the instructions for automatically closing the form.

5. Execute the JadeScript **runForm** method and test that you can add a customer.

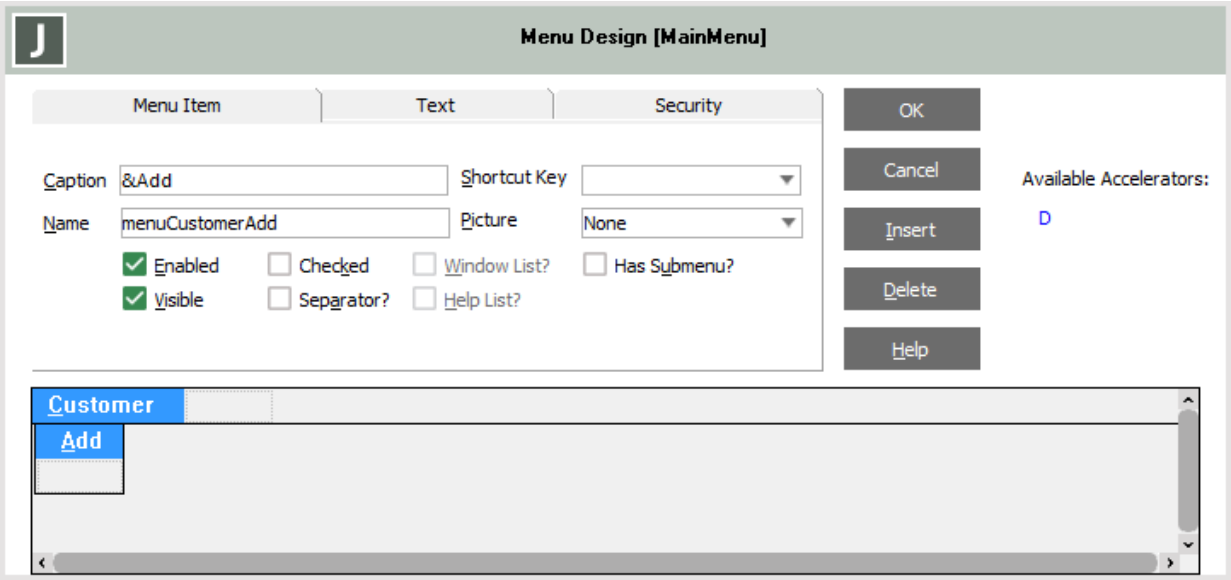
Menus

The menu designer in Painter is accessed by selecting the File menu **Menu Design** command.

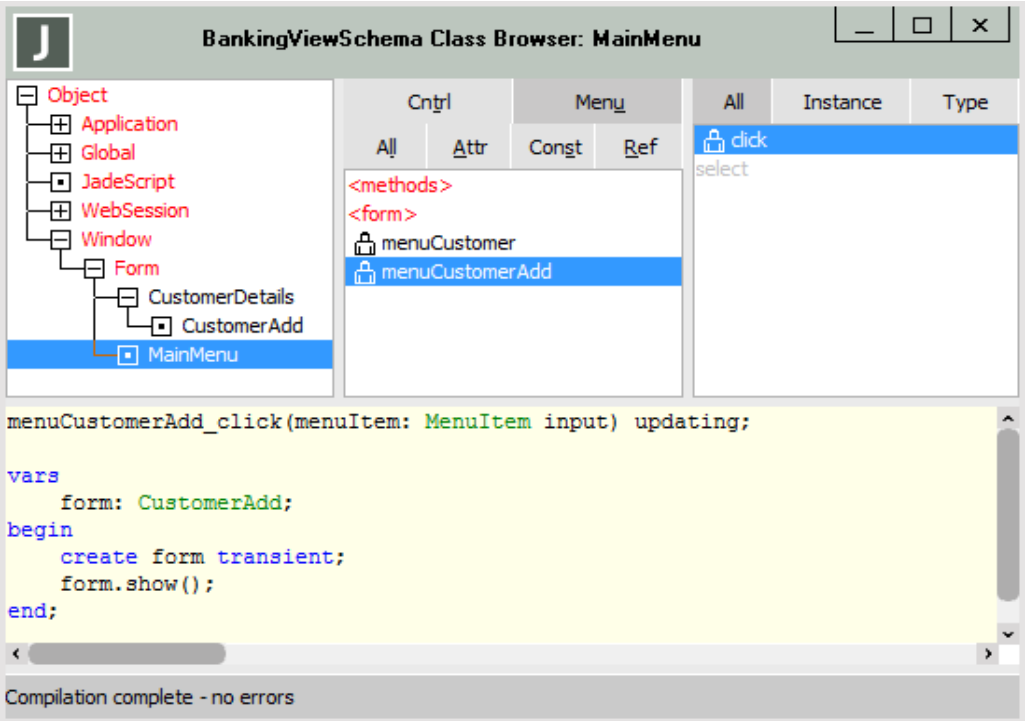


Note An ampersand character (**&**) in the caption causes the character that follows to be underlined. The underlined character becomes an accelerator key when the form is run.

Select a menu item in the designer and then enter values for the **Caption** and **Name**.

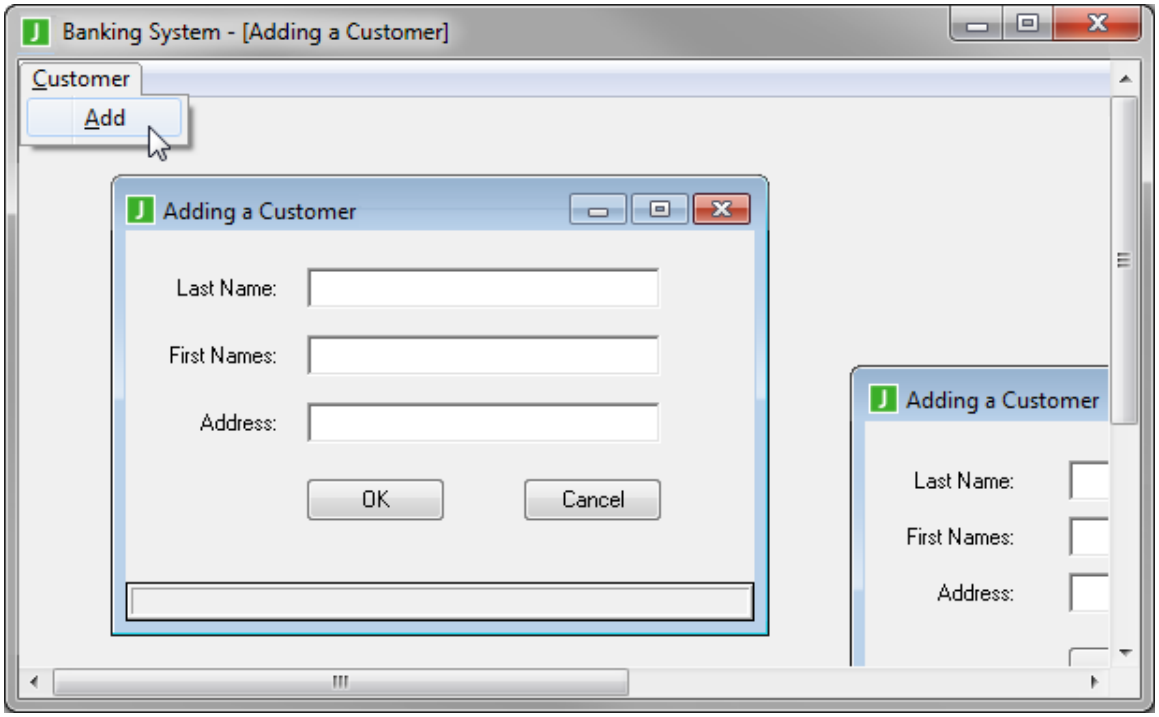


When you save the form and return to the Class Browser, the menu items are displayed in the central Properties List. Select a menu item and then code its **click** event method, as follows.



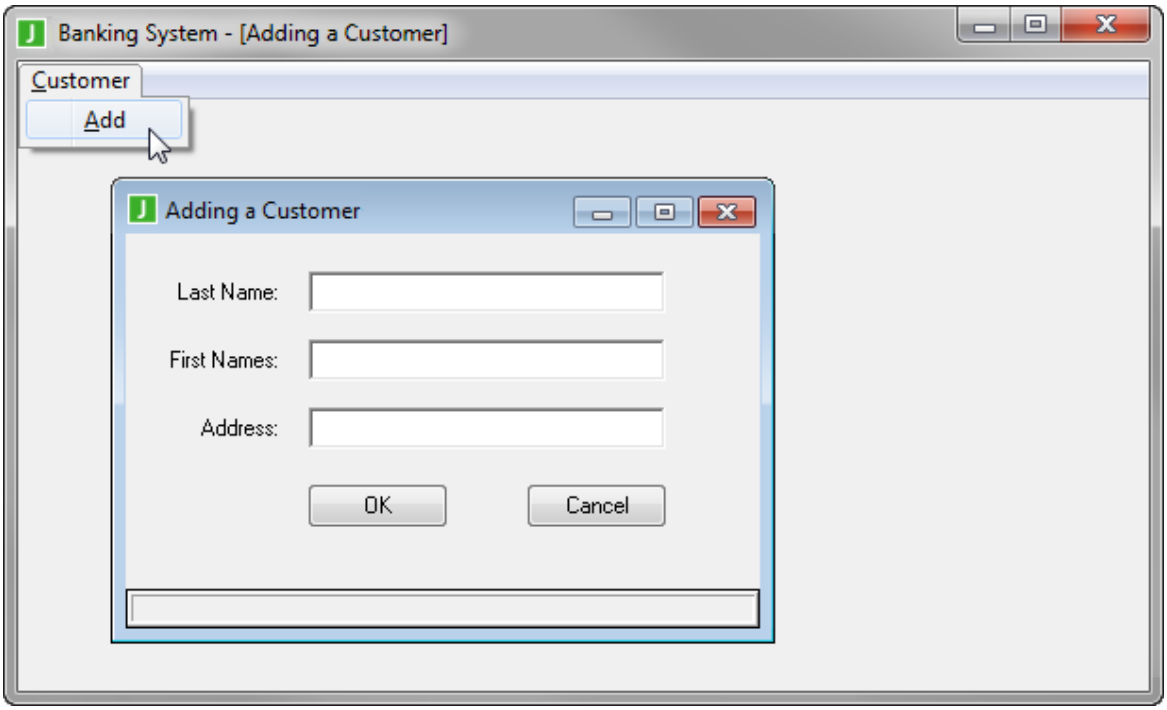
Multiple Document Interface

When you ran the **CustomerAdd** form in the previous exercise, it ran as a *multiple document application* (MDI), as shown in the following diagram.



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 diagram.

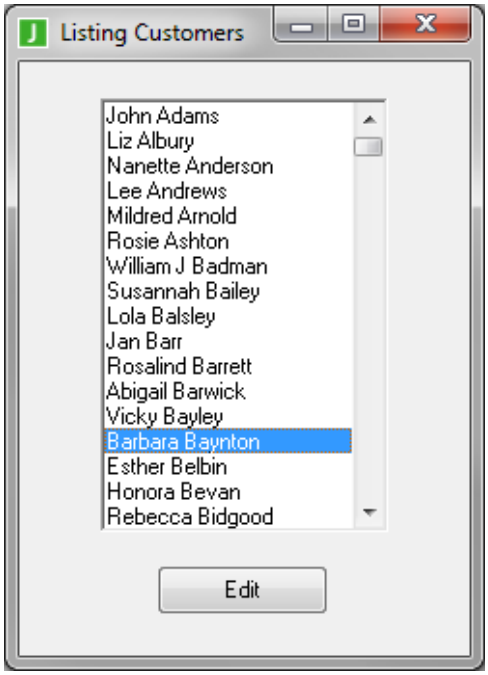


To make a form into an MDI frame, set the **mdiFrame** property to **true** and then add the following instruction when the form is loaded.

```
app.mdiFrame := MainMenu;
```

List Boxes

List boxes are used to display collections of objects in an application; for example, the root object's collection of customers.



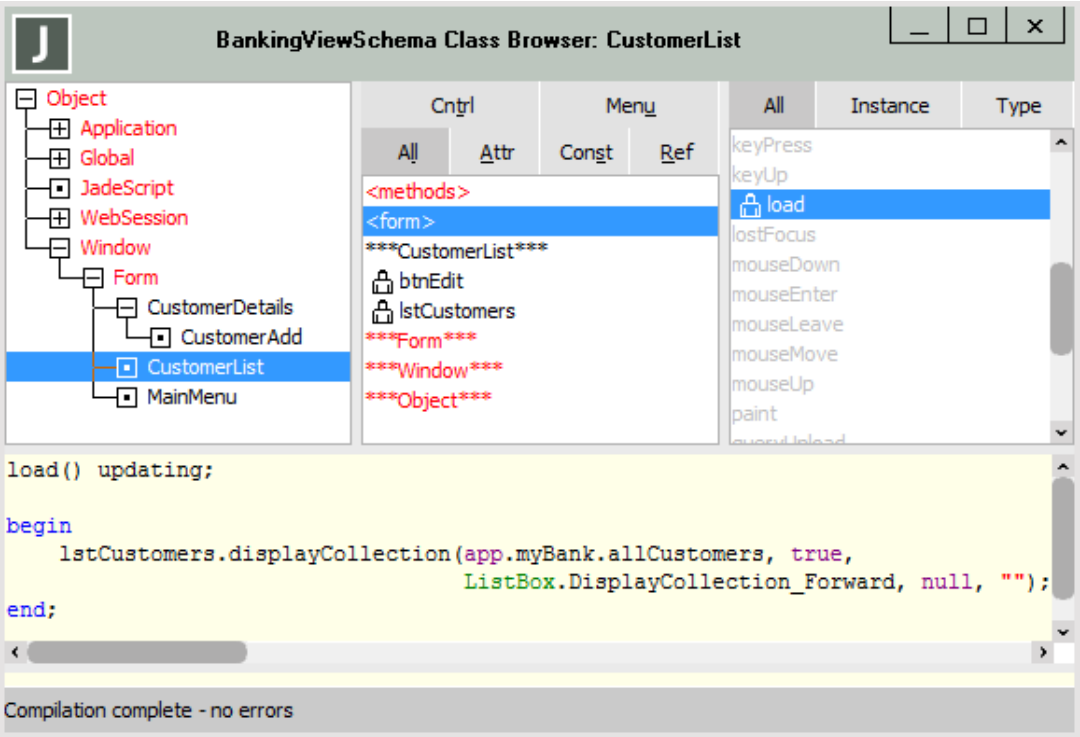
The **ListBox** class provides methods and properties for populating a list box and for determining the customer that the user has selected.

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.

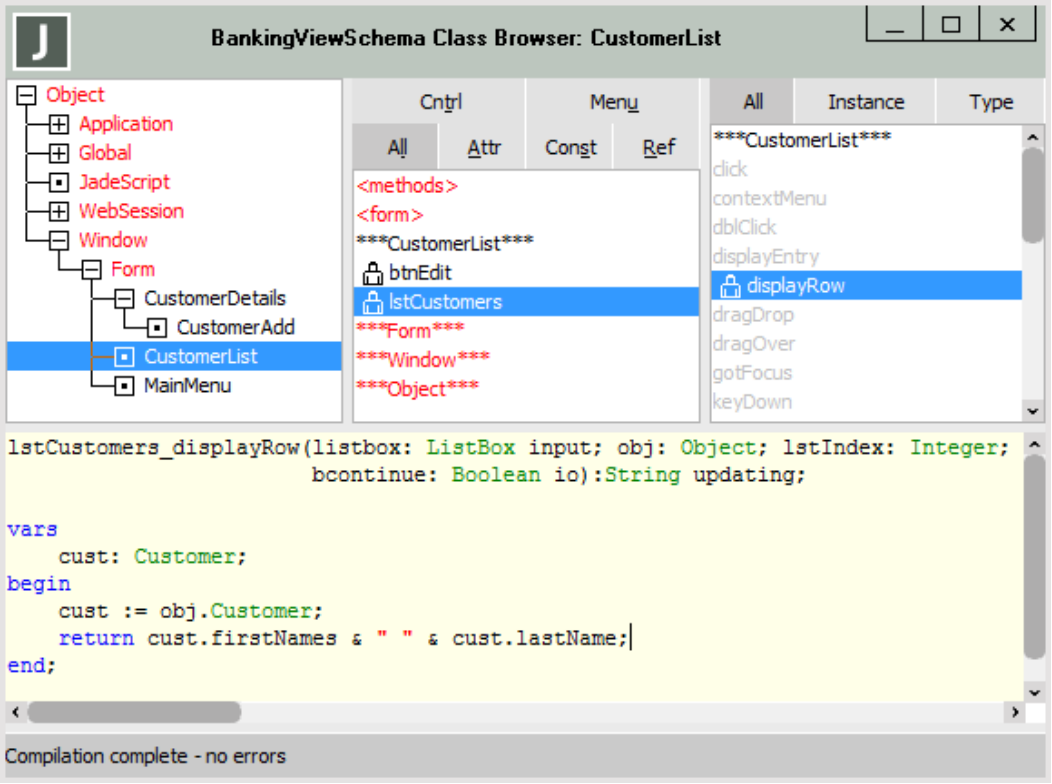
   This is usually done when the form loads.



   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)
   - 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 cust in app.myBank.allCustomers do
    lstCustomers.addItem(cust.firstNames & " " & cust.lastName);
    lstCustomers.itemObject[lstCustomers.listCount] := cust;
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.

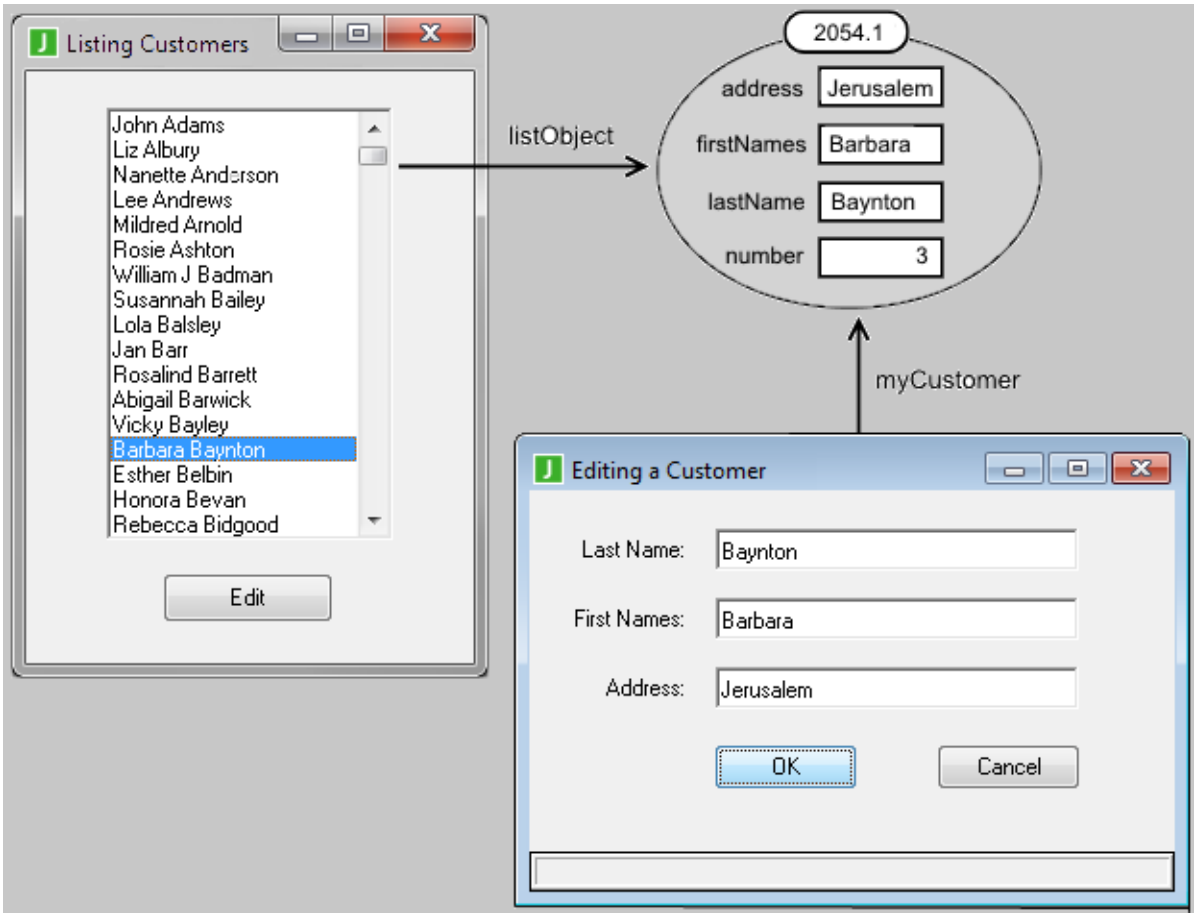
```
cust := lstCustomers.itemObject[lstCustomers.listIndex].Customer;
```

You can achieve the same result by using the **listObject** property, as follows.

```
cust := 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 **CustomerEdit** form is created. The **CustomerEdit** form has a **myCustomer** reference, which identifies the **Customer** object whose details are loaded into the text boxes.



When the customer details are changed, a **setPropsOnUpdate** method will be used instead of the **setPropsOnCreate** method.

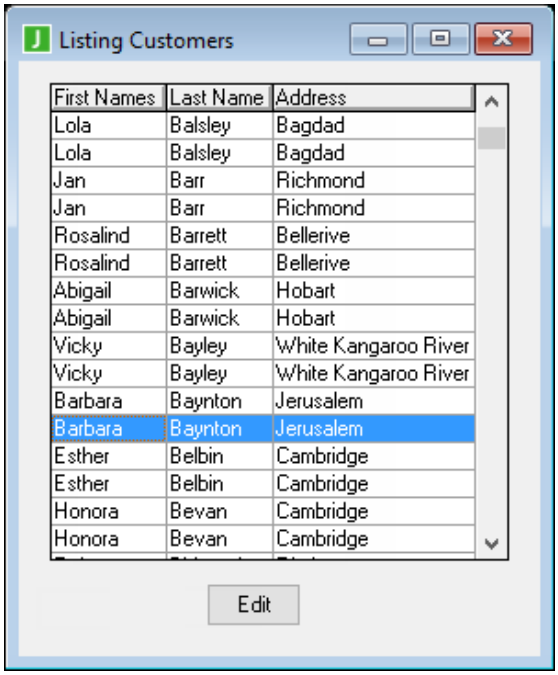
```
setPropsOnUpdate(addr, first, last: String) updating;

begin
    self.address := addr.trimBlanks();
    self.firstNames := first.trimBlanks();
    if not self.lastName = last.trimBlanks() then
        self.lastName := last.trimBlanks();
    endif;
end;
```

The important differences from the **setPropsOnCreate** method are:

- The **lastName** property, which is a dictionary key, is updated only if it has changed. Avoid setting a property that is a dictionary key when the value has not changed, because it avoids the dictionary maintenance that always takes place when a key is set.
- The **myBank** reference is not set because a reference to the root object never changes.

Tables

A table can display objects in a collection, using a number of columns.



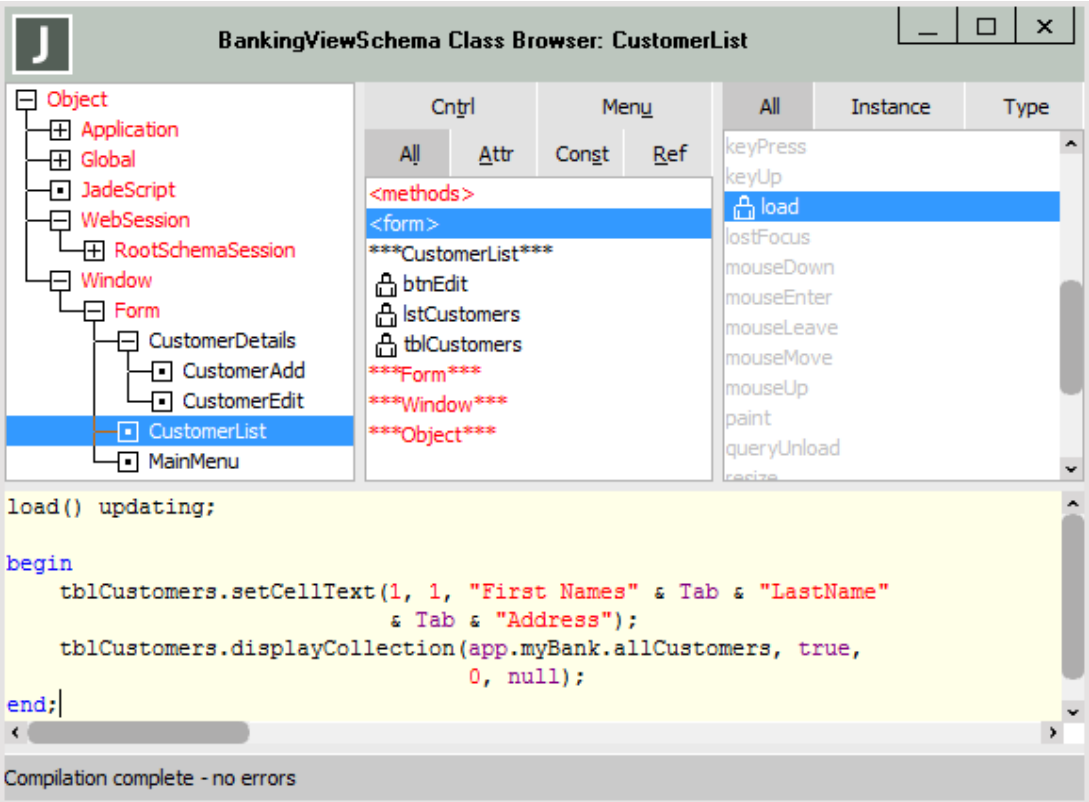
The **Table** class provides similar methods and properties to the **ListBox** class for populating a table and for determining the customer that the user has selected.

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.

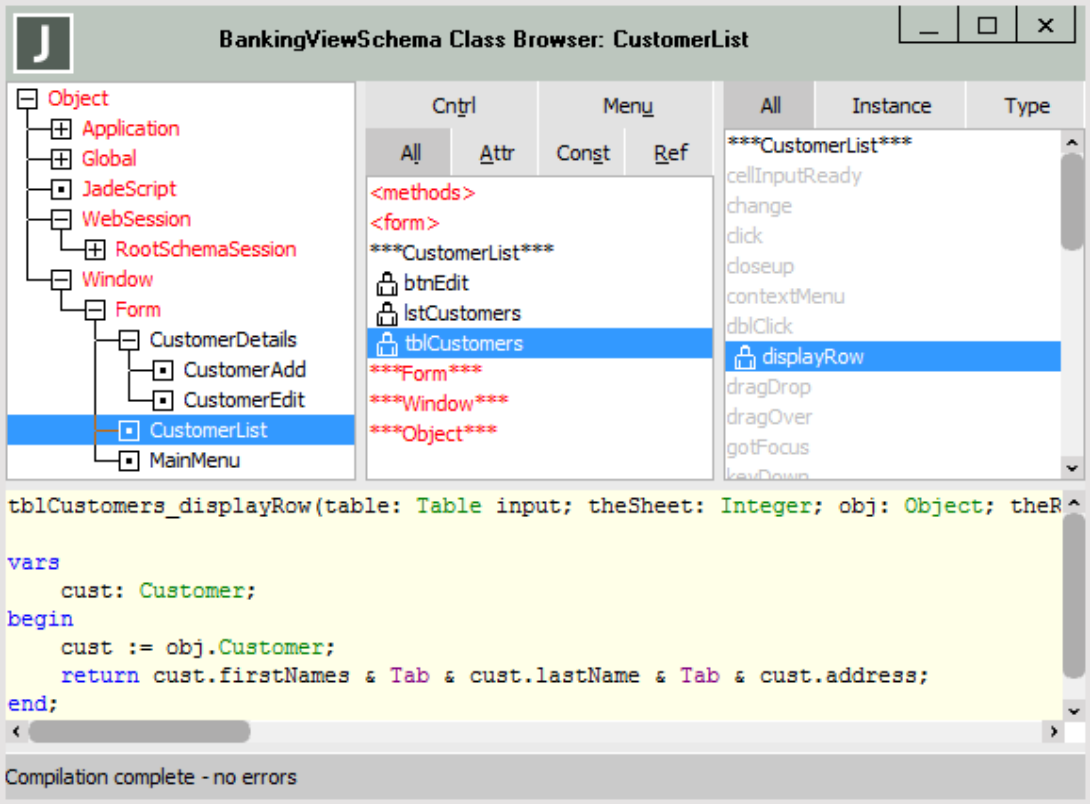
   This is usually done when the form loads.



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)
- 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 cust in app.myBank.allCustomers do
   tblCustomers.addItem(cust.firstNames & Tab &
                        cust.lastName & Tab & cust.address);
   tblCustomers.accessRow(tblCustomers.rows).itemObject := cust;
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.

```
cust := tblCustomers.accessRow(tblCustomers.row).itemObject.Customer;
```

Exercise 11.7 – Adding a MainMenu 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 **MainMenu** as the name of the form.
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 **load** method for the **MainMenu** form, by selecting **<form>** in the central Properties List, and then **load** from the event methods in the Methods List.
8. Code the method as follows.

```
load() updating;

begin
    app.mdiFrame := MainMenu;
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 **menuCustomer** in the **Name** field.
11. Select the first menu item under the **Customer** menu and then enter **&Add** in the **Caption** field and **menuCustomerAdd** in the **Name** field.
12. Click the **OK** button to close the menu designer, and then save the form.
13. In the Class Browser, select the **menuCustomerAdd** menu item and then select the **click** event method.
14. Code the method as follows.

```
menuCustomerAdd_click(menuItem: MenuItem input) updating;

vars
    form: CustomerAdd;

begin
    create form transient;
    form.show();
end;
```

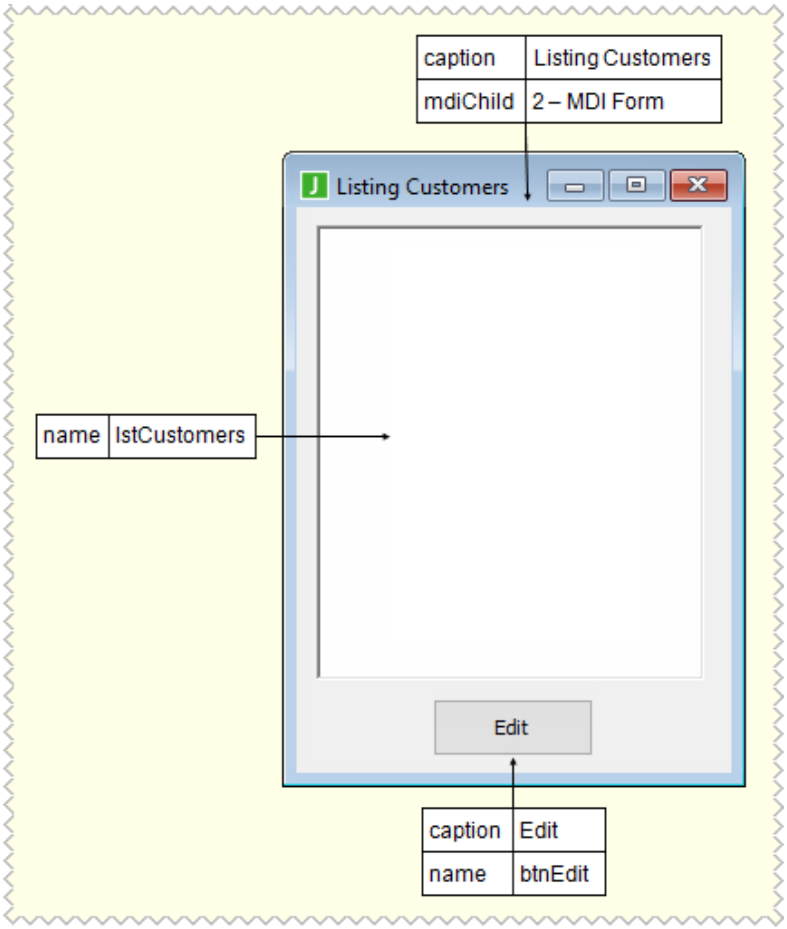
15. Change the JadeScript **runForm** method to open **MainMenu** instead of **CustomerAdd**.
16. Execute the JadeScript **runForm** method and test the MDI parent-child functionality.

Exercise 11.8 – Adding a CustomerList Form

In this exercise, you will add a **CustomerList** form that will display the root object's collection of customers. You will then add an option to the **Customer** menu on the **MainMenu** form to open the **CustomerList** form.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **CustomerList** as the name of the form.

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 **load** method for the **CustomerList** form by selecting **<form>** in the central Properties List, and then **load** from the event methods in the Methods List.

6. Code the method as follows.

```
load() updating;

begin
    lstCustomers.displayCollection(app.myBank.allCustomers, true, 0, null, "");
end;
```

7. Select the **lstCustomers** list box, and then select the **displayRow** event.

8. Code the **displayRow** method as follows.

```
lstCustomers_displayRow(listbox: ListBox input; obj: Object;
                        lstIndex: Integer; bcontinue: Boolean io):String updating;
vars
    cust: Customer;
begin
    cust := obj.Customer;
    return cust.firstNames & " " & cust.lastName;
end;
```

9. Select the **btnEdit** button, and then select the **click** event.

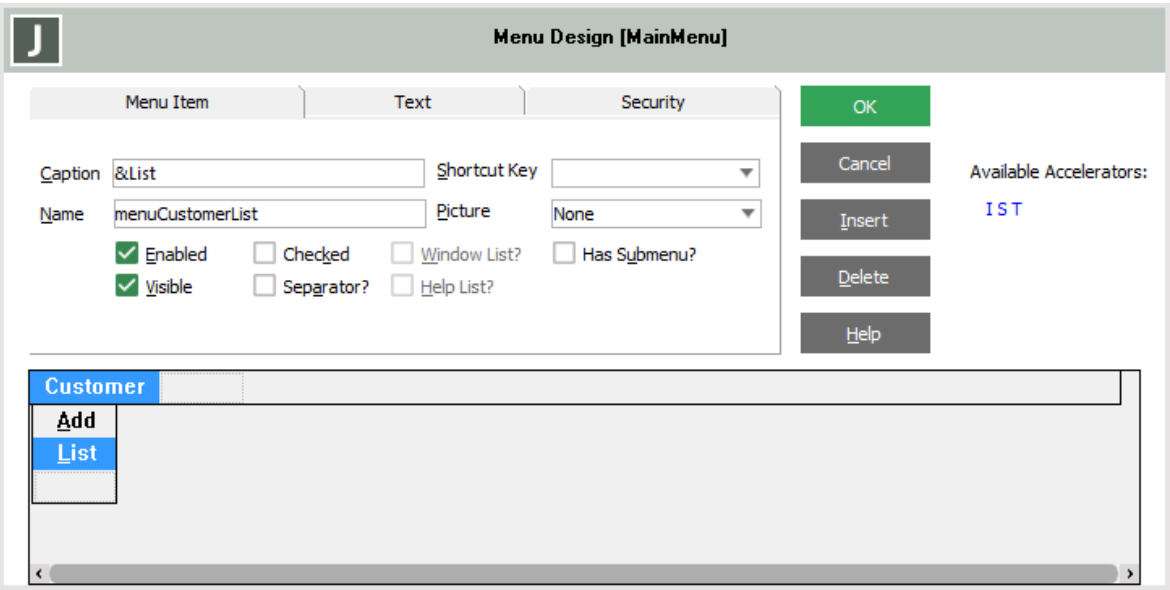
10. Code the **click** event method to write the last name of the selected customer. (You will change this method in a later exercise.)

```
btnEdit_click(btn: Button input) updating;

vars
    cust: Customer;
begin
    cust := lstCustomers.listObject.Customer;
    if cust = null then
        app.msgBox("Select a customer first", "Error", MsgBox_OK_Only);
    else
        write cust.lastName;
    endif;
end;
```

11. Open the **MainMenu** form in Painter.

12. Open the menu designer by selecting the File menu **Menu Design** command.

13. Select the cell below the **Add** menu, and then enter **&List** in the **Caption** field and **menuCustomerList** in the **Name** field.



14. Click the **OK** button to close the menu designer, and then save the form.

15. In the Class Browser, select the **menuCustomerList** menu item and then select the **click** event method.

16. Code the method as follows.

```
menuCustomerList_click(menuItem: MenuItem input) updating;

vars
    form: CustomerList;
begin
    create form transient;
    form.show();
end;
```

17. Execute the **runForm** JadeScript method and open the **CustomerList** form.

    Test that the **btnEdit** button writes the correct message.

Exercise 11.9 – Adding a setPropsOnUpdate Method

In this exercise, you will return to the **Customer** class in the **BankingModelSchema** and add a **setPropsOnUpdate** method.

1. Select **BankingModelSchema** in the Schema Browser.
2. Open a Class Browser and then select the **Customer** class.
3. Select the Methods menu **New Jade Method** command, enter **setPropsOnUpdate** as the name, and then click the **OK** button.
4. Code the method as follows .

```
setPropsOnUpdate(addr, first, last: String) updating;

begin
    self.address := addr.trimBlanks();
    self.firstNames := first.trimBlanks();
    if not self.lastName = last.trimBlanks() then
        self.lastName := last.trimBlanks();
    endif;
end;
```

Exercise 11.10 – Adding a CustomerEdit Form

In this exercise, you will create a new subform of **CustomerDetails** called **CustomerEdit**.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **CustomerEdit** as the name of the form and then select **CustomerDetails** from the **Sub-Form** combo box.
3. Change the form caption property to **Editing a Customer**.
4. Save the form.
5. Return to the Class Browser and then select the View menu **Show Inherited** command, so that inherited controls from **CustomerDetails** are displayed when you view the **CustomerEdit** form.
6. In the **CustomerEdit** form, add a public reference called **myCustomer** of type **Customer**.

This reference will be set by the user selecting a customer in the **CustomerList** form and then clicking the **Edit** button.

7. Select the **load** method for the **CustomerEdit** form, by selecting **<form>** in the central Properties List and then **load** from the event methods in the Methods List.

8. Code the method to load information for the **myCustomer** object into the text boxes, as follows.

```
load() updating;

begin
    txtAddress.text := myCustomer.address;
    txtFirstNames.text := myCustomer.firstNames;
    txtLastName.text := myCustomer.lastName;
end;
```

9. In the **CustomerEdit** form, add a protected method called **editCustomer** and code it as follows.

```
editCustomer() protected;

begin
    beginTransaction;
    myCustomer.setPropsOnUpdate(txtAddress.text,
                                txtFirstNames.text,
                                txtLastName.text);
    commitTransaction;
end;
```

10. Select the **btnOK** button, and then select the **click** event. Code the method as follows.

```
btnOK_click(btn: Button input) updating;

begin
    if self.isDataValid() then
        self.editCustomer();
        self.unloadForm();
    endif;
end;
```

11. Finally, in the **CustomerList** form, change the **click** method of the **Edit** button to open **CustomerEdit** form and set the **myCustomer** reference, as follows.

```
btnEdit_click(btn: Button input) updating;

vars
    cust: Customer;
    form: CustomerEdit;
begin
    cust := lstCustomers.listObject.Customer;
    if cust = null then
        app.msgBox("Select a customer", "Error", MsgBox_OK_Only);
    else
        // write cust.lastName;
        create form transient;
        form.myCustomer := cust;
        form.show();
    endif;
end;
```

12. Execute the JadeScript **runForm** method and then open the **CustomerList** form.
13. Select the customer **Barbara Baynton** and change the name to **Barbara Jackson**, by clicking the **Edit** button.

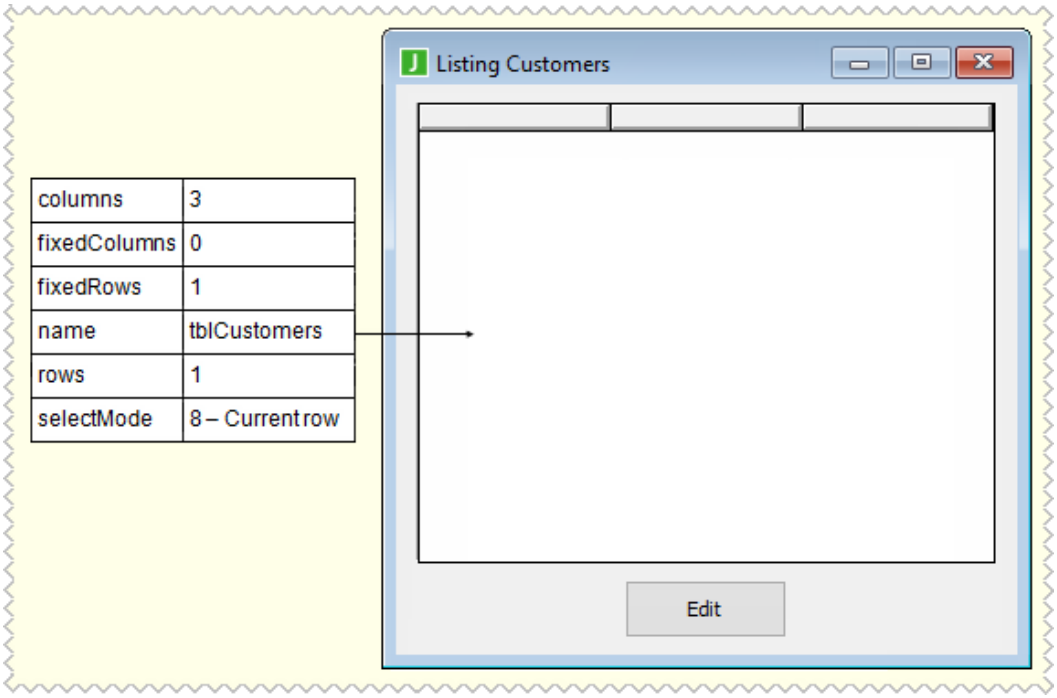
    Does the list box on the **CustomerList** form update? Why?
14. On the **CustomerList** form, select the customer **Barbara Jackson** and change the name to **Alice Jackson**, by clicking the **Edit** button.

    Does the list box on the **CustomerList** form update? Why?

Exercise 11.11 – Changing the CustomerList Form

In this exercise, you will change the **CustomerList** form to use a table instead of a list box.

1. Open the JADE Painter.
2. Select the File menu **Edit Form** command, select **CustomerList**, 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 **load** method for the **CustomerList** form.

6. Replace the code, as follows.

```
load() updating;

begin
   // lstCustomers.displayCollection(app.myBank.allCustomers, true, 0, null, "");
   tblCustomers.setCellText(1,1,"First Names"& Tab &"Last Name"& Tab &"Address");
   tblCustomers.displayCollection(app.myBank.allCustomers, true, 0, null);
end;
```

7. Select the **tblCustomers** table, and then select the **displayRow** event.

8. Code the **displayRow** method as follows.

```
tblCustomers_displayRow(table: Table input; theSheet: Integer; obj: Object;
                        theRow: Integer; bcontinue: Boolean io): String updating;
vars
    cust: Customer;
begin
    cust := obj.Customer;
    return cust.firstNames & Tab & cust.lastName & Tab & cust.address;
end;
```

9. Select the **btnEdit** button, and then select the **click** event.

10. Replace the code in the **click** method, as follows.

```
btnEdit_click(btn: Button input) updating;

vars
    cust: Customer;
    form: CustomerEdit;
begin
    // cust := lstCustomers.listObject.Customer;
    cust := tblCustomers.accessRow(tblCustomers.row).itemObject.Customer;
    if cust = null then
        app.msgBox("Select a customer", "Error", MsgBox_OK_Only);
    else
        // write cust.lastName;
        create form transient;
        form.myCustomer := cust;
        form.show();
    endif;
end;
```

11. Test that the **CustomerList** form works correctly.

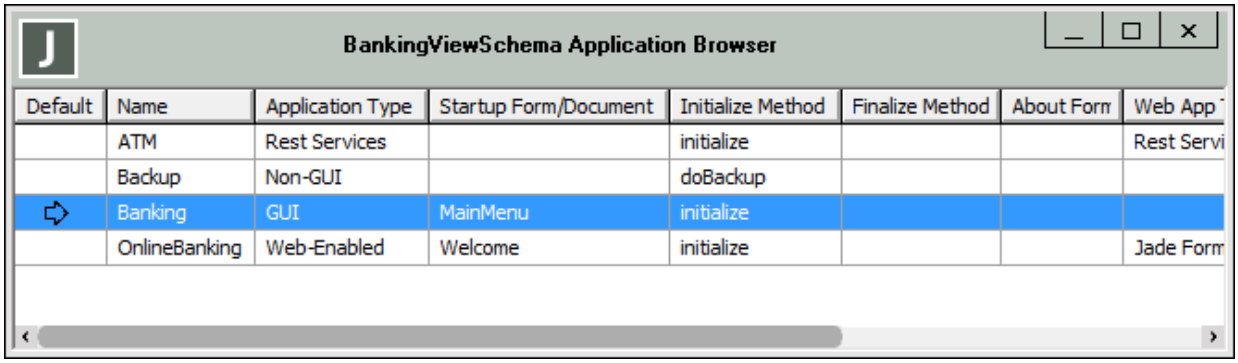
Module 12 Applications

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 development environment toolbar.

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 diagram.



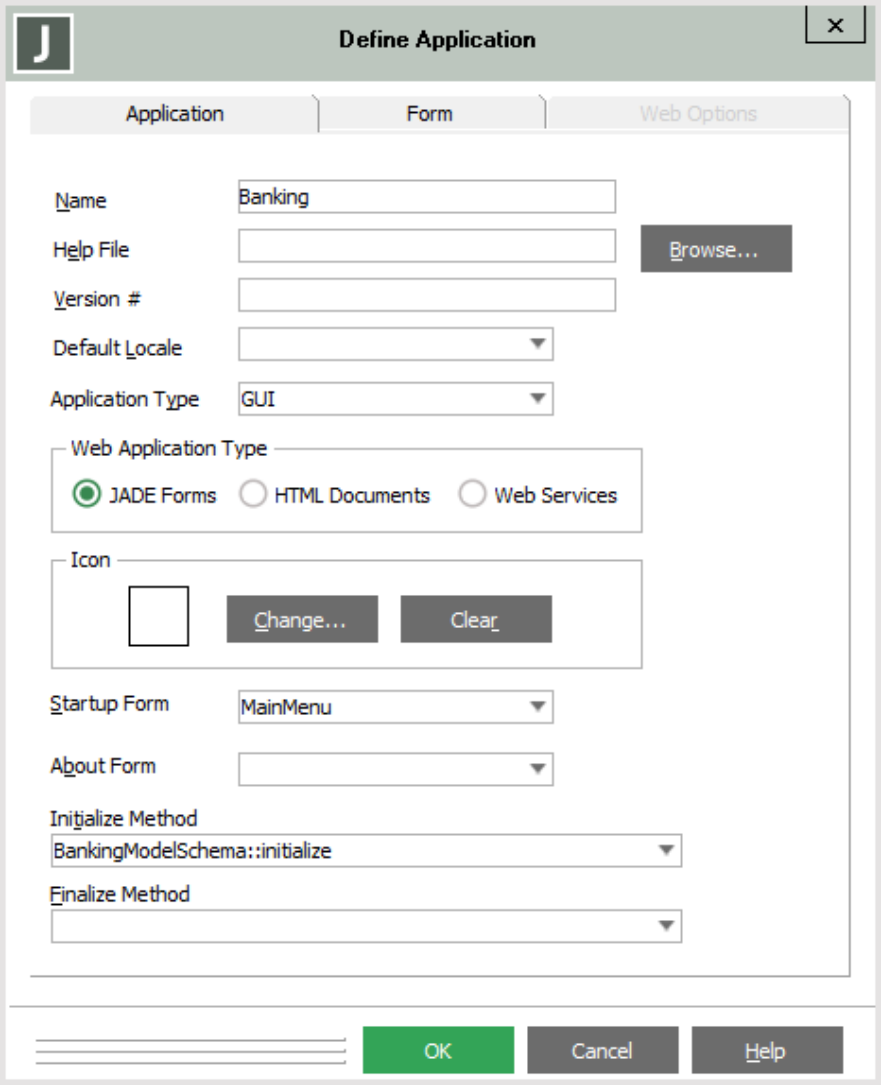
Default	Name	Application Type	Startup Form/Document	Initialize Method	Finalize Method	About Form	Web App
	ATM	Rest Services		initialize			Rest Servi
	Backup	Non-GUI		doBackup			
	Banking	GUI	MainMenu	initialize			
	OnlineBanking	Web-Enabled	Welcome	initialize			Jade Form

You can select an application in the Application Browser and set it as the *default* application, by using the Application menu **Set** command.

→ You can start the default application by right-clicking the arrow button (Run Application) in the JADE development environment 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 diagram.



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 JADE, 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** is the form in the current schema (or a superschema) that is displayed when the application starts.

The **Initialize Method** is executed when the application starts before the startup form is displayed. The **Finalize Method** 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

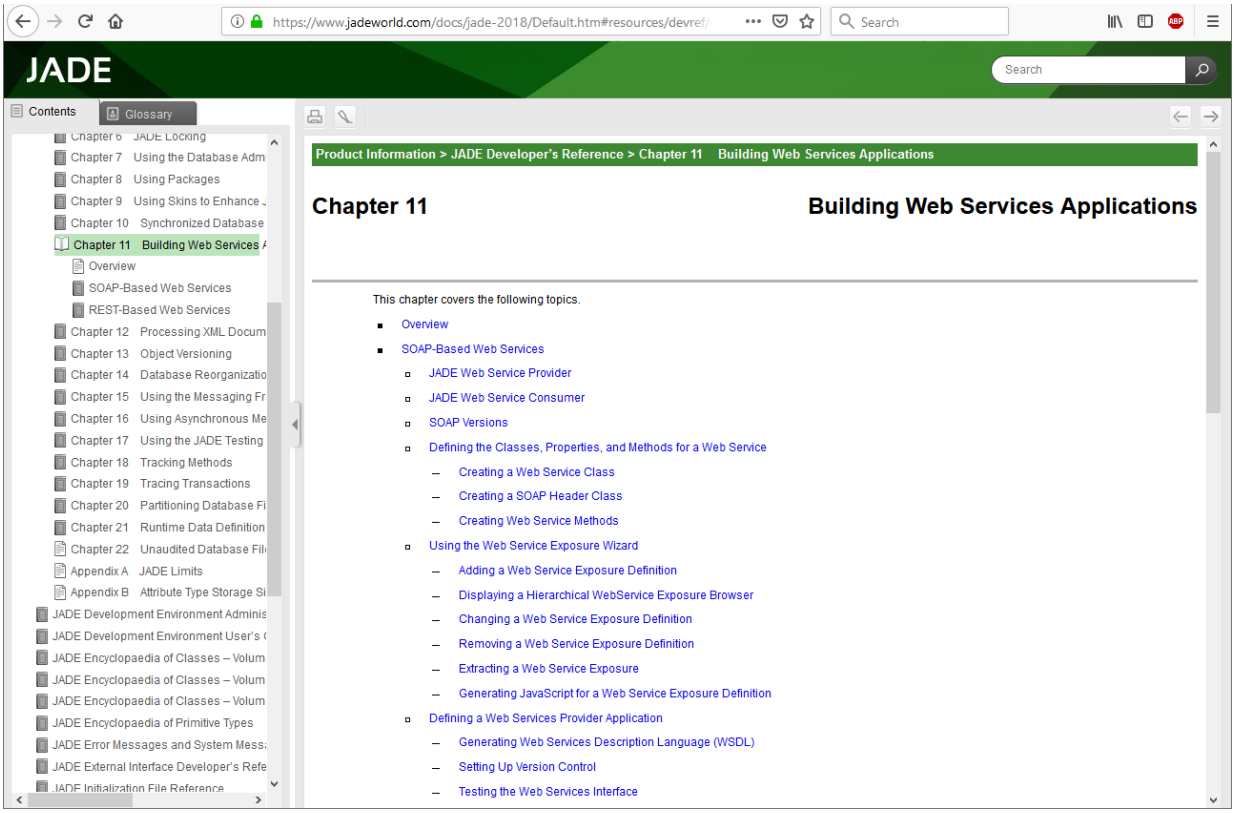
A web service usually uses HTTP to exchange data. Unlike a web application, which is typically HTML over HTTP, a web service is Extensible Markup Language (XML) over HTTP. A client sends a request in XML, and the server responds with an XML response. This XML can be Plain Old XML (POX), which is typically 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 **Json** 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 product information library provides you with resources that enable you to develop web service and REST service applications.

The following diagram shows the JADE 2018 HTML5 contents pane in a browser with the "Building Web Services Applications" chapter of the *JADE 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 three web services white papers, and the REST services white paper in the JADE product information library, see:

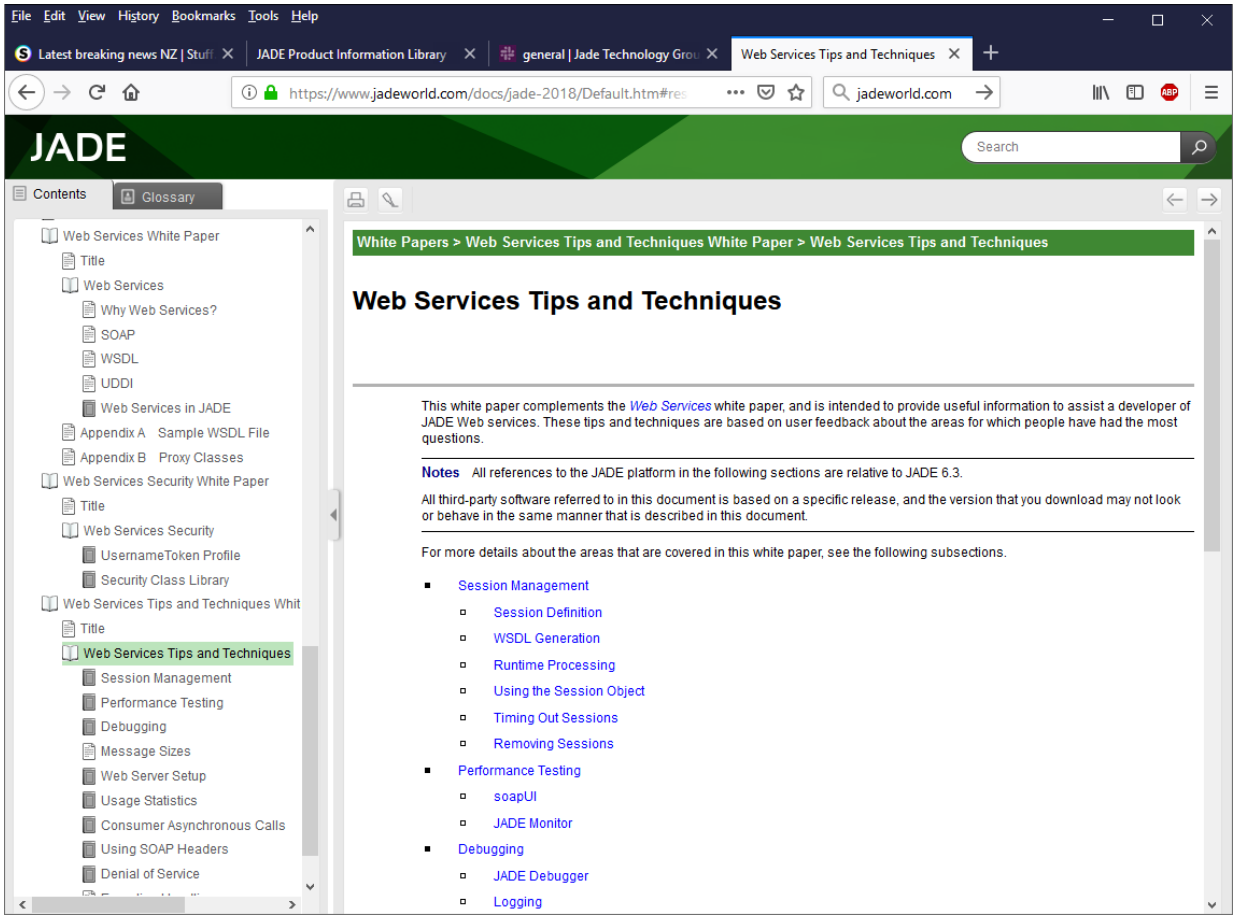
https://www.jadeworld.com/docs/jade-2018/Default.htm

In addition, you can download the:

- PDF (print) format of the *JADE Developer's Reference* from the **Development Environment** section of the **JADE 2018 Documents** at https://www.jadeworld.com/developer-center/resource-library
- Three web services and the REST services white papers in print (PDF) format from **White Papers** in the

Resource Library section of **Developer-Center** at https://www.jadeworld.com/developer-center/resource-library/white-papers

The following diagram shows the JADE 2018 HTML5 contents pane in a browser with the three web services-related white papers expanded in the **Contents** pane at the left.

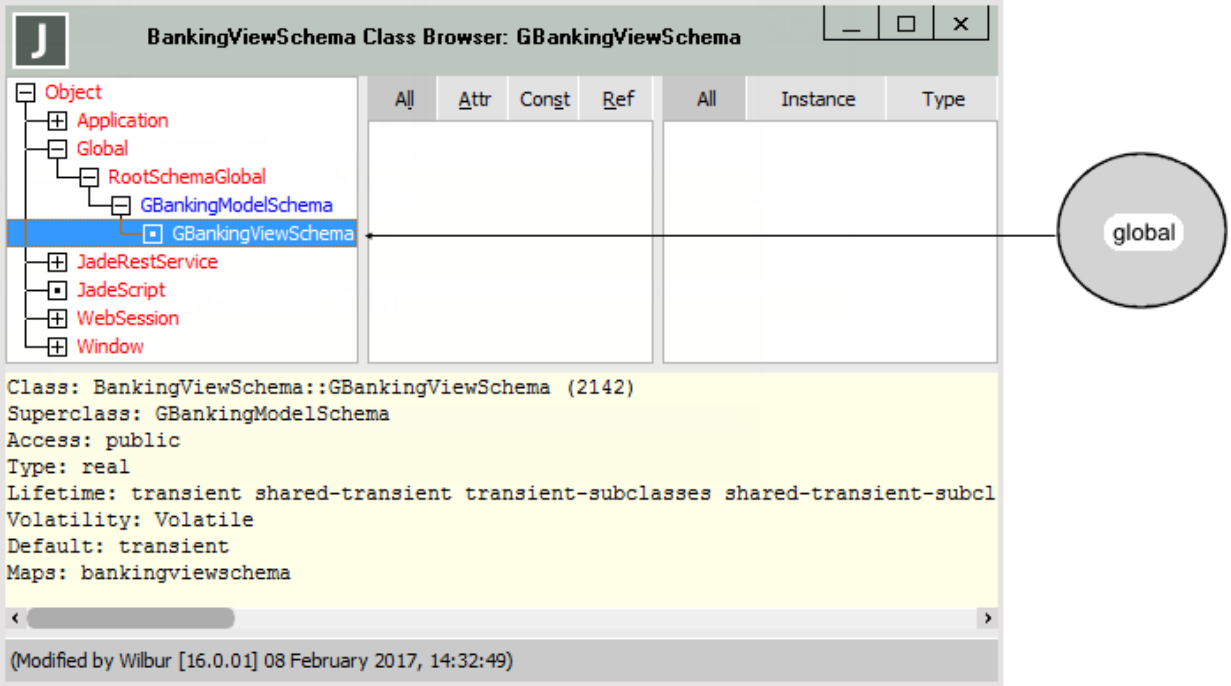


Tip As the HTML5 format of the JADE 2018 product information library contains not only the product information but the white papers and the *Erewhon Demonstration System Reference*, you can search the complete product information library. See the "Search and Print Tips for HTML5 Help" topic in the **Contents** 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**.

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;
```

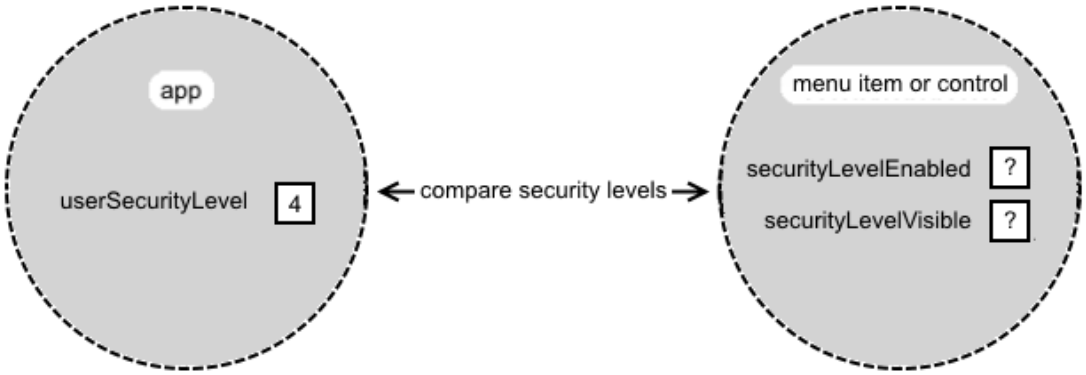
Application Security

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.

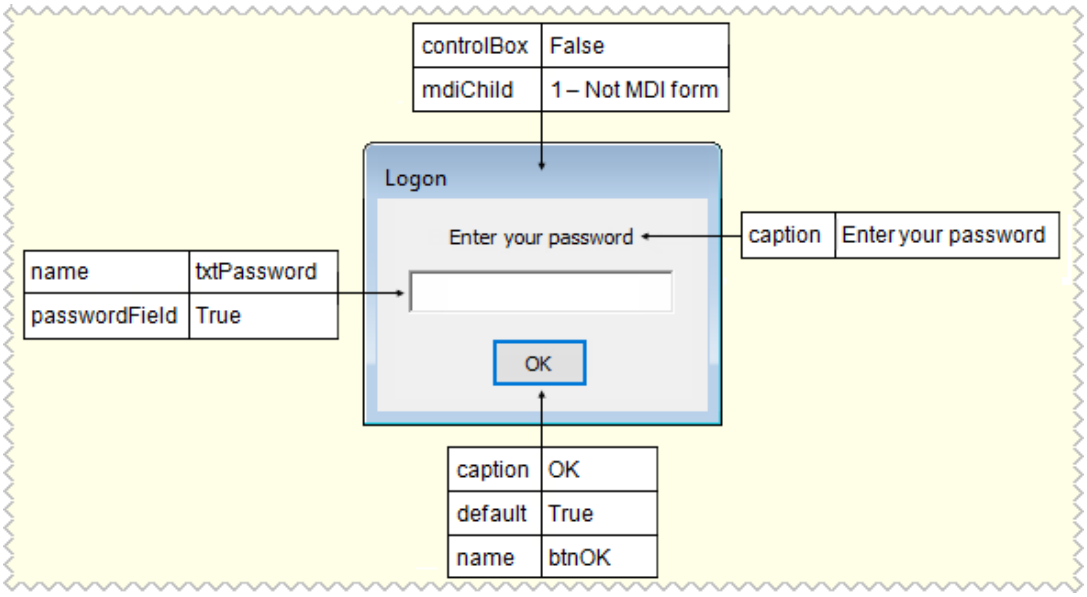
1. Open the Application Browser and then select the **BankingViewSchema** application.
2. Select the Application menu **Change** command.
3. Change the name of the application to **Banking**.

4. Select **MainMenu** 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 development environment toolbar.

Exercise 12.2 – Adding a Logon Form

In this exercise, you will create a new form called **Logon**.

1. Open the Painter.
2. Select the File menu **New Form** command. Enter **Logon** as the name of the form.
3. Paint the form as shown in the following diagram.



4. Save the form and then return to the Class Browser.
5. In the **Logon** 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 getAndValidateUser

In this exercise, you will reimplement the **getAndValidateUser** method to test whether the correct password, which is **secret**, is entered on the **Logon** form.

1. Select the **GBankingModelSchema** 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
    form: Logon;
begin
    // Skip authentication if application not Windows desktop-type
    if not app.applicationType = Application.ApplicationType_GUI then
        return true;
    endif;
    create form transient;
    form.showModal();
    if form.txtPassword.text.toLower() = "secret" then
        return true;
    else
        app.msgBox("Incorrect password", "Logon Error", MsgBox_OK_Only);
        return false;
    endif;
end;
```

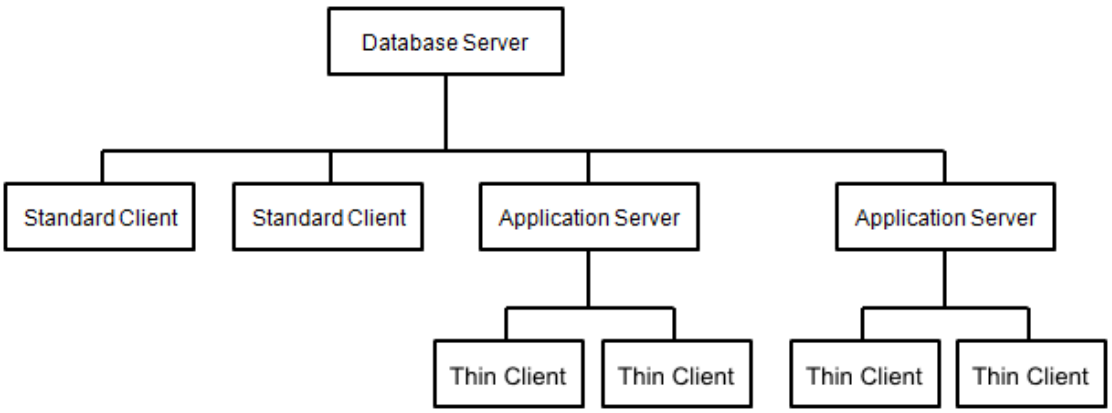
4. Run the **Banking** application and test the logon authentication.

Challenge

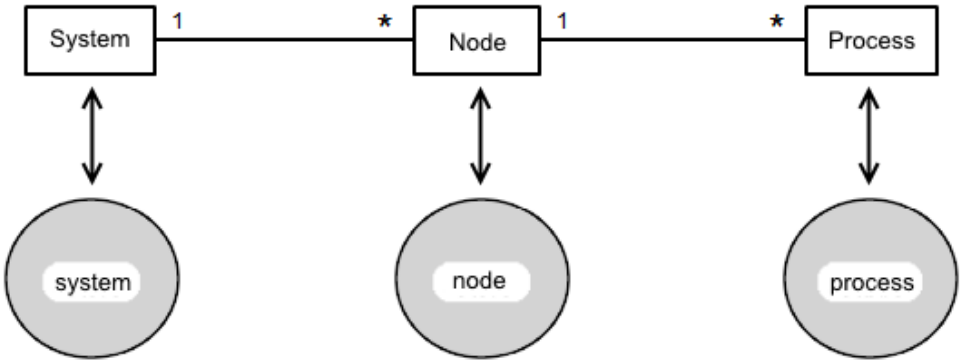
Change the code to give the user three chances to enter the password correctly.

Environmental Objects

The architecture of a JADE 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.

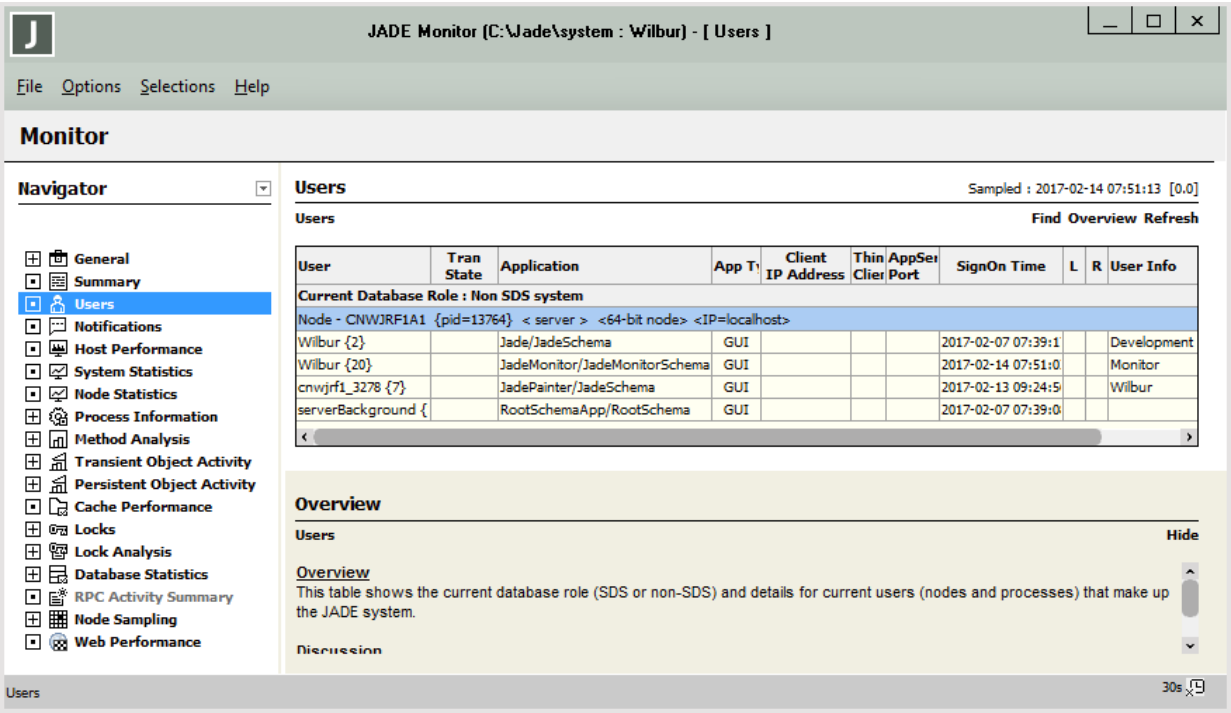
You can use persistent objects or shared transient objects to share information between the applications.

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.

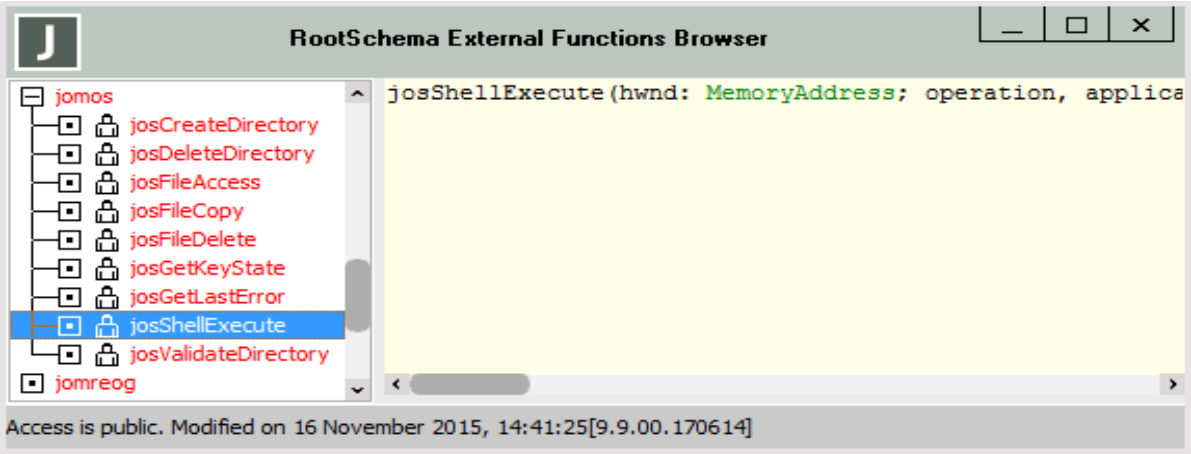
Setting the **modal** parameter to **true** suspends the JADE application until the external application terminates.

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);
```

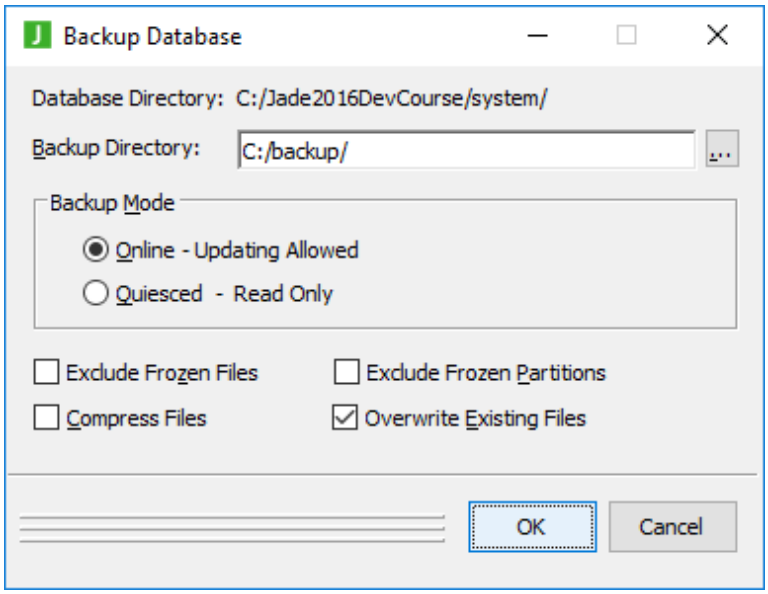
```
// Open default e-mail client
call josShellExecute(null, "open", "mailto:wilbur@jadeworld.com?" &
 "subject=Hello World&body=A traditional greeting.", "", "", 0);
```

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.



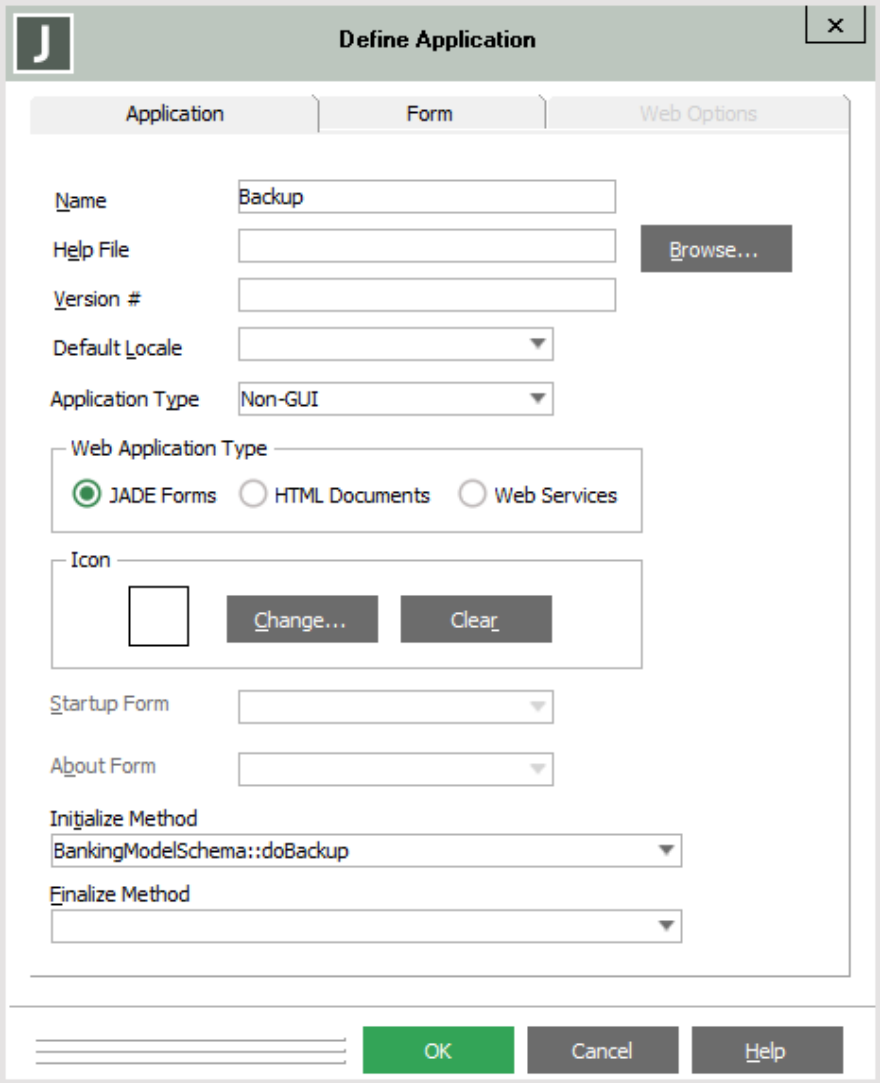
The **JadeDatabaseAdmin** class provides backup and database-related operations. The **backupAllDbFiles** method requires the same kind of information as the **JadeBackupDatabaseDialog** form but it enables the backup to be carried out as a non-GUI operation.

```
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 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

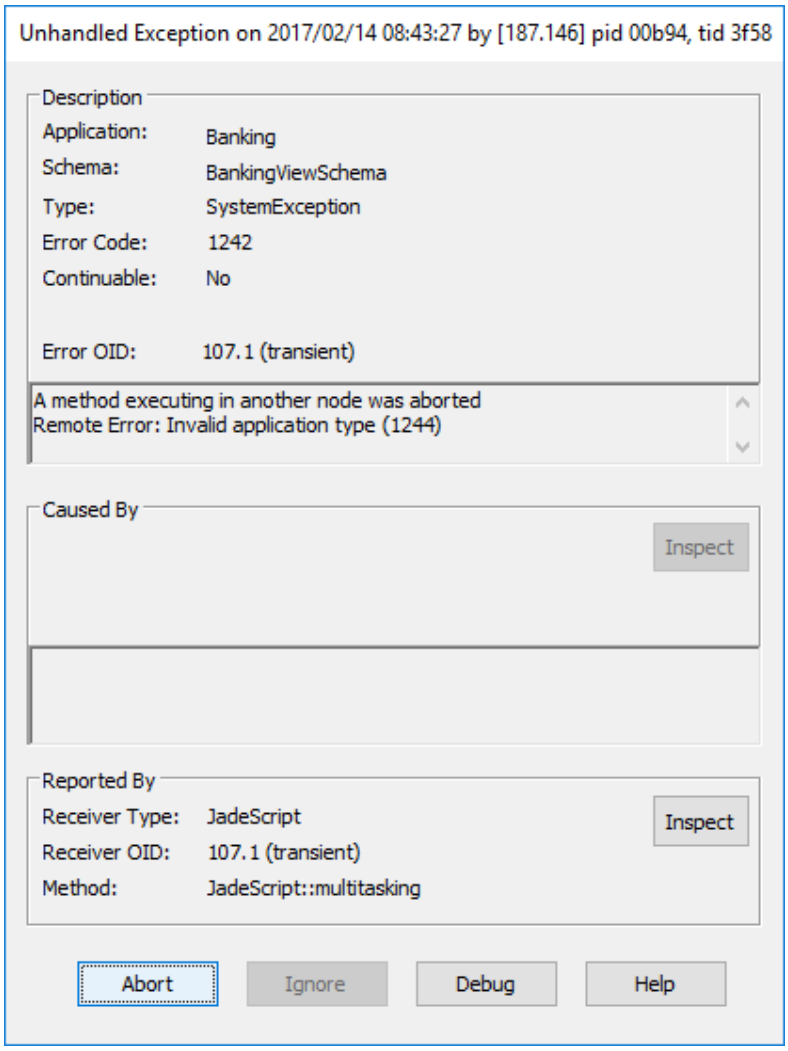
In this exercise, you will write a JadeScript method that uses the **startApplication** method to run a number of applications in parallel.

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();

vars
    dba: JadeDatabaseAdmin;
    dir: FileFolder;
begin
    create dba transient;
    create dir transient;
    dir.fileName := "C:\backup";
    dir.make();
    dba.backupAllDbFiles("C:\backup", true, false, false, true, false, null);
    terminate;
epilog
    delete dba;
    delete dir;
end;
```

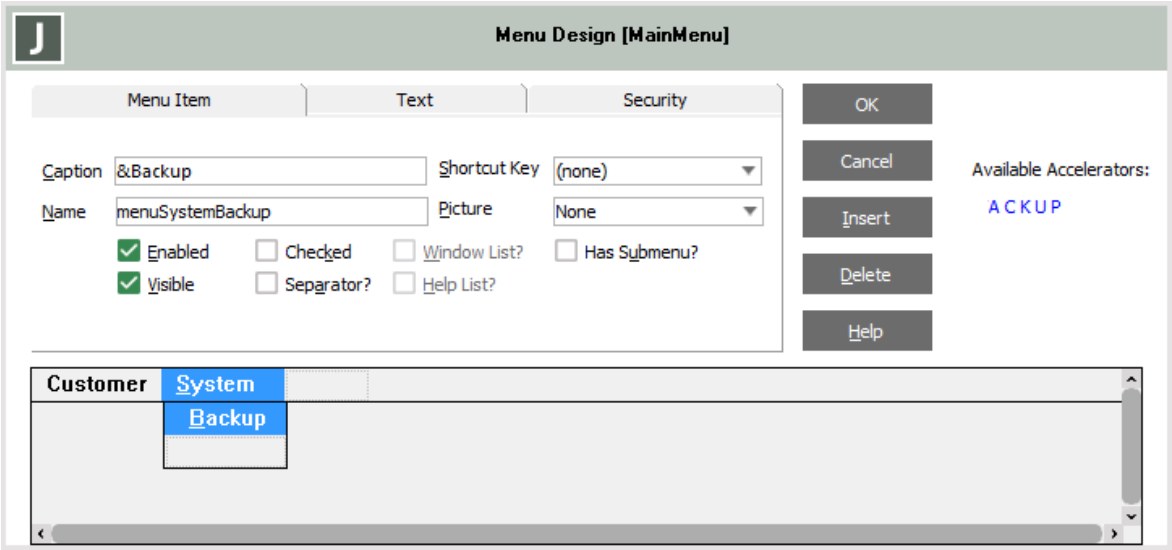
4. Open the Application Browser.
5. Select the Application menu **Add** command.
6. Enter **Backup** as the name of the application.
7. Select **Non-GUI** as the application type.
8. Select **backup** as the **Initialize Method**, and then click the **OK** button.
9. Run the application, by clicking the green arrow in the JADE development environment toolbar and then selecting **Backup** from the combo box.

Exercise 12.6 – Adding Backup to the MainMenu

In this exercise, you will change the application that was automatically added when the schema was created. This application has the same name as the schema.

1. Open the **MainMenu** 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 **menuSystem** in the **Name** field.

4. Select the menu item under the **System** menu, and then enter **&Backup** in the **Caption** field and **menuSystemBackup** 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 **menuSystemBackup** menu item and then select the **click** method.

7. Code the method as follows.

```
menuSystemBackup_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

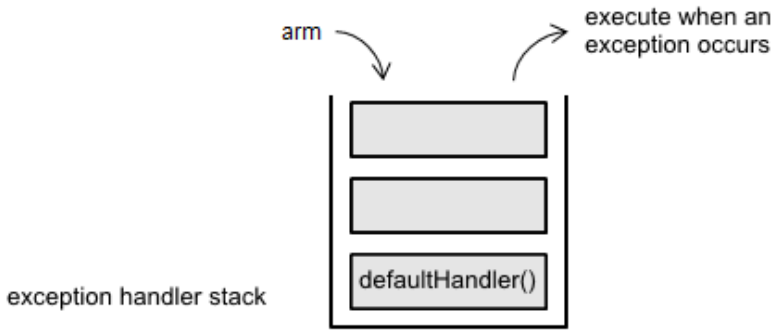
When an application is running, methods execute without error most of the time. Exceptions are error conditions that occur relatively rarely. A pessimistic approach to errors is to check constantly for things that could possibly go wrong, thereby attempting to prevent exceptions from ever occurring. 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 on disk in a non-GUI application.

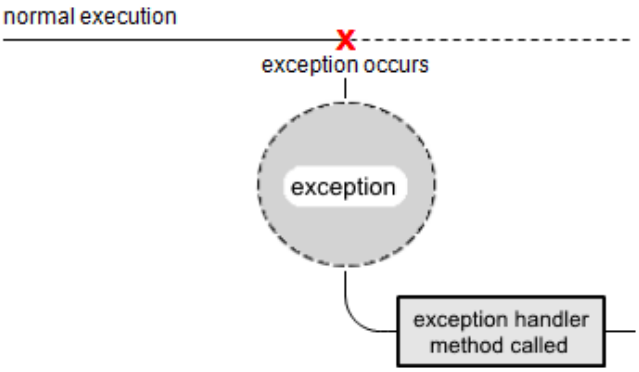
When an error occurs in an application, an instance of **Exception** or one of its subclasses is created by JADE or by your application code. This object contains information about the condition that resulted in the exception being raised; for example, a **FileException** object contains a reference 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.



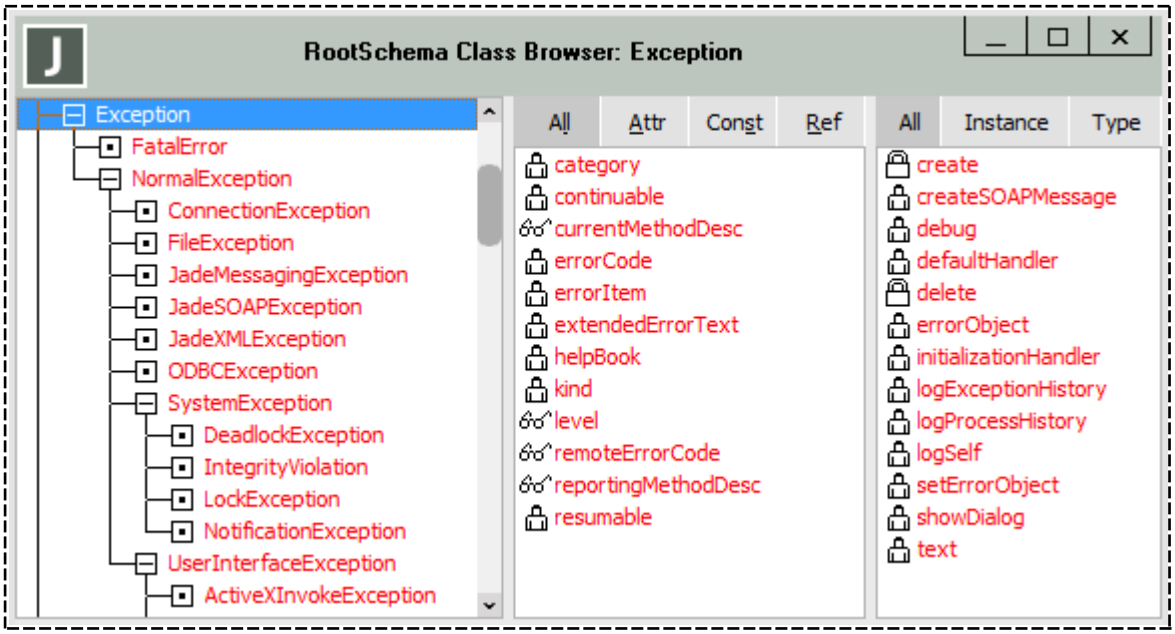
There are two exception handler stacks: a stack for locally armed handlers that are automatically disarmed at the end of the method, and a stack for globally armed handlers that are usually armed when an application starts and that are not disarmed until the application terminates. You can arm up to 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 to disk.

Default Exception Handler

JADE provides a default exception handler, which displays the Unhandled Exception dialog and logs exception information to disk. 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 next 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.

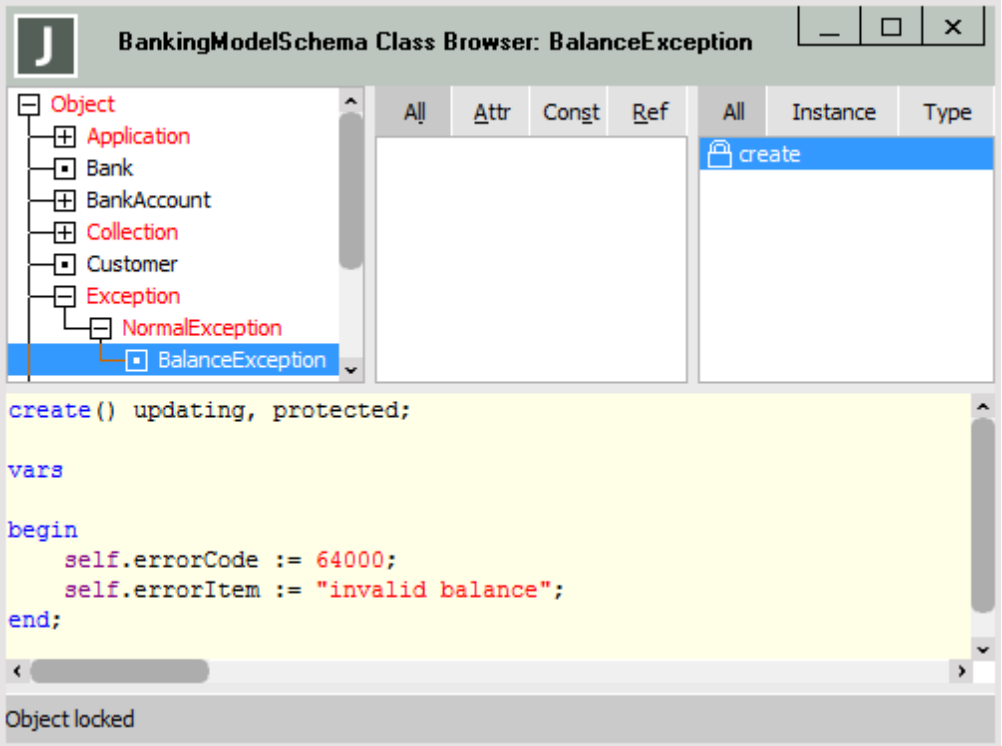
User Exceptions

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 diagram.



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 **setPropsOnCreate** and **setPropsOnUpdate** methods of a bank account class.

```
setPropsOnCreate(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.

If **set** is **false**, **_value** is the value of the property returned to the calling method.

Exercise 13.1 – Causing an Exception

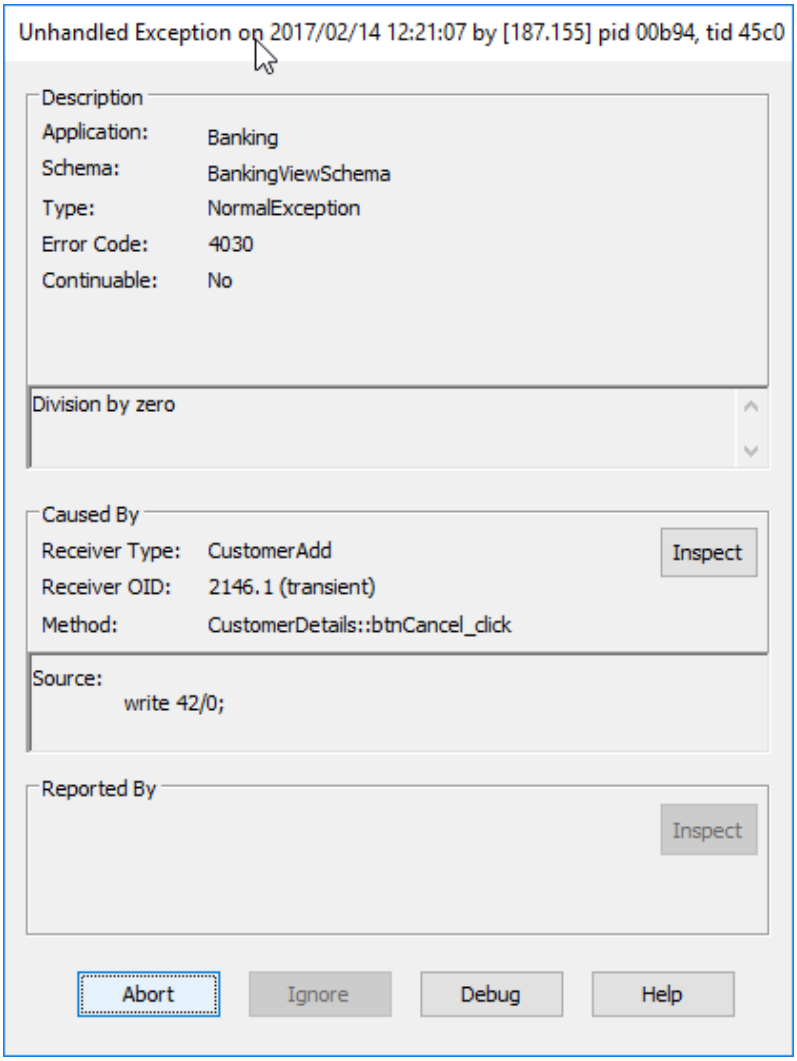
In this exercise, you will add code that deliberately causes an exception.

1. Open a Class Browser for the **BankingViewSchema**.
2. Select the **CustomerDetails** 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 **CustomerAdd** form.

5. Click the **Cancel** button, to display the unhandled exception dialog shown in the following diagram.



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.

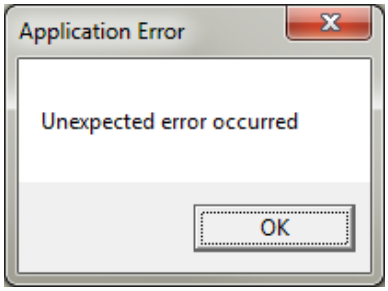
```
genericExceptionHandler(exObj: Exception): Integer;

begin
    abortTransaction;
    exObj.logSelf("errors.log");
    app.msgBox("Unexpected error occurred", "Application Error", MsgBox_OK_Only);
    return Ex_Abort_Action;
end;
```

4. Arm the exception handler globally at the start of the **initialize** method, as follows.

```
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;
```

5. Run the **Banking** application in the **BankingViewSchema**.

6. Open the **CustomerAdd** 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 **CustomerAdd** form.

3. Code the **load** event method for the form as follows.

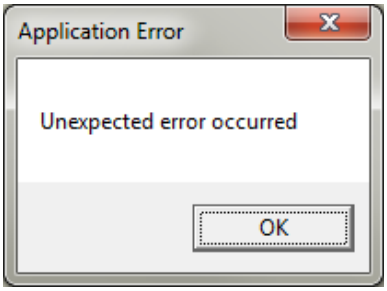
```
load() updating;

begin
    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 **CustomerAdd** 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 diagram.



Exercise 13.4 – Adding a Local Exception Handler

In this exercise, you will add a local exception handler in your **CustomerAdd** 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 **CustomerAdd** class.

2. Add a method called **stringTooLongHandler** and code the method as follows.

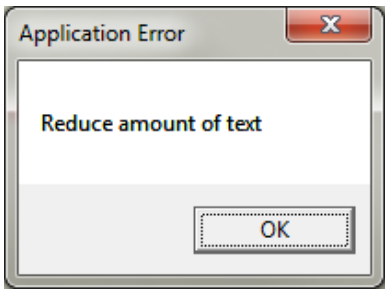
```
stringTooLongHandler(exObj: Exception): Integer;

begin
    if exObj.errorCode = 1035 then
        // Abort database transaction to release locks
        abortTransaction;
        exObj.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 exception 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;

begin
    on Exception do self.stringTooLongHandler(exception);
    if self.isDataValid() then
        self.createCustomer();
        self.clearTextBoxes();
        statusLine.caption := "Customer successfully added";
    endif;
end;
```

4. Run the **Banking** application and then open the **CustomerAdd** form.

5. 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 – Raising an Exception

In this exercise, you will raise a user exception to enforce the business rule that the address of a customer should not be **Tax Haven**, by raising an exception when an attempt is made to assign that value. You will implement this rule by adding a mapping method for the **address** property, and then test it by running the **Banking** application.

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

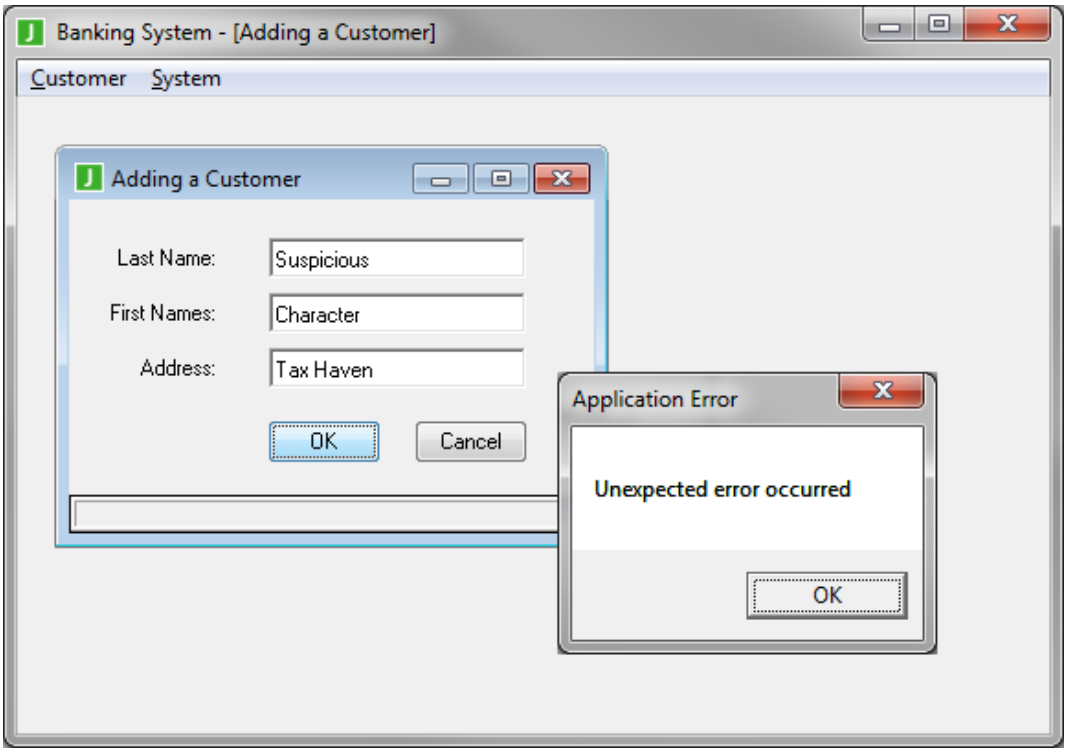
2. Select the **Customer** class.

3. Add a method called **address** and code the method as follows.

```
address(set: Boolean; _value: String io) mapping;

vars
    ex: Exception;
begin
    if set and _value = "Tax Haven" then
        create ex;
        ex.errorCode := 12345;
        raise ex;
    endif;
end;
```

4. Run the **Banking** application and then open the **CustomerAdd** form.

5. Enter information for a new customer with an address of **Tax Haven**. When you click the **OK** button, an exception should be raised.



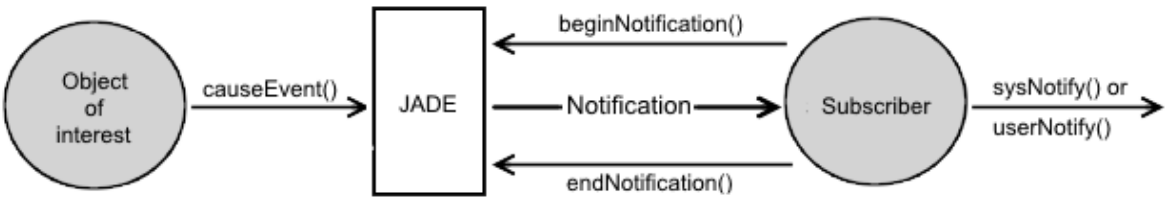
Module 14 Notifications and Timers

This module contains the following topics.

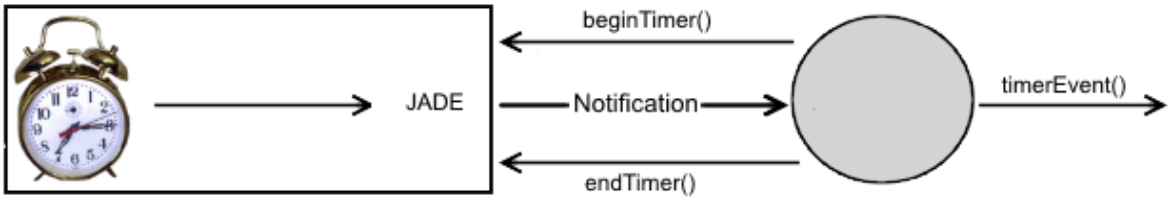
Introduction

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 diagram 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.

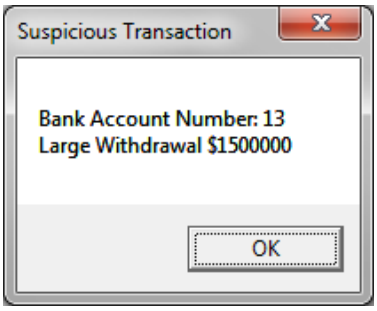
User Events

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 diagram 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
```

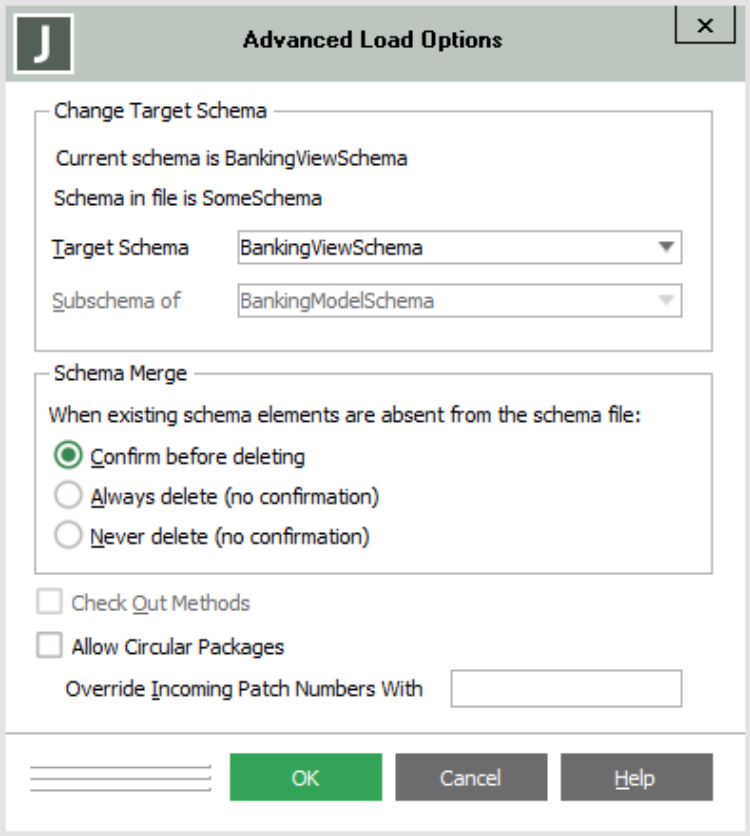
For controls and forms, you can code the **sysNotify** and **userNotify** event methods instead of the corresponding **sysNotification** and **userNotification** methods.

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.

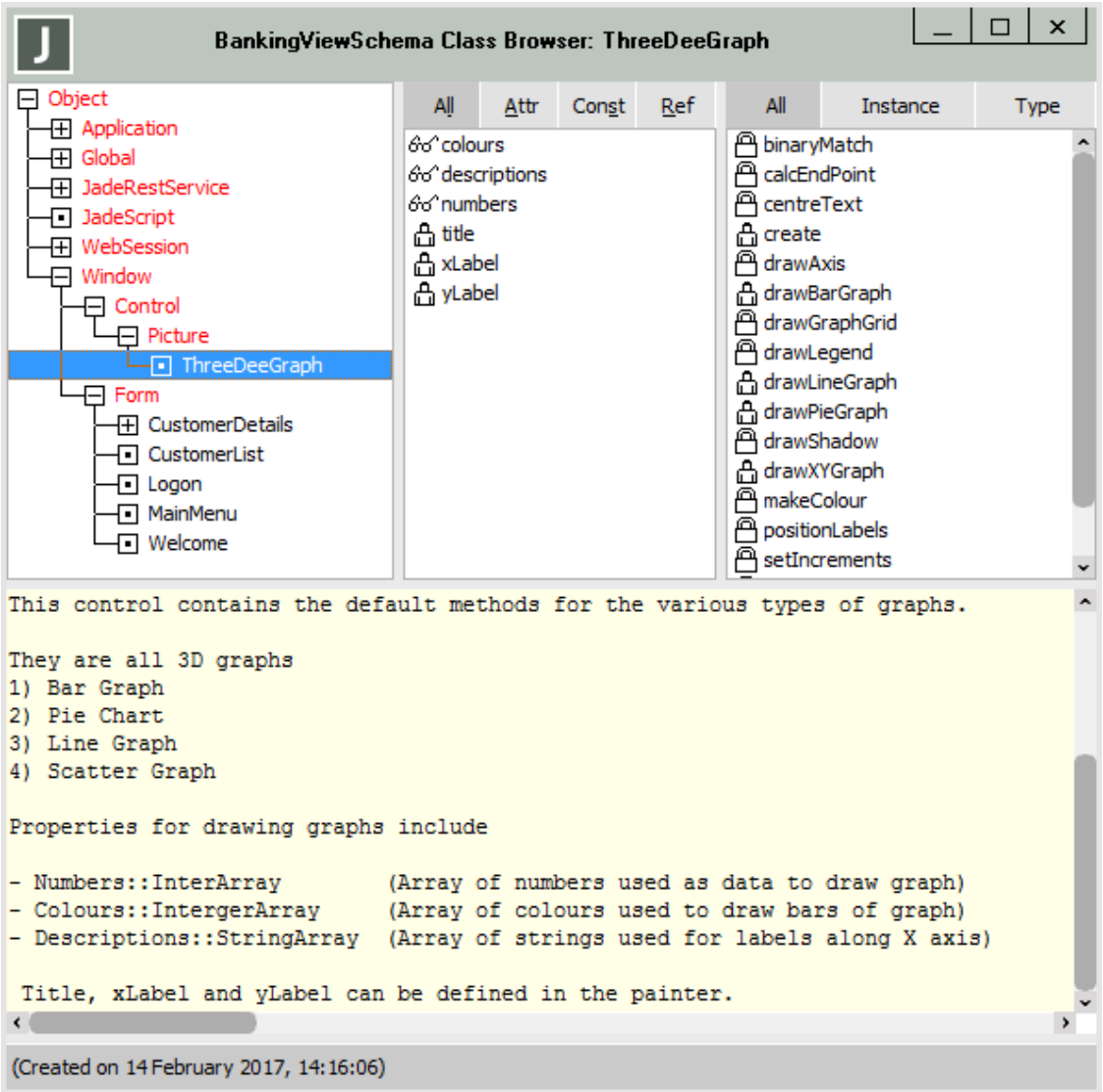
1. Select the Schema Browser.
2. Select the Schema menu **Load** command.
3. In the **Schema File Name** text box, browse for the **C:\JadeCourse\Files\ThreeDeeGraph.cls** file.

4. In the **Forms File Name** text box, browse for the **C:\JadeCourse\Files\ThreeDeeGraph.ddb** file.
5. Click the **Advanced** button, to open the Advanced Load Options dialog shown in the following diagram.



6. Select **BankingViewSchema** as the **Target Schema** and then click the **OK** button.

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.

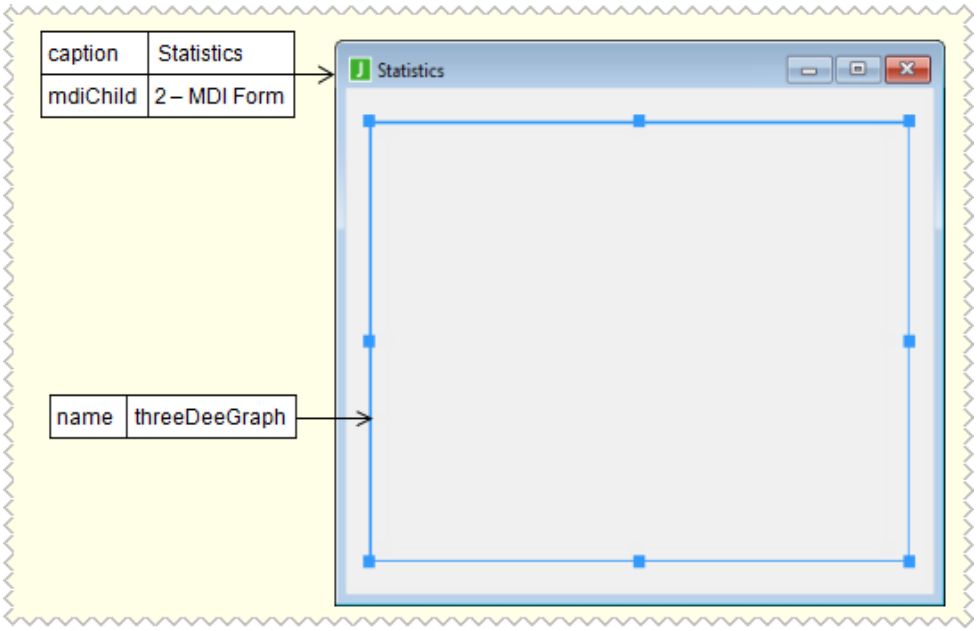
The arrays control the appearance of the bars when the control's **drawBarGraph** method is executed. The **numbers[1]** value is the height of the first bar, which is the number of customers. The value is obtained from the **size** of the **app.myBankAllCustomers** collection. The **colours[1]** value 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 **Statistics** as the name of the form.

3. Paint a **ThreeDeeGraph** control on the form and then save the form.



4. Add a **draw** method to the **Statistics** form and code it as follows.

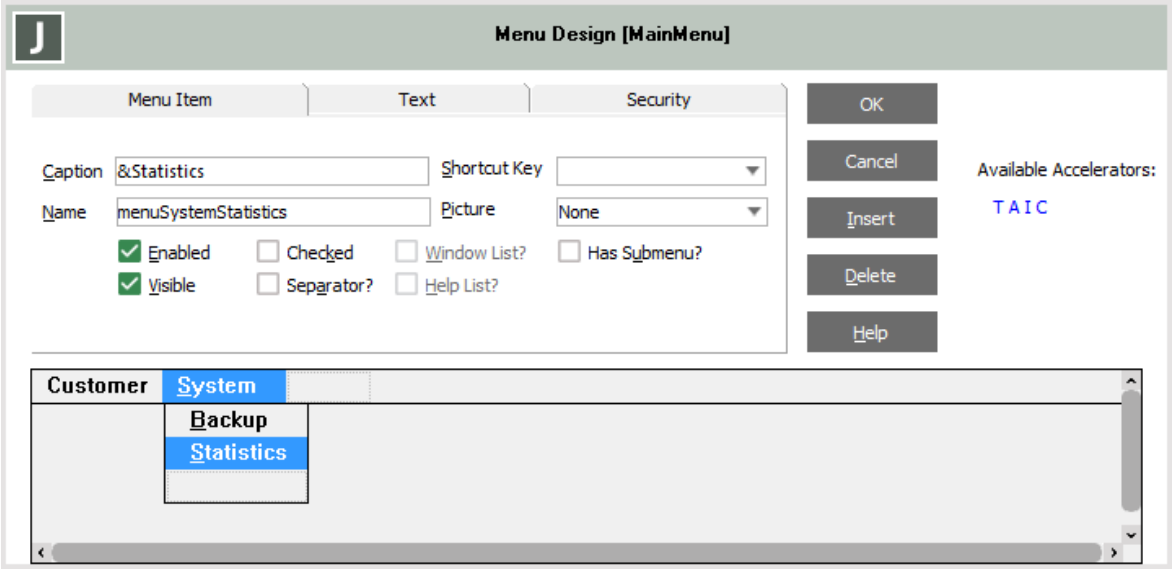
```
draw();

begin
    threeDeeGraph.descriptions[1] := "Customers";
    threeDeeGraph.descriptions[2] := "Accounts";
    threeDeeGraph.colours[1] := Red;
    threeDeeGraph.colours[2] := Blue;
    threeDeeGraph.numbers[1] := app.myBank.allCustomers.size();
    threeDeeGraph.numbers[2] := app.myBank.allBankAccounts.size();
    threeDeeGraph.drawBarGraph();
end;
```

5. Add code to the **load** method for the **Statistics** form to call the **draw** method and subscribe to create and delete notifications on the **Customer** and **BankAccount** classes, as follows

```
load() updating;

begin
    beginClassNotification(Customer, false, Object_Create_Event,
                           Response_Continuous, 0);
    beginClassNotification(Customer, false, Object_Delete_Event,
                           Response_Continuous, 0);
    beginClassNotification(BankAccount, false, Object_Create_Event,
                           Response_Continuous, 0);
    beginClassNotification(BankAccount, false, Object_Delete_Event,
                           Response_Continuous, 0);
    self.draw();
end;
```

6. Add code to the **unload** method for the **Statistics** form, to unsubscribe from the notifications.

7. Add code to the **sysNotify** method for the **Statistics** form, to redraw the graph by calling the **draw** method.
8. Add a menu item called **menuSystemStatistics** to the **MainMenu** form, as shown in the following diagram.



9. Add code to the **menuSystemStatistics_click** method, to display the **Statistics** form.
10. Test your notifications, by leaving the **Statistics** form open while you add customers.

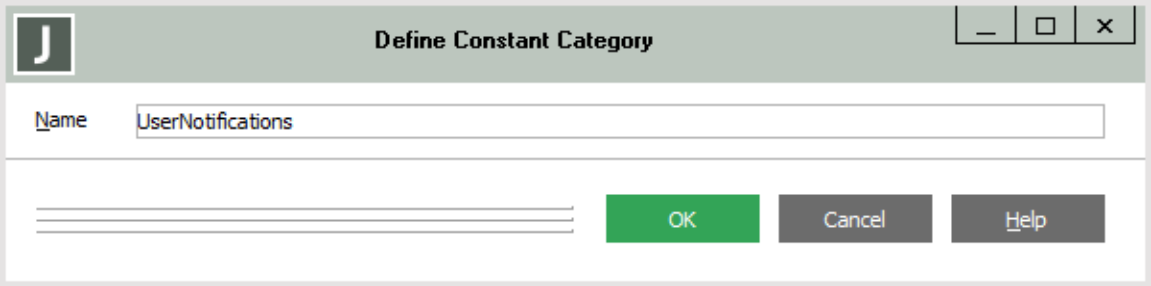
Exercise 14.3 – Defining a Global Constant

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**.

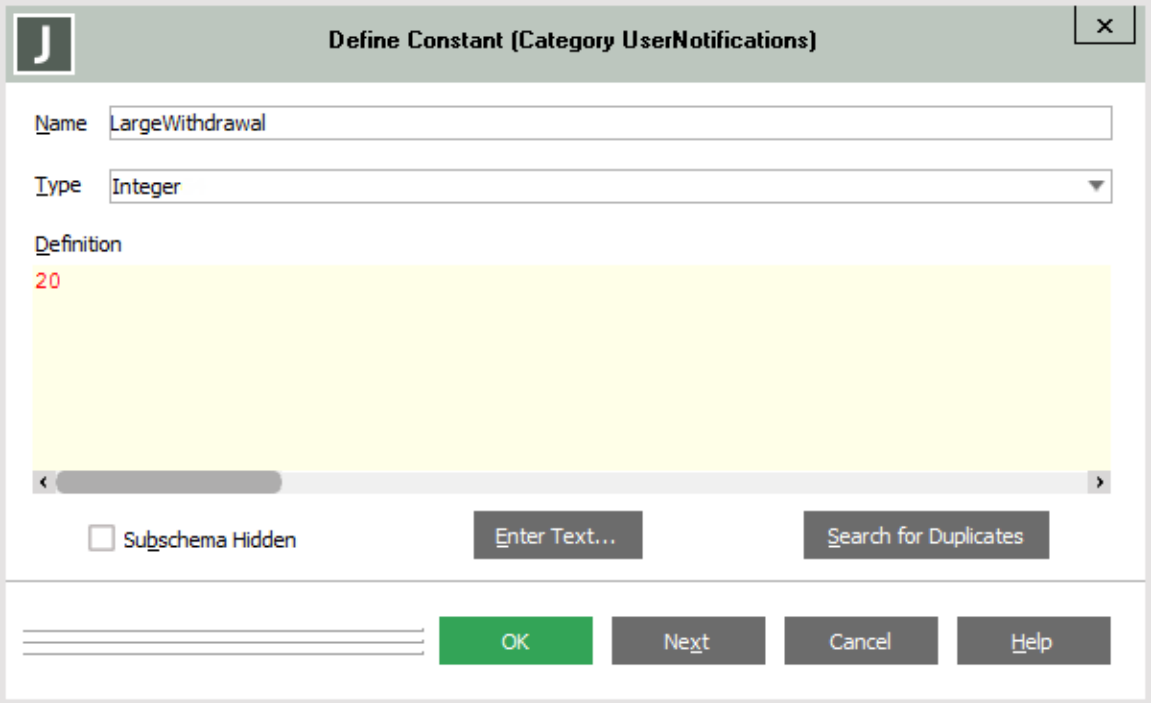
In the next exercise, you will use the **LargeWithdrawal** constant for a user notification.

1. Select **BankingModelSchema** in the Schema Browser.
2. Open the Global Constants Browser by selecting the Browse menu **Global Constants** command.

3. Select the Categories menu **Add** command and then enter **UserNotifications** as the name.



4. Select the Constants menu **Add** command and then enter a constant called **LargeWithdrawal** with a value of **20**.



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 **withdraw** method of the **BankAccount** class will cause a **LargeWithdrawal** user event if more than $1,000,000 is withdrawn.
- In the **BankingViewSchema**, the **MainMenu** form will subscribe to notifications of the **LargeWithdrawal** 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 uses the **withdraw** method to withdraw $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 **withdraw** method, 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;
```

4. Open a Class Browser for the **BankingViewSchema** schema.
5. Select the **MainMenu** form.
6. In the **load** event method, subscribe to notifications of the **LargeWithdrawal** event, as follows.

```
load() updating;

begin
   app.mdiFrame := MainMenu;
   beginClassNotification(BankAccount,false,LargeWithdrawal,Response_
Continuous,0);
end;
```

7. Unsubscribe from notifications in the **unload** event method.
8. Code the **userNotify** method, as follows.

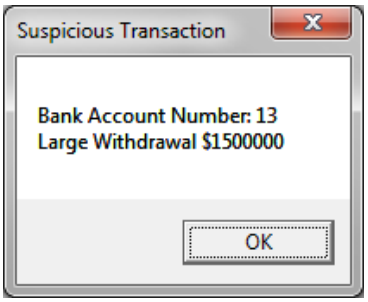
```
userNotify(eventType: Integer; theObject: Object; eventTag: Integer;
           userInfo: Any) updating;

vars
    account: BankAccount;
begin
    account := theObject.BankAccount;
    app.msgBox("Bank Account Number: " & account.number.String & CrLf &
               "Large Withdrawal $" & userInfo.String,
               "Suspicious Transaction", MsgBox_OK_Only);
end;
```

9. Add a JadeScript method called **makeLargeWithdrawal**, and code it as follows.

```
makeLargeWithdrawal();

vars
    cheque: ChequeAccount;
begin
    app.initialize();
    beginTransaction;
    create cheque persistent;
    cheque.setPropsOnCreate(2000000, 0, null);
    cheque.withdraw(1500000);
    commitTransaction;
end;
```

10. Run the **Banking** application.

11. Execute the **makeLargeWithdrawal** JadeScript method. The **Banking** application should display the following message box.



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 **MainMenu** form to change its background color every second. The timer will be started in the **load** event method and stopped in the **unload** event method. You will implement the **timerEvent** method for the form.

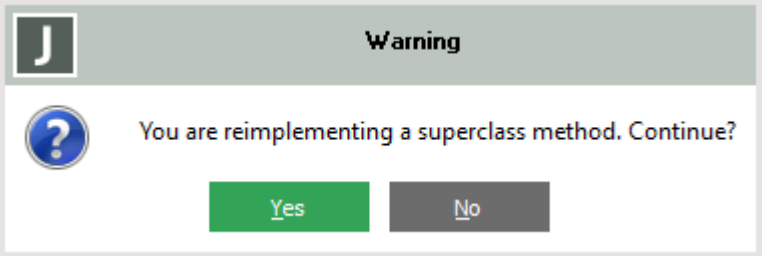
1. Open a Class Browser for the **BankingViewSchema**.
2. Select the **MainMenu** form.
3. Add an instruction to the **load** event method to start the timer, as follows.

```
load() updating;

begin
  app.mdiFrame := MainMenu;
  beginClassNotification(BankAccount,false,LargeWithdrawal,Response_
Continuous,0);
  self.beginTimer(1000, Timer_Continuous, 0);
end;
```

4. Stop the timer in the **unload** event method, as follows.

```
unload() updating;

begin
    self.endClassNotification(BankAccount, false, LargeWithdrawal);
    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;
```

7. Run the **Banking** application and test that the background color of the **MainMenu** form changes randomly.

Module 15 Nodes, Processes, and Caches

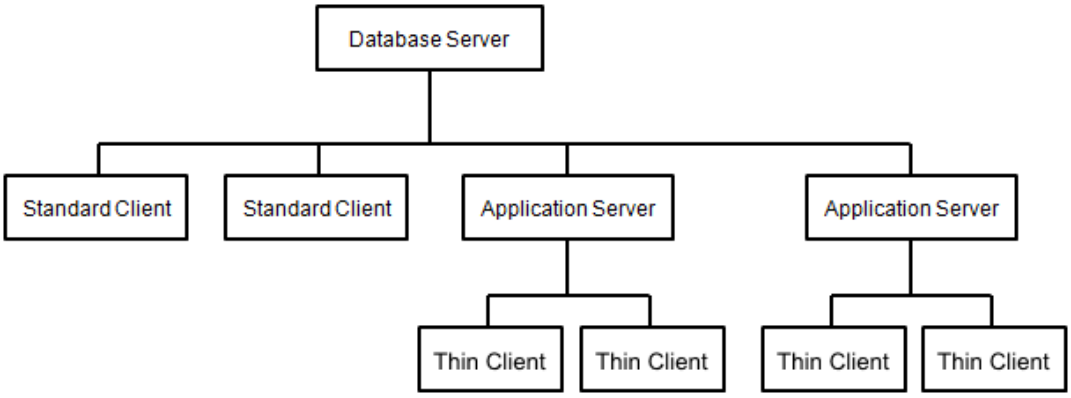
This module contains the following topics.

Introduction

This module contains an overview of the architecture of a JADE system, which is based on the concept of a *node*.

Distributed Processing

JADE 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

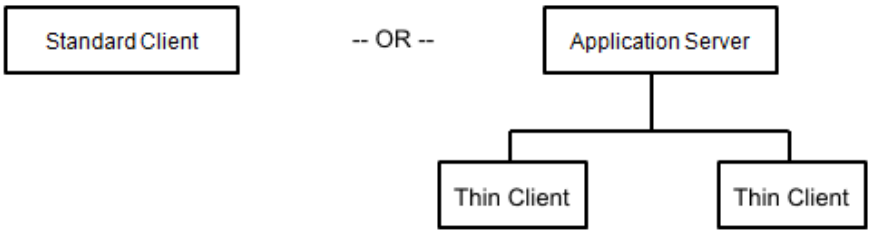
An application server:

- 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):

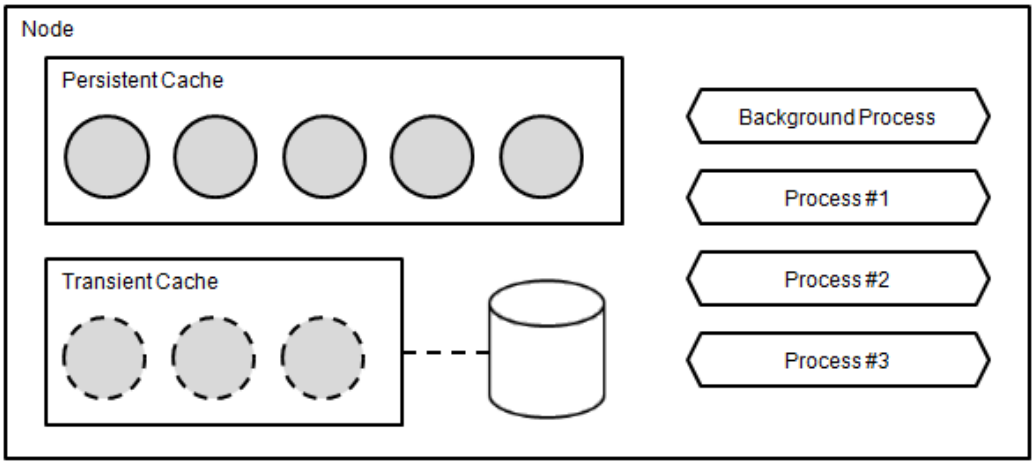
- Connects to an application server
- Does *not* execute application code or process objects (that is done by the application server)
- Does *not* require a high-bandwidth (LAN) connection to the application server

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;
```

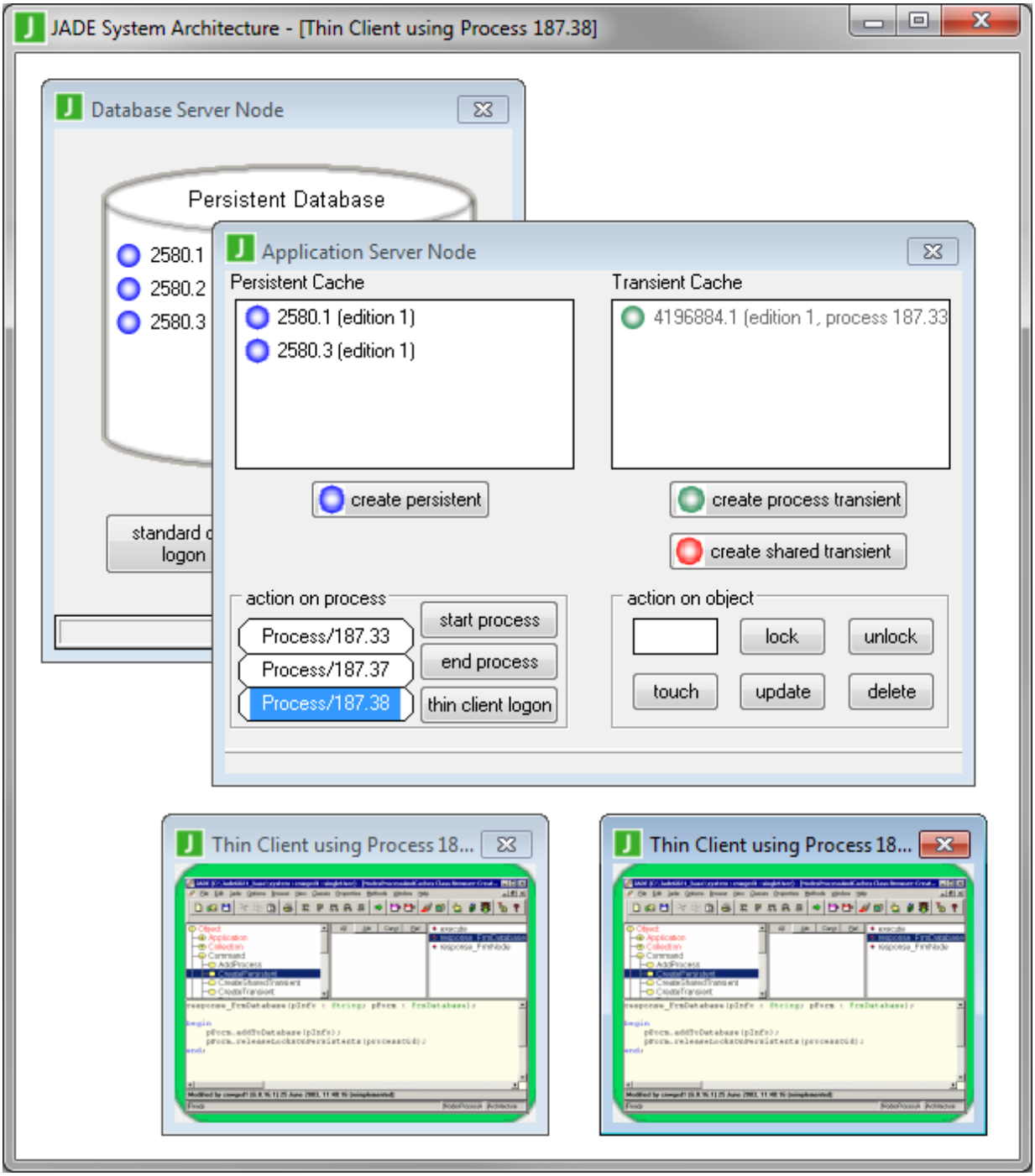
A transient object is stored locally in transient cache. It can be accessed only by the process that created it, and becomes unavailable when that process terminates or when it is explicitly deleted.

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.

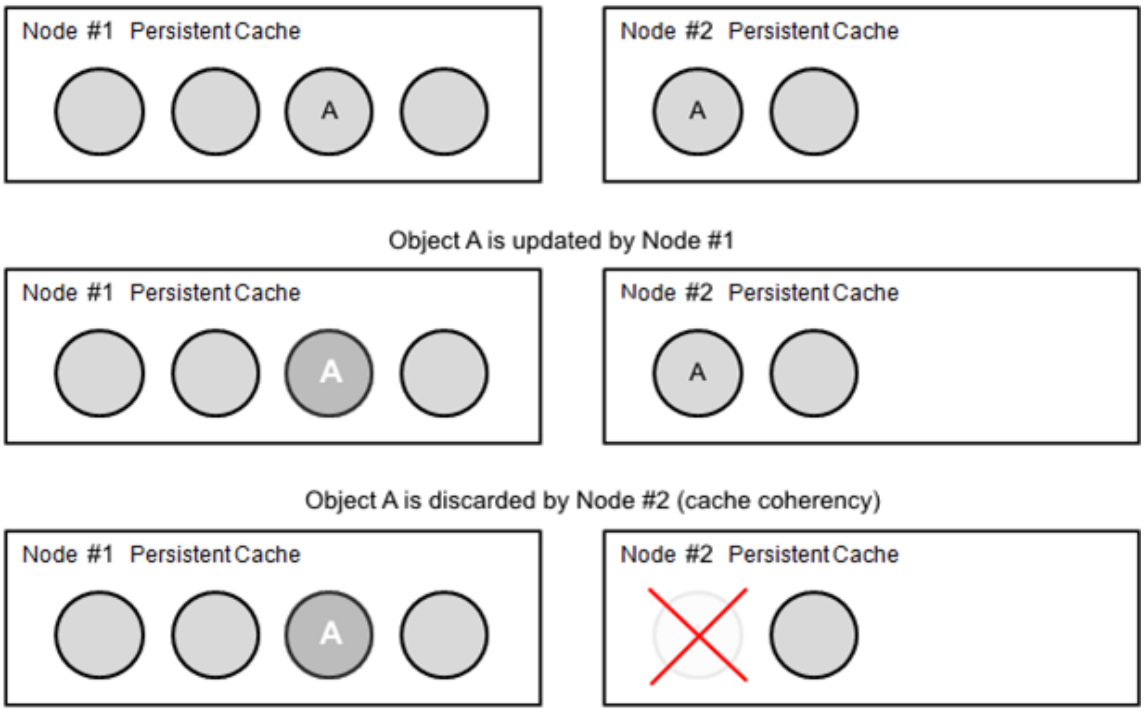
An updating transaction starts with the **beginTransaction** instruction. If the transaction is successful, the **commitTransaction** instruction releases all transaction duration locks and causes the new, updated, and deleted objects to be committed to the database.

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.

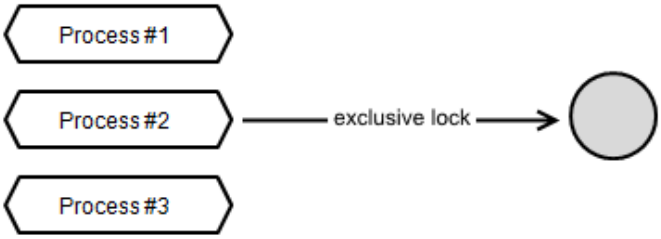
Lock Types

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.

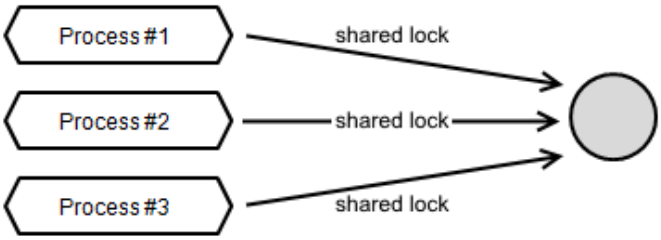
The lock types are:

- 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.

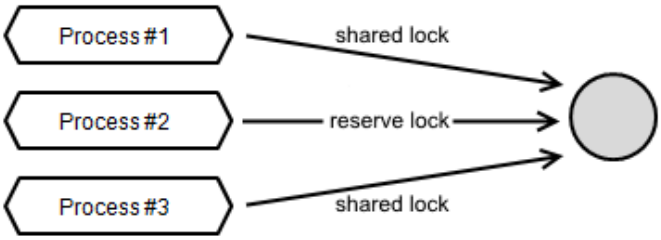
- Shared lock, which prevents other processes from updating the object while it is locked.



  Other processes can share lock the same object and one process can reserve lock the object.

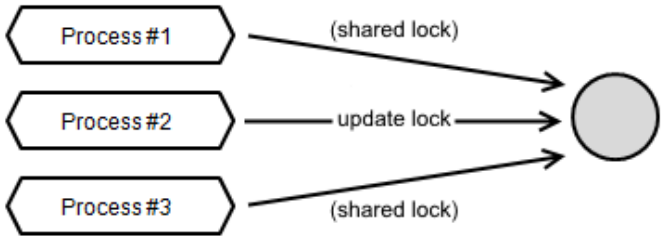
  Shared locks are automatically acquired on a collection that is being iterated using a **foreach** instruction, unless the **discreteLock** clause is specified. The shared lock is acquired for the duration of the iteration.

- 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.

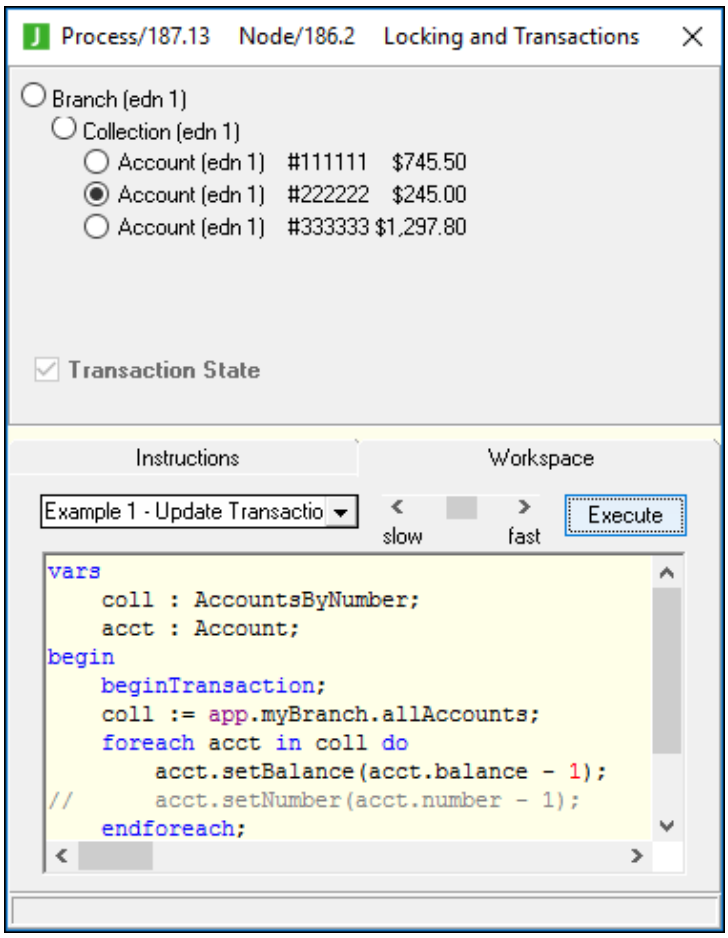
```
unlock(unlockTarget: Object);
```

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;
```

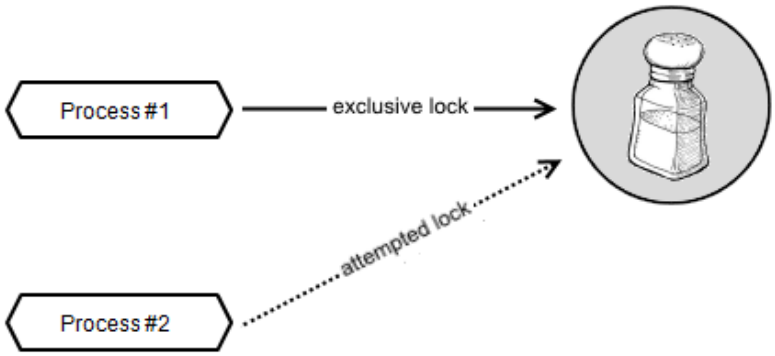
The **beginLoad** and **endLoad** instructions are similar to the **beginLock** and **endLock** instructions, but enable you to selectively lock objects.

```
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

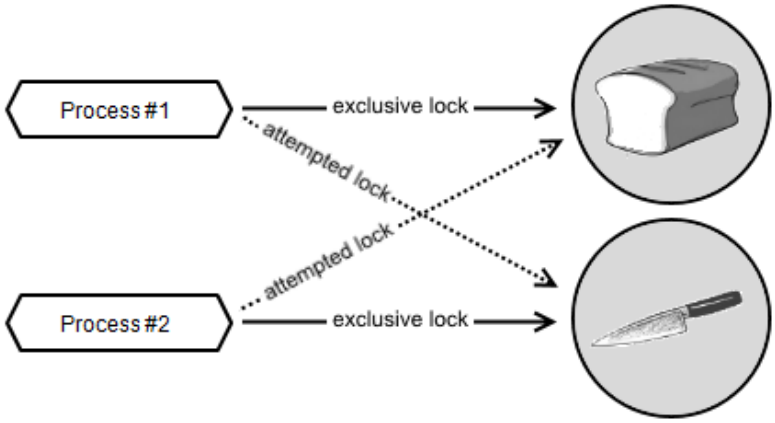
When a lock cannot be obtained (because another process already has the object locked with an incompatible lock), an exception is raised. The following analogies explain the difference between lock exceptions and deadlock exceptions, and the different ways they are handled.

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

JADE 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.

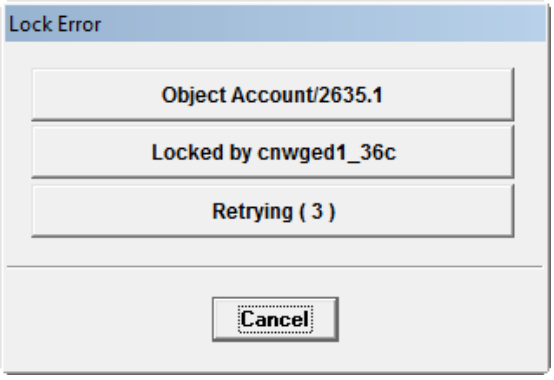
Lock Exception Object

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.

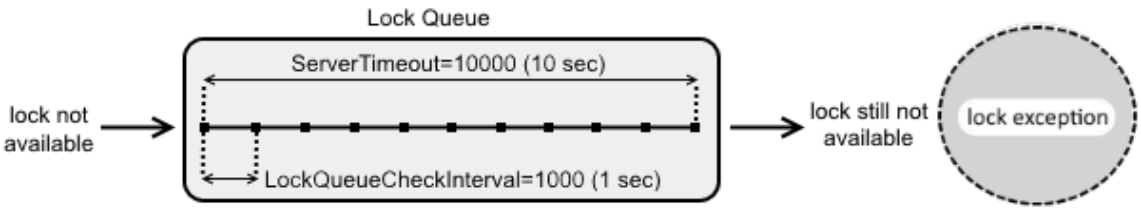


You would arm a lock exception handler globally when the application starts, as follows.

```
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.

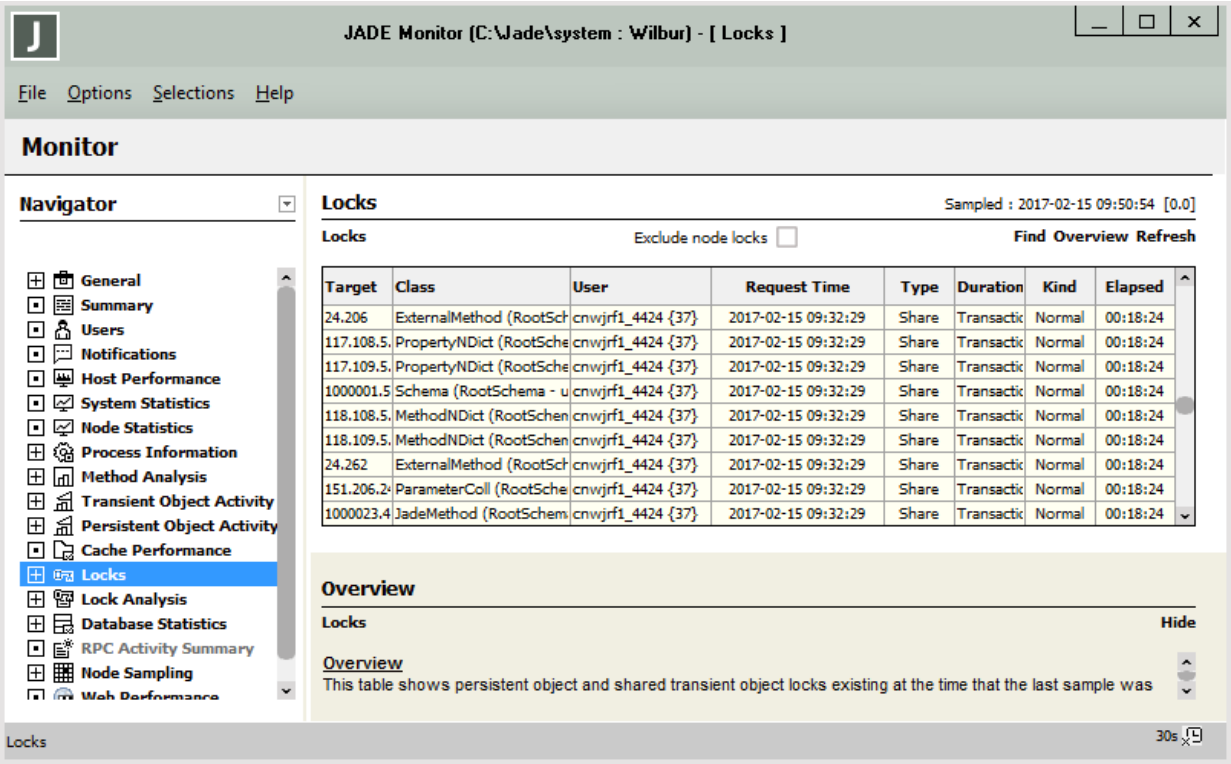


The lock queue is checked when an object is unlocked. It is also checked periodically, at an interval specified by the value of the **LockQueueCheckInterval** parameter in the [JadeServer] section of the JADE initialization file.

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.

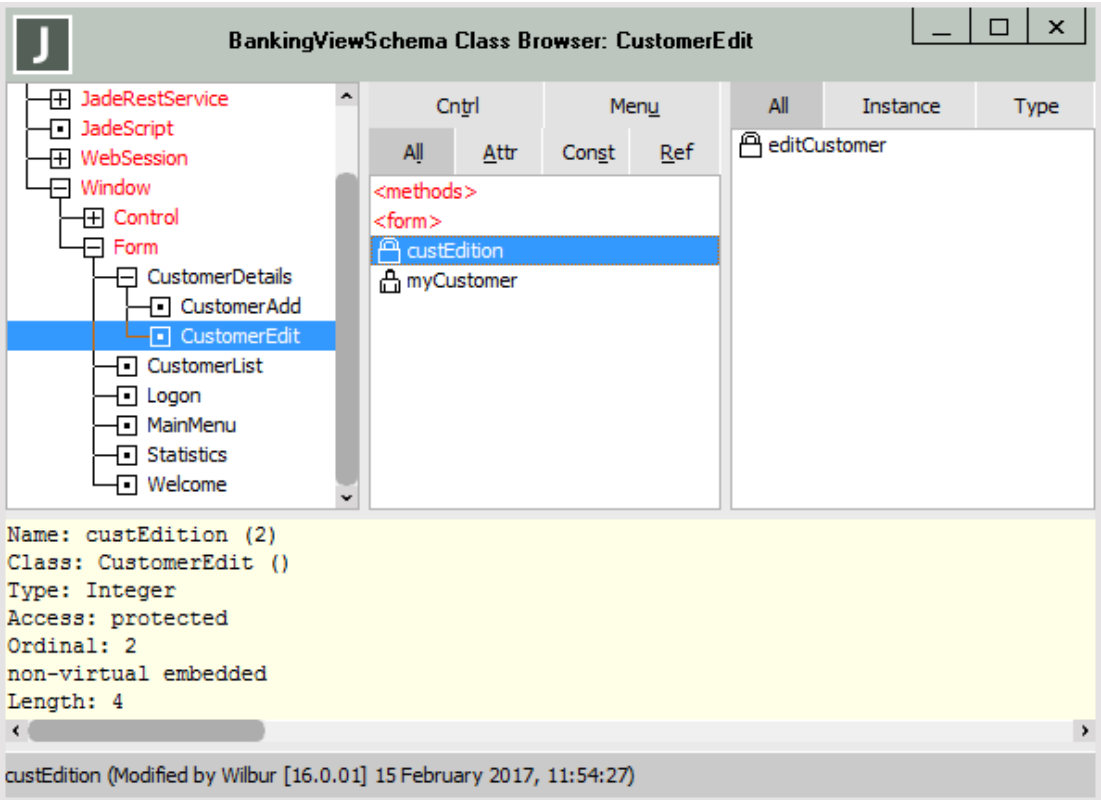
Exercise 16.1 – Using Locking to Check Editions

In this exercise, you will modify the **CustomerEdit** form to store the edition of the customer when the form is loaded. The edition will be checked when the **OK** button is clicked.

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 form reloaded with the latest edition of the customer.

Finally, you will test the edition, checking by opening two **CustomerEdit** forms for the same customer and then updating the customer on each.

1. Select the **CustomerEdit** form.
2. Add a protected attribute of type **Integer** called **custEdition**.



3. Change the **load** method to store the edition of the **myCustomer** object, as follows.

```
load() updating;

begin
    self.sharedLock(myCustomer);
    self.custEdition := myCustomer.edition();
    self.unlock(myCustomer);
    txtAddress.text := myCustomer.address;
    txtFirstNames.text := myCustomer.firstNames;
    txtLastName.text := myCustomer.lastName;
end;
```

4. Change the **btnOK_click** method to check the edition of the **myCustomer** object before proceeding with the update.

```
btnOK_click(btn: Button input) updating;

begin
    self.exclusiveLock(myCustomer);
    if self.custEdition < myCustomer.edition then
        app.msgBox("Customer changed by another user", "Reload", MsgBox_OK_Only);
        txtAddress.text := myCustomer.address;
        txtFirstNames.text := myCustomer.firstNames;
        txtLastName.text := myCustomer.lastName;
        self.unlock(myCustomer);
        return;
    endif;
    if self.isDataValid() then
        self.editCustomer();
        self.unloadForm();
    endif;
end;
```

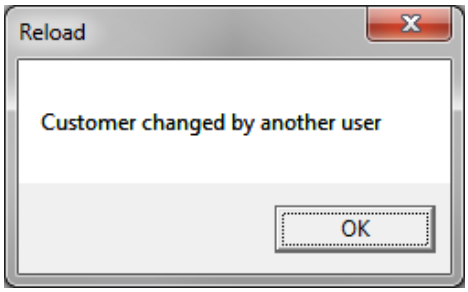
5. Run the **Banking** application and then open the **CustomerList** form.

6. Select **Charles Piggott** and then click the **Edit** button twice.

7. On the first **CustomerEdit** form, change the name to **Charles Smith** and then click the **OK** button.

8. On the second **CustomerEdit** form, change the name to **Charles Jones** and then click the **OK** button.

   The following message box should then be displayed.



Module 17 Printing

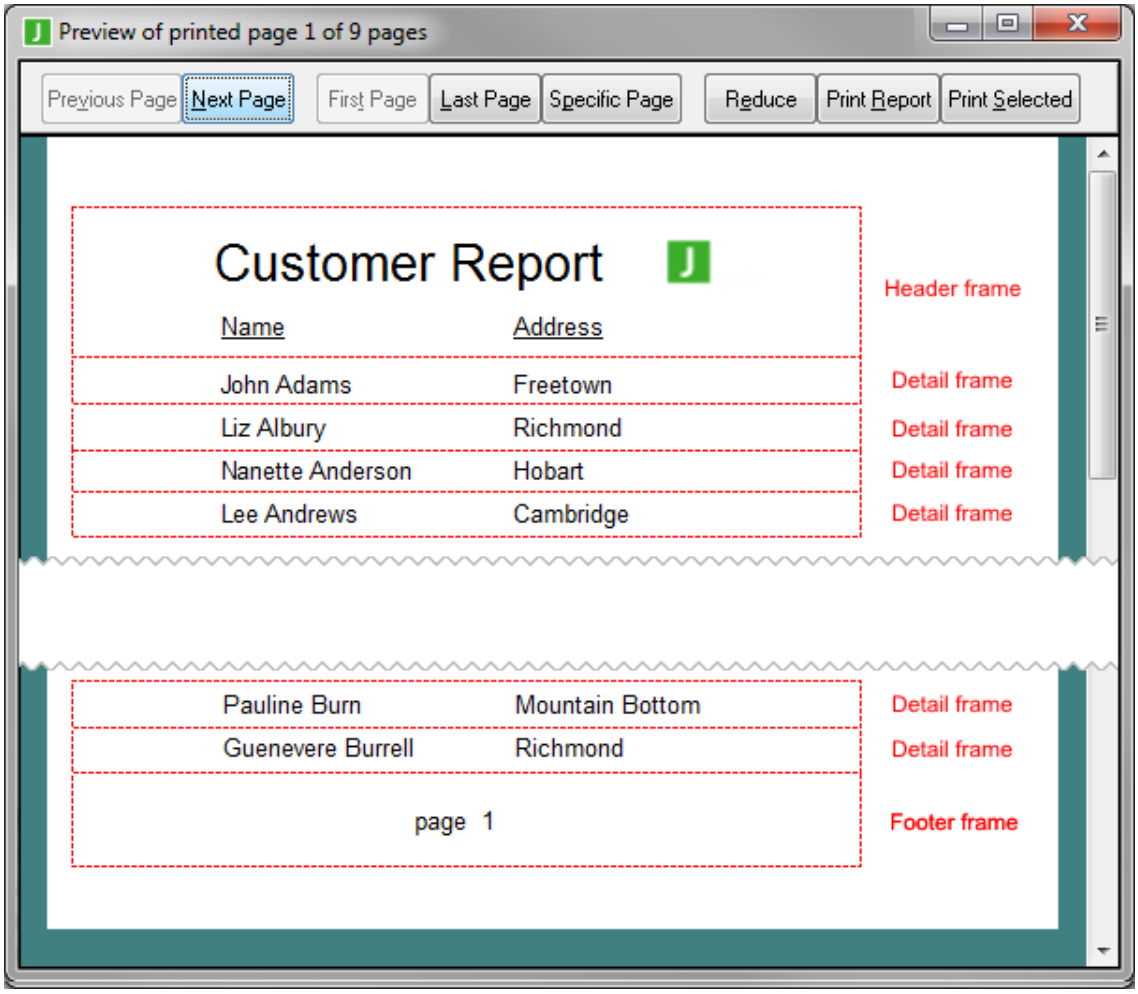
This module contains the following topics.

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 diagram 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.

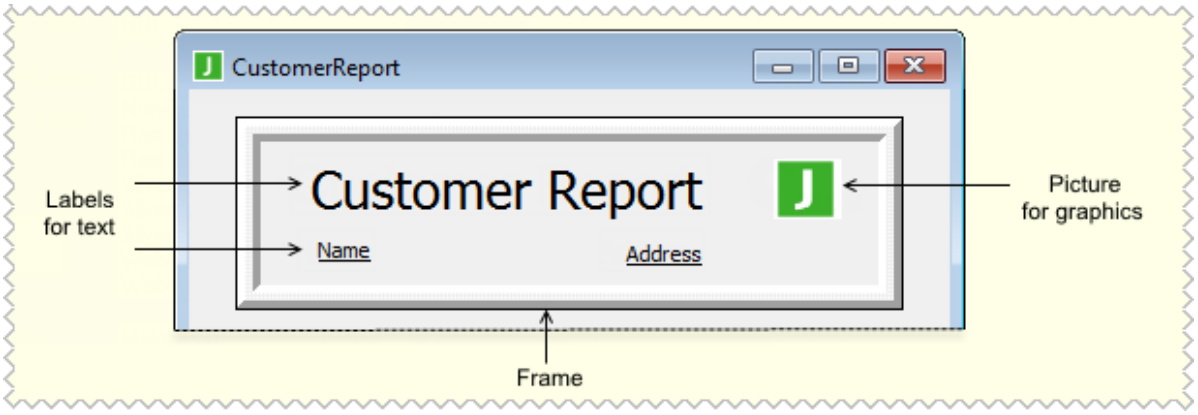
Designing a Report

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.

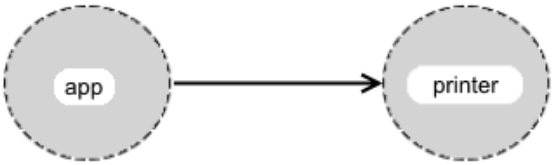
The following diagram shows a header frame containing three labels for text and a picture control for the company logo.



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;`

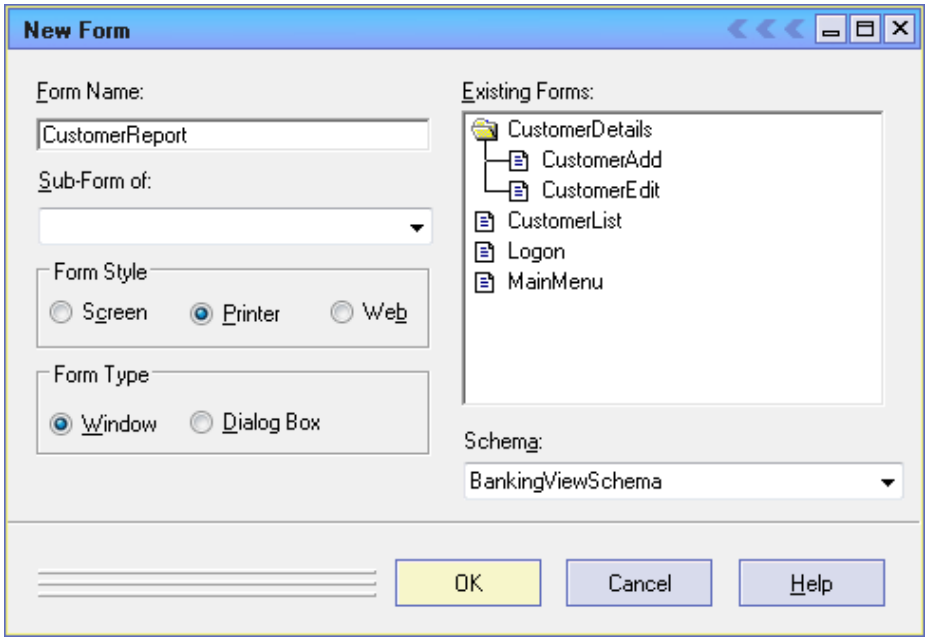
frameFits and newPage methods	Returns **true** if the specified report frame fits on the current page. The **newPage** method causes printing to skip to the next page.
	`if not app.printer.frameFits(fraDetail) then`
	`  app.printer.newPage();`
	`endif;`

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

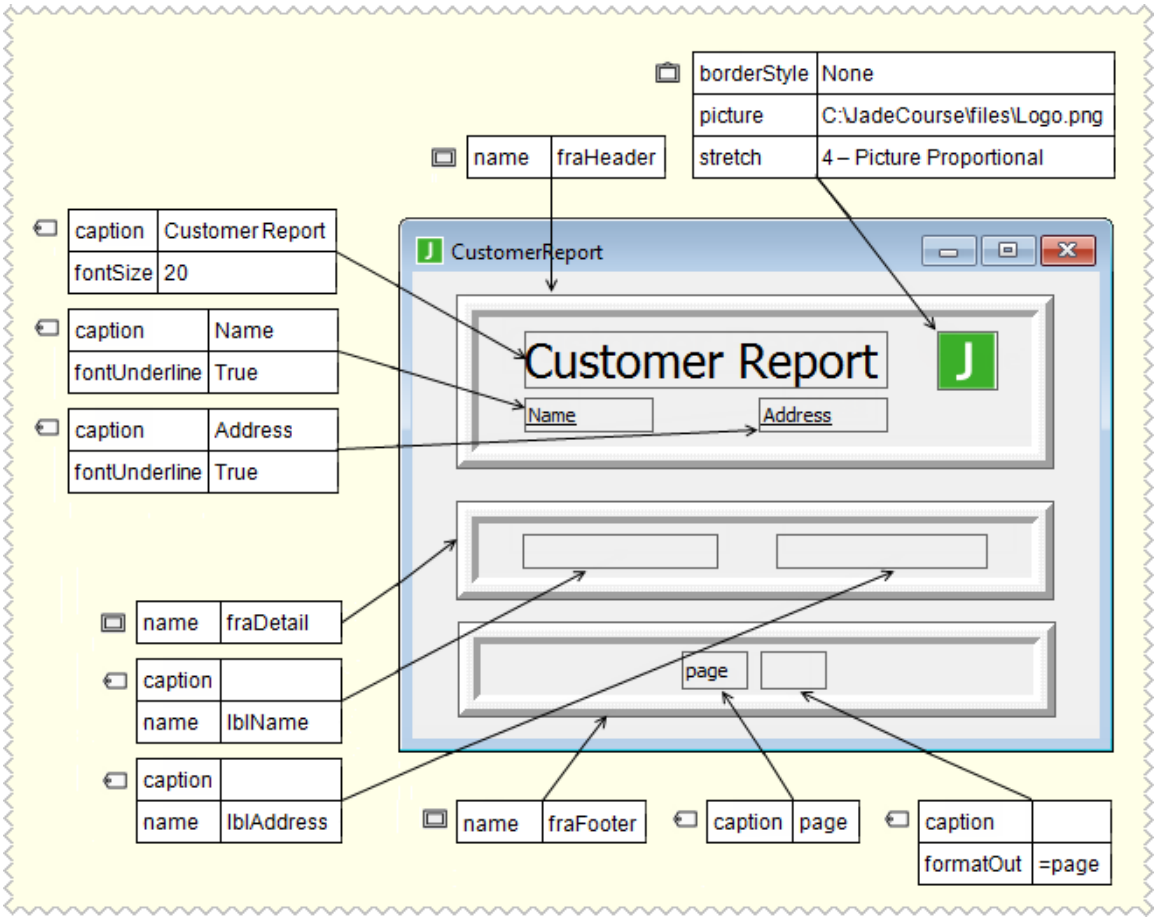
In this exercise, you will add a **CustomerReport** form in the JADE Painter.

1. Open the JADE Painter.
2. Select the File menu **New Form** command.



3. Enter **CustomerReport** 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 **CustomerReport** class. This method will print a report using the root object's collection of all customers.

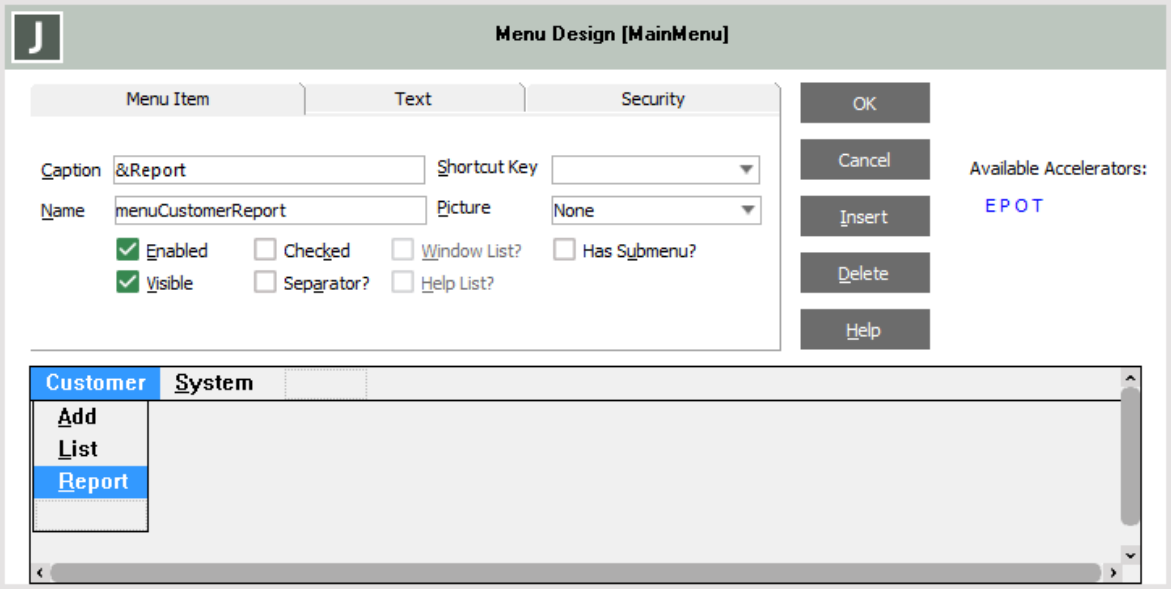
You will then add an option to the **Customer** menu on the **MainMenu** form to print the **CustomerReport**.

1. In the **CustomerReport** class, add a method called **print**.
2. Code the **print** method as follows.

```
print();

vars
    cust: Customer;
    result: Integer;
begin
    app.printer.printPreview := true;
    app.printer.setMargins(Print_Portrait, 10, 10, 10, 10);
    app.printer.setHeader(fraHeader);
    app.printer.setFooter(fraFooter);
    foreach cust in app.myBank.allCustomers do
        lblName.caption := cust.firstNames & " " & cust.lastName;
        lblAddress.caption := cust.address;
        result := app.printer.print(fraDetail);
        if result = Print_Cancelled then
            app.printer.abort();
            break;
        elseif result = Print_Stopped then
            app.printer.close();
            break;
        endif;
    endforeach;
epilog
    app.printer.close();
end;
```

3. Open the **MainMenu** 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 **menuCustomerReport** 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 **menuCustomerReport** menu item and then select the **click** method.

8. Code the method as follows.

```
menuCustomerReport_click(menuItem: MenuItem input) updating;

vars
    rpt: CustomerReport;
begin
    create rpt transient;
    rpt.print();
epilog
    delete rpt;
end;
```

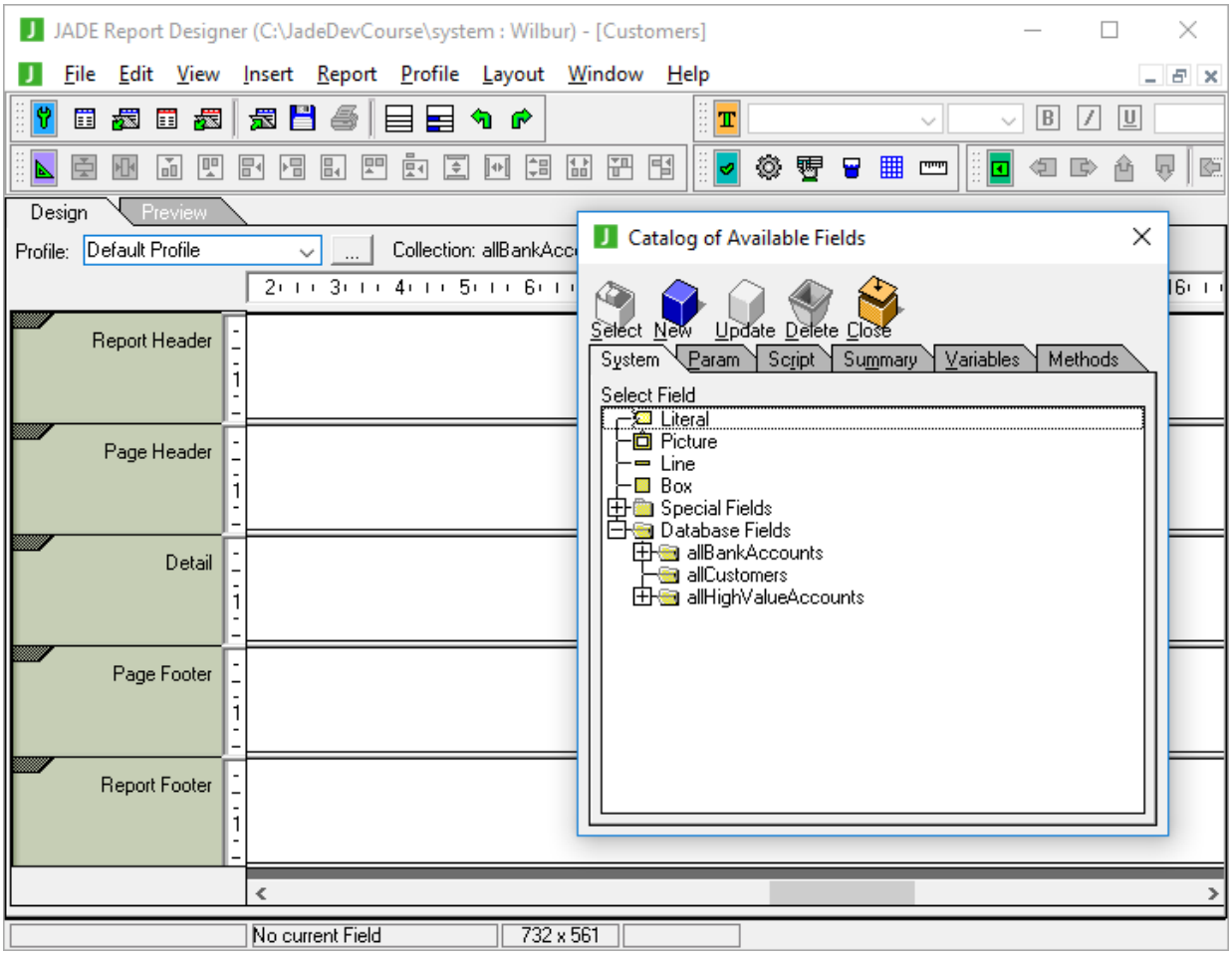
9. Run the **Banking** application and then view the report.

Report Writer

A report writer tool is provided to enable end-users and application developers to develop simple reports without programming, simply by dragging and dropping.

The first step is to load the **JadeReportWriterSchema** schema as a subschema of the JADE RootSchema. As the JADE Report Writer is installed with JADE itself, download the **JadeReportWriterSchema.scm** and **JadeReportWriterSchema.ddb** files from the **reportwriter** folder in the directory into which you installed JADE in Module 1 of this course (for example, **C:\JadeCourse\reportwriter***).

The report writer has two applications: a configuration application for designing *views*, which specify what is visible to the report designers, and a designer application for building reports.



In addition to printing the report, files can be output in text, RTF, CSV, HTML, and XML formats.

Tip You can download the **WP_ReportWriter.pdf** JADE Report Writer white paper in print (PDF) format from **White Papers** in the **Resource Library** section of **Developer-Center** at https://www.jadeworld.com/developer-center/resource-library/white-papers or view it in HTML5 format in your browser from https://www.jadeworld.com/docs/jade-2018/Default.htm.

JADE Developer's Course

Evaluation Form

jade™

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

JADE Developer's Course

Evaluation Form

jade™

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.

technologies
jadeworld.com