Multithreading

Version 2018

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

JADE Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of JADE Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2019 JADE Software Corporation Limited.

All rights reserved.

JADE is a trademark of JADE Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

Contents

Multithreading 4

Introduction 4
Synchronous versus Asynchronous 4
Nodes and Processes 5
Initiating Asynchronous Processes 5
    Exercise 1 – Synchronous versus Asynchronous 7
Shared Transient Objects 10
Notifications and Callbacks 11
    Exercise 2 – Generating a Callback 13
    Exercise 3 – Dealing with Shared Transient Leaks 16
Asynchronous Method Calls 18
    Exercise 4 – Invoking a Method using JadeMethodContext 19
    Exercise 5 – Waiting for Asynchronous Operations 23

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Multithreading

Introduction

In JADE applications, it is sometimes useful to initiate asynchronously executing applications, threads, or processes. These tasks may need to run asynchronously for performance reasons, to separate the execution of specific functions or to allow the processes to run on different machines.

Synchronous versus Asynchronous

Typically, when you call one method from another method, the calling method waits until the called method completes before executing the next line of code, as shown in the following example.

```
method1();

begin
    write "This is the first instruction";
    self.method2();
    write "I will wait until method2 finishes before continuing";
end;
```

However, sometimes the method to be called represents a background process and there is no reason to wait for it to complete before proceeding. In this case, we would start that process in a new thread (by *multithreading*, or having multiple threads) and have it run asynchronously so that it doesn't wait for it to complete before proceeding.

The most usual way to do this in JADE is to start a new application to run the method; for example:

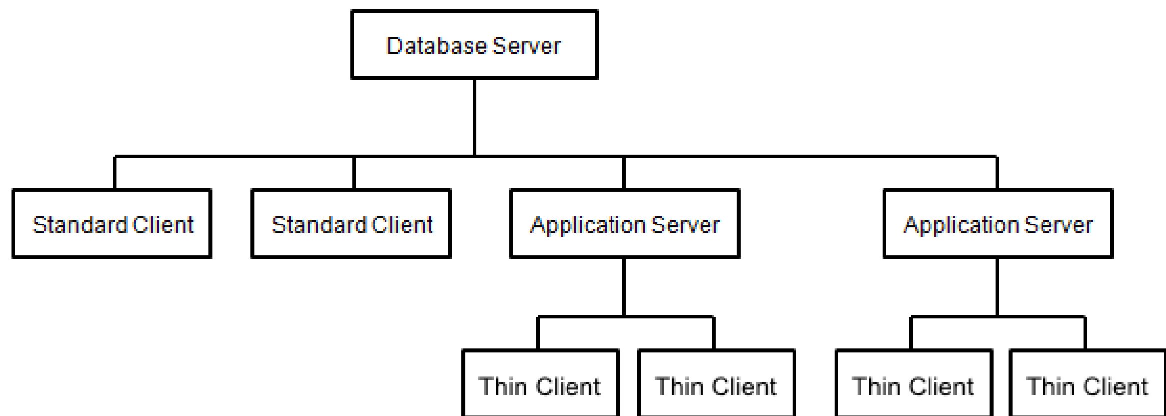
```
exampleMethod();

begin
    write "This is the first instruction";
    app.startApplication("Multithreading", "ExampleApp");
    write "I will NOT wait until method2 finishes before continuing";
end;
```

In this example, **Multithreading** is the name of the schema and **ExampleApp** is an application. The **exampleMethod** method starts the **ExampleApp** application, which will call the **method2** method. However, it will not wait for the **method2** method to complete before proceeding to the next instruction.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Nodes and Processes

In JADE, a database is broken down into multiple nodes, where each connection to the database contains a database server, an application server, and clients.



The application server and client can be combined into a single standard client (often called a *fat* client).

In a single user system, the database server, application server, and clients are all combined into a single **jade.exe** program.

Note For more information, see “Module 15 - Nodes, Processes, and Caches” in the JADE Developer’s course.

Each of the programs that contains an application server or a database server, or both an application server and a database server, represents a JADE *node.* These can include any of the following executing programs.

- A **jade.exe** program for a standard (fat) client.
- A **jadapp.exe** program for an application server.
- A **jadrap.exe** program for the database server.
- An external program that uses the JADE Object Manager; for example, a .NET application that uses a JADE exposure.

Each node can have several processes running asynchronously at any time.

Initiating Asynchronous Processes

There are five **RootSchema** JADE methods that allow you to start a new application process from an existing JADE process. Any application process that is started using one of these methods run asynchronously to the process that starts it.

- Application::startApplication
- Application::startApplicationWithParameter
- Application::startApplicationWithString
- Application::startAppMethod
- Node::createExternalProcess (which runs an external application asynchronously)

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

The simplest way to start a new application process is with the **Application** class **startApplication** method. The **startApplication** method takes two string parameters: the name of the schema containing the application and the name of the application.

```
exampleMethod();

begin
    write "This is the first instruction";
    app.startApplication("Multithreading", "ExampleApp");
    write "I will NOT wait until method2 finishes before continuing";
end;
```

If the application to be started requires a parameter for its **initialize** method, you can use the **Application** class **startApplicationWithParameter** method. This method takes one additional parameter, which is a shared transient object. (Although it can be persistent, it is usually a shared transient, because otherwise the other application could merely access it directly from the database.)

Note The parameter required by the **initialize** method must be an object; it cannot be a primitive type such as an **Integer**. If you need to pass through a primitive type, you can set it as a property on the object unless it is exactly a **String**, which is handled by the **startApplicationWithString** method.

```
exampleMethod();

vars
    objParam : ExampleClass;

begin
    beginTransientTransaction;
    objParam := create ExampleClass("Example String") sharedTransient;
    commitTransientTransaction;
    write "This is the first instruction";
    app.startApplicationWithParameter("Multithreading", "ExampleParamApp", objParam);
    write "I will NOT wait until method2 finishes before continuing";
end;
```

The passed object is then available to the **initialize** method of the application. In this case, the **ExampleParamApp** application has an **initialize** method called **paramMethod**.

Note The **initialize** method of an application is often called **initialize**, and **initialize** is the default method that is called by an application if you do not set one. However, you can call your **initialize** method whatever you like. This is especially useful if you have multiple applications with different **initialize** methods defined on the same schema.

```
paramMethod(str : ExampleClass io);

begin
    app.doWindowEvents(1000);
    write str.myString;
    beginTransientTransaction;
    delete str;
    commitTransientTransaction;
    terminate;
end;
```

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Tip Make sure you delete shared transient objects and terminate applications when you have finished with them.

A common use case of passing a parameter to an asynchronous application is to pass a single string. If the parameter you want to pass is exactly one string, you can use the **startApplicationWithString** method of the **Application** class to avoid having to wrap it in an object (as would be needed to use the **startApplicationWithParameter** method).

```
exampleMethod();

begin
    write "This is the first instruction";
    app.startApplicationWithString("Multithreading", "ExampleStringApp", "A Message.");
    write "I will NOT wait until the application finishes before continuing";
end;
```

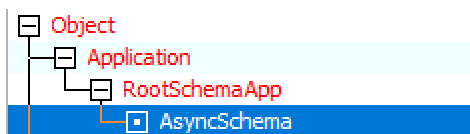
The **Application** class **startAppMethod** method allows the application to be started with an alternative **initialize** method. As this method provides the most flexibility of any methods that start an application, it also has the most parameters, which are listed in the following table.

Parameter	Type	Description
schemaName	String	Specifies the name of the schema in which the application is located.
appName	String	Specifies the name of the application to start.
methodName	String	Specifies the method that is to be invoked on the application; that is, the method to be called as the initialize method of the application.
methodParam	Object	A shared transient object to be passed to the initialize method of the application.
checkSecurity	Boolean	If set to **true**, the **getAndValidateUser** method is called to validate user codes and passwords. If **false**, the application inherits the security profile from the invoking application.

Exercise 1 – Synchronous versus Asynchronous

In this exercise, you will create a form to explore the difference between running a method synchronously versus asynchronously.

1. Create a schema called **AsyncSchema**.
2. In the Class Browser, navigate to the **AsyncSchema** subclass of the **Application** class.



Copyright 2019, JADE Software Corporation Ltd. All rights reserved

3. Add a method called **waitThenMsg** to the **AsyncSchema** application subclass, and code it as follows.

```
waitThenMsg();

vars

begin
    app.doWindowEvents(3000);
    app.msgBox("Thanks for waiting.", "MsgBox", MsgBox_OK_Only);
end;
```

4. Open the Application Browser and create an application, as follows.

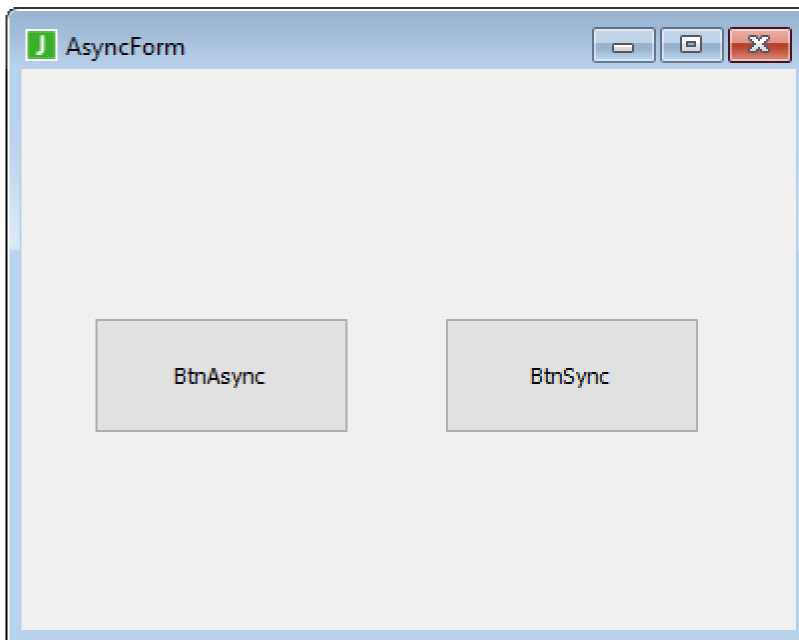
Define Application

Field	Value
Name	AsyncApp
Help File	
Version #	
Default Locale	
Application Type	GUI
Web Application Type	JADE Forms (selected), HTML Documents, Web Services
Icon	Change..., Clear
Startup Form	
About Form	
Show Super Class Methods	(unchecked)
Initialize Method	AsyncSchema::waitThenMsg
Finalize Method	

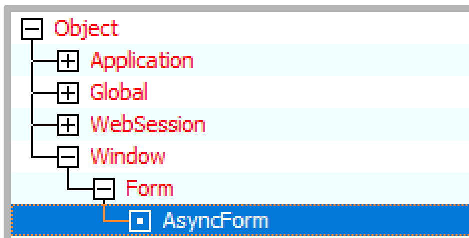
Tabs: Application, Form, Web Options. Buttons: Browse..., OK, Cancel, Help

 All rights reserved

5. Open the JADE Painter and create a form called **AsyncForm** with two buttons: **btnAsync** and **btnSync**, as follows.



6. Ensuring that you have first saved **AsyncForm**, navigate to **AsyncForm** in the Class Hierarchy Browser.



7. Modify the **click** method of **btnAsync**, by coding it as follows.

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

begin
    btnAsync.enabled := false;
    btnSync.enabled := false;

    app.startApplication("AsyncSchema", "AsyncApp");

    btnAsync.enabled := true;
    btnSync.enabled := true;
end;
```

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

8. Modify the **click** method of **btnSync**, by coding it as follows.

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

begin
    btnAsync.enabled := false;
    btnSync.enabled := false;

    app.waitThenMsg();

    btnAsync.enabled := true;
    btnSync.enabled := true;
end;
```

9. Create a JadeScript method called **createAsyncForm**, coding it as follows.

```
createAsyncForm();

vars
    form : AsyncForm;
begin
    create form;
    form.showModal();
epilog
    delete form;
end;
```

10. Run the **createAsyncForm** method and click on the two buttons.

    How do they behave differently?

Shared Transient Objects

If you have completed the JADE Developer's course, you are likely familiar with the difference between transient and persistent objects. If not, see "Module 15 - Nodes, Processes, and Caches" and "Module 16 - Transactions and Locking" of the JADE Developer's course.

Shared transient objects are half-way between transient and persistent objects.

- Persistent objects are stored in the database and are therefore accessible to *all* processes on *every* node of the database.
- Transient objects are stored in the transient cache on a node and are accessible to *one* process on *one* node.
- Shared transient objects are also stored in the transient cache on a node, but they are accessible to *all* processes on that *one* node.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

As multiple processes can access the same shared transient object at the same time, changes to shared transient objects must be performed inside transactions, which is like persistent objects, but using the **beginTransientTransaction** and **commitTransientTransaction** instructions.

```
exampleTransaction();

vars
    sharedTransientObj : SharedTransientClass;

begin
    beginTransientTransaction;
    create sharedTransientObj sharedTransient;
    commitTransientTransaction;
end;
```

Shared transient objects are an integral part of multithreading, as they allow for communication between applications within a node.

Notifications and Callbacks

When running multiple threads asynchronously, it is often a requirement to communicate progress from the asynchronous operation back to the caller, usually called a *callback*.

The callback strategy commonly used in JADE systems is to subscribe to user events on a shared transient object, pass that shared transient to the asynchronous process, and then cause events from the asynchronous process.

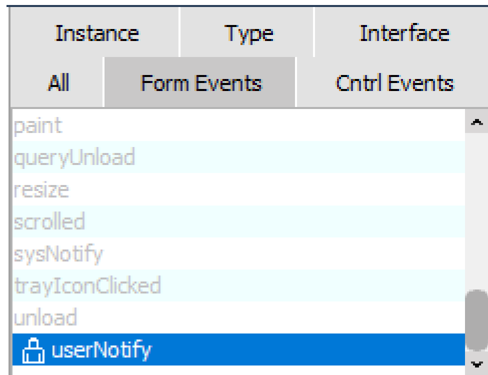
The shared transient object that is used does not need to be at all complex; in fact, no methods or properties are required on it for it to perform its role in the callback process. Once created, the initiating process should subscribe to notifications from it by using the **beginNotification** method of the **Object** class.

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

vars
    handler : CallbackHandler;

begin
    beginTransientTransaction;
    create handler sharedTransient;
    commitTransientTransaction;

    beginNotification(handler, Example_Event, Response_Continuous, 0);
end;
```

In this example, **CallbackHandler** is an empty class with no properties or methods, and **Example_Event** is a global constant of type **Integer**, with a value of **1234**.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Once the caller is subscribed to notifications by calling the **beginNotification** method, it still needs to define a behavior to perform when it is notified of that event by that object. For a form, the easiest way to do this is by adding behavior to the **userNotify** method on the **Form Events** folder of the Class Browser.



The following is an example of the code required for the **userNotify** method.

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

vars

begin
    /*
     *  The behaviour to be perfomed, this can be
     *  modifying an element on the form or bringing
     *  up a message box etc.
     */
    write eventType.String;
end;
```

To generate the event from the asynchronous process, first ensure that the process has access to the shared transient object, which typically involves passing it in by using the **startApplicationWithParameter** method when that process is first created.

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

vars
    handler : CallbackHandler;

begin
    beginTransientTransaction;
    create handler sharedTransient;
    commitTransientTransaction;

    beginNotification(handler, Example_Event, Response_Continuous, 0);

    app.startApplicationWithParameter(currentSchema.name, "CallbackApp", handler);
end;
```

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

In this example, the **CallbackApp** application has an initialize method called **createEvent**. To generate an event on the shared transient object, the **causeEvent** method of the **Object** class is used.

```
createEvent(callback : CallbackHandler);

begin
    app.doWindowEvents(1000);
    callback.causeEvent(Example_Event, true, null);
end;
```

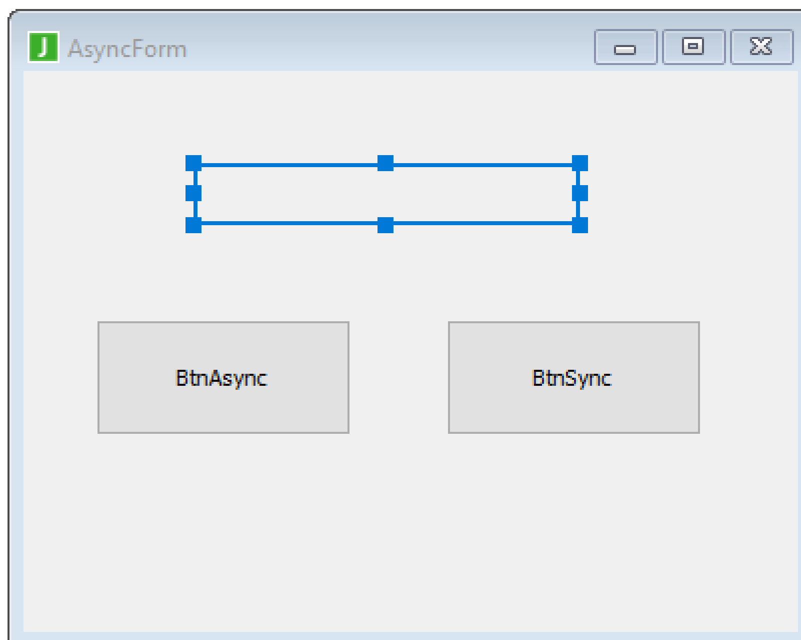
When this event is caused by the asynchronous process on the shared transient **CallbackHandler** object, the initiating process is notified of the event and the **userNotify** method is invoked.

Exercise 2 – Generating a Callback

In this exercise, you will use a shared transient object to report back as an asynchronous operation begins and finishes.

As the **BtnAsync** method creates asynchronous applications, a counter will display how many operations are pending, and that counter will decrement as they complete.

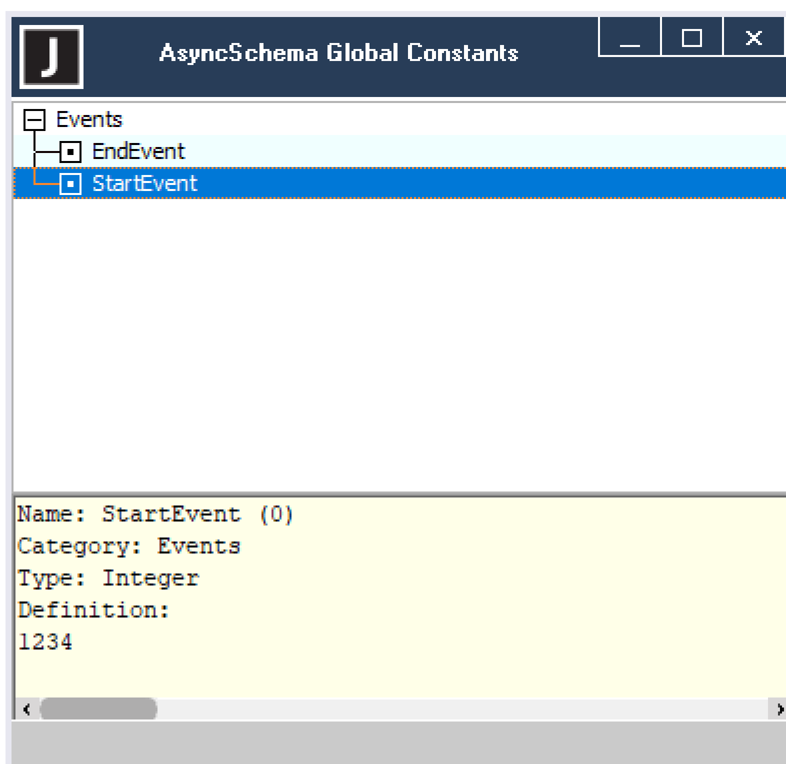
1. Modify the **AsyncForm** in the JADE Painter to add a label called **lblCallback** with an empty caption.



2. Create a class called **CallbackHandler**. You do not yet need to add any methods or properties to this class.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

3. Add two global constants (CTRL+G): **StartEvent** as an **Integer** with a value **1234** and **EndEvent** as an **Integer** with a value **1235**.



4. Create a method called **getHandler** in the **AsyncForm** class, and code it as follows.

```
getHandler() : CallbackHandler;

vars
    handler : CallbackHandler;

begin
    beginTransientTransaction;
    create handler sharedTransient;
    commitTransientTransaction;

    beginNotification(handler, StartEvent, Response_Cancel, 0);
    beginNotification(handler, EndEvent, Response_Cancel, 0);

    return handler;
end;
```

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

5. Modify the **click** method of the **btnAsync** button control as follows.

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

begin
    btnAsync.enabled := false;
    btnSync.enabled := false;

    app.startApplicationWithParameter("AsyncSchema", "AsyncApp", self.getHandler());

    btnAsync.enabled := true;
    btnSync.enabled := true;
end;
```

6. Modify the **click** method of the **btnSync** button control as follows.

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

begin
    btnAsync.enabled := false;
    btnSync.enabled := false;

    app.waitThenMsg(self.getHandler());

    btnAsync.enabled := true;
    btnSync.enabled := true;
end;
```

7. Add an **Integer** attribute property called **pendingOperations** to **AsyncForm**.

8. Modify the **userNotify** method in the **Form Events** as follows.

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

vars

begin
    if eventType = StartEvent then
        self.pendingOperations := self.pendingOperations + 1;
    elseif eventType = EndEvent then
        self.pendingOperations := self.pendingOperations - 1;
    endif;
    self.lblCallback.caption := "There are "
                            &   self.pendingOperations.String
                            &   " pending operations remaining.";
end;
```

Caution The various elements of the **AsyncForm** such as **btnAsync** and **lblCallback** also have **userNotify** methods, but it is the **userNotify** method of the form itself that we want; that is, the one found in the **Form Events** folder.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

9. Modify the **waitThenMsg** method in the **AsyncSchema** subclass of the **Application** class as follows.

```
waitThenMsg(callback : CallbackHandler);

begin
    callback.causeEvent(StartEvent, true, null);
    app.doWindowEvents(3000);
    callback.causeEvent(EndEvent, true, null);
    app.msgBox("Thanks for waiting.", "MsgBox", MsgBox_OK_Only);
end;
```

10. Run the JadeScript **createAsyncForm** method.

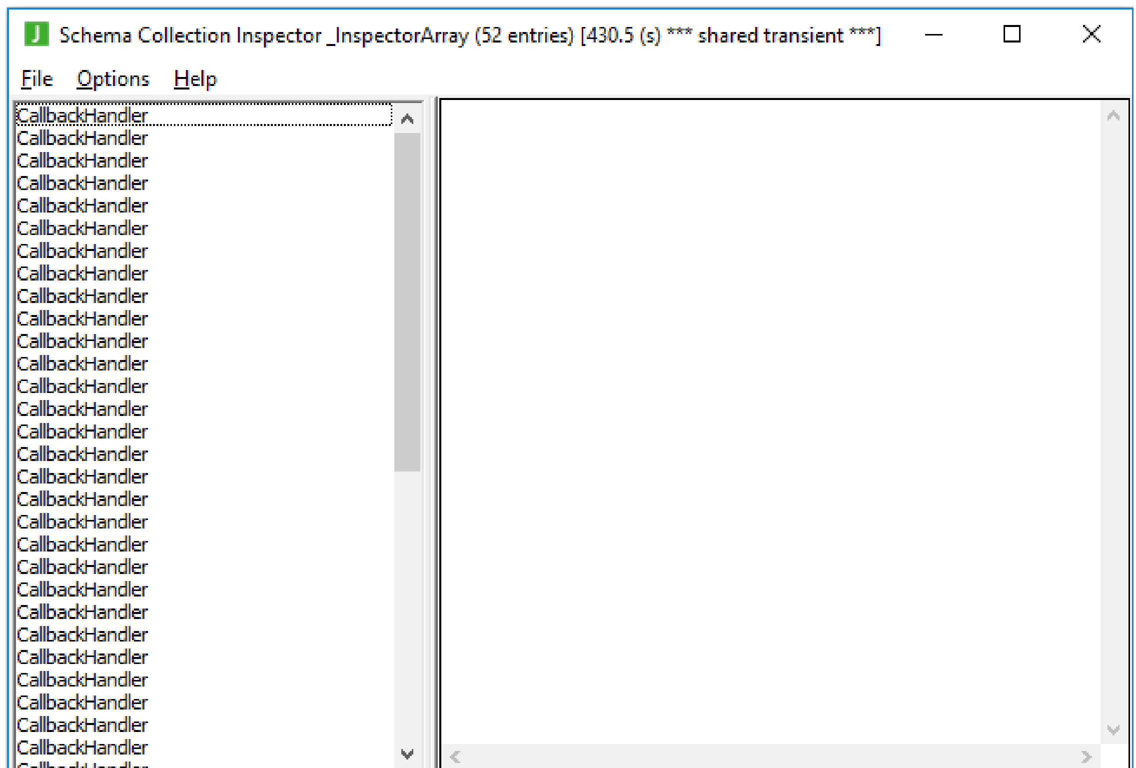
    As you click **BtnAsync**, the number of pending operations increases, and as the message boxes appear, the number falls again.

Exercise 3 – Dealing with Shared Transient Leaks

Unlike standard transient objects. which are deleted at the termination of their application, shared transient objects have a lifetime equal to that of their node so they can therefore build up if they are not deleted after use.

In this exercise, you will identify, fix, and close a shared transient leak.

1. Select **CallbackHandler** in the Class Browser and press the Ctrl+J shortcut keys to invoke the Schema Collection Inspector form.
2. You should see many shared transient instances of **CallbackHandler**, depending on how many times you have created shared transient **CallbackHandler** instances since you last restarted your application server.



 All rights reserved

3. To remove this build-up of instances of **CallbackHandler**, which is likely filling your transient cache, we could restart the application server (whether a single user JADE system, a standard client, or an application server plus presentation client). However, it is easier to create a JadeScript method to simply remove the excess transient instances manually.

   Create a JadeScript method called **deleteTransients** and code it as follows.

```
deleteTransients();

vars
    transients : ObjectArray;

begin
    create transients transient;
    CallbackHandler.allSharedTransientInstances(transients, 0, false);

    beginTransientTransaction;
    transients.purge();
    commitTransientTransaction;
end;
```

4. To prevent the shared transient instances from leaking in future, we should delete them when we are finished with them. The last time we use the callback handler is in the handling of the **EndEvent**. Modify the **userNotify** method as follows.

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

vars
    handler : Object;

begin
    if eventType = StartEvent then
        self.pendingOperations := self.pendingOperations + 1;
    elseif eventType = EndEvent then
        self.pendingOperations := self.pendingOperations - 1;
        handler := theObject;
        beginTransientTransaction;
        delete handler;
        commitTransientTransaction;
    endif;

    self.lblCallback.caption := "There are "
                            &   self.pendingOperations.String
                            &   " pending operations remaining.";
end;
```

Note In the **userNotify** method, **theObject** parameter represents the object that notified the method of the event. We cannot directly delete **theObject**, as it is a constant parameter; however, we can assign it to a new variable and delete that.

5. Run the application again (using the JadeScript **createAsyncForm** method).

   The **CallbackHandler** shared transient instances are now deleted automatically after use.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Asynchronous Method Calls

The strategy with creating a new process for each asynchronous method and using a callback to report progress, described so far in this module, is only one of two ways to implement asynchronicity in JADE. The other strategy is the use of a dedicated asynchronous worker process and using the **invoke** and **waitForMethods** methods of the **JadeMethodContext** class.

To use the **JadeMethodContext** class to call methods asynchronously, you will need to prepare the following.

1. A worker process in which **app.asyncInitialize** has been called (usually in the **initialize** method of the application; that is, the method that fulfils the **initialize** event, which is often, but not always, called **initialize**).
2. The method to be called, which does not need anything special, as in JADE, the method does not have to be marked as asynchronous; only the process that runs it. However, as it is going to be called from another process, it must not be a method of a transient object.
3. A transient instance of the **JadeMethodContext** class for the asynchronous task.
4. A transient instance of the **JadeMethodContext** class to receive the results of the asynchronous tasks.

When the **JadeMethodContext** instances have been instantiated and the worker process application has been started, the **workerAppName** property of the contexts must be set to the name of the worker process application. This allows the contexts to invoke methods on the worker processes asynchronously, using the **invoke** method. As soon as the **invoke** method has been called, the methods will run on the targeted process.

```
asyncMethodCall();

vars
    context1, context2  : JadeMethodContext;
    awaiter             : JadeMethodContext;
    worker              : Process;
    targetClass         : ExampleClass;
begin
    create context1 transient;
    create context2 transient;

    targetClass := ExampleClass.firstInstance();

    worker := app.startApplication(currentSchema.name, "WorkerApplication");

    context1.workerAppName := "WorkerApplication";
    context2.workerAppName := "WorkerApplication";

    context1.invoke(targetClass, longMethod);
    context2.invoke(targetClass, longMethod);

epilog
    delete context1;
    delete context2;
end;
```

Note The **invoke** method takes an object, a method of that object, and optionally any parameters that the method might have.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

If you require a return value from these methods, you can wait for them to finish on the caller by using **process.waitForMethods**. This method takes any number of **JadeMethodContext** instances as parameters, which can be individual object references or collections of **JadeMethodContext**. It returns the **JadeMethodContext** of the first context to finish its method, and you can then use the **getReturnValue** method of that context to obtain the return value of the called method.

```
asyncMethodCall();

vars
    context1, context2  : JadeMethodContext;
    awaiter             : JadeMethodContext;
    worker              : Process;
    targetClass         : ExampleClass;
begin
    create context1 transient;
    create context2 transient;

    targetClass := ExampleClass.firstInstance();

    worker := app.startApplication(currentSchema.name, "WorkerApplication");

    context1.workerAppName := "WorkerApplication";
    context2.workerAppName := "WorkerApplication";

    context1.invoke(targetClass, longMethod);
    context2.invoke(targetClass, longMethod);

    while true do
        awaiter := process.waitForMethods(context1, context2);
        if awaiter = null then
            break;
        else
            write awaiter.getReturnValue();
        endif;
    endwhile;

epilog
    delete context1;
    delete context2;
end;
```

Note If there are no more methods executing, the **waitForMethods** method returns null.

Exercise 4 – Invoking a Method using JadeMethodContext

In this exercise, you will use the **invoke** method of **JadeMethodContext** class to call a method asynchronously. You will perform the following actions to set up and use the **invoke** method.

1. Create a new class with a method that takes some time to execute.
2. Create a new worker application that can perform asynchronous tasks.
3. Modify the **AsyncForm** to start this worker on loading and terminate it on closing.
4. Modify the **AsyncForm** to allow for the creation of instances of the new class and the asynchronous execution of its method.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

To invoke a method using **JadeMethodContext**, perform the following steps.

1. In the **Object** class, add a subclass called **Turtle** with a method called **moveSlowly**, coded as follows.

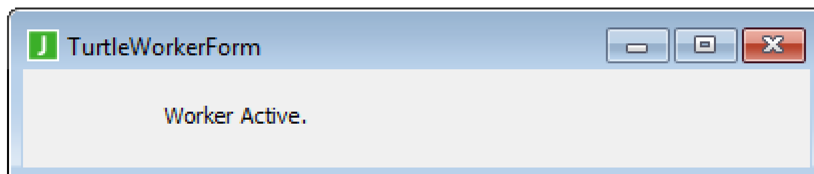
```
moveSlowly();

begin
    app.doWindowEvents(5000);
    app.msgBox("I have finally arrived...", "A Turtle's Journey", MsgBox_OK_Only);
end;
```

2. In the **AsyncSchema** subclass of the **Application** class, add the following methods.

```
turtleWorkerInit();

begin
    app.asyncInitialize();
end;
```

```
turtleWorkerFinal();

begin
    app.asyncFinalize;
end;
```

3. Open the JADE Painter and create a new form called **TurtleWorkerForm**, as follows.



Note This form does not have any usable controls, as it is used only to give a visual indicator of active workers and to provide a simple mechanism to terminate the worker (closing the window). Normally, workers would always be non-GUI applications.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

4. Open the Application Browser and create an application called **TurtleWorker**, as follows.

Define Application

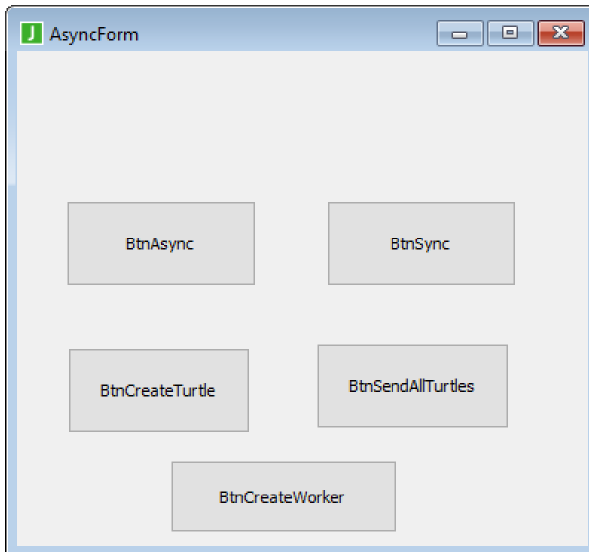
Application | Form | Web Options

Name: TurtleWorker
Help File:
Browse...
Version #:
Default Locale:
Application Type: GUI
Web Application Type: JADE Forms | HTML Documents | Web Services
Icon
Change... | Clear
Startup Form: TurtleWorkerForm
About Form:
Show Super Class Methods
Initialize Method: AsyncSchema::turtleWorkerInit
Finalize Method: AsyncSchema::turtleWorkerFinal

OK | Cancel | Help

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

5. In the JADE Painter, modify the **AsyncForm** to add three new buttons, as follows.



6. Modify the **click** method of the **btnCreateTurtle** button control, as follows.

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

vars
    babyTurtle : Turtle;
begin
    beginTransaction;
    create babyTurtle persistent;
    commitTransaction;
end;
```

7. Modify the **click** method of the **btnCreateWorker** button control, as follows.

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

vars

begin
    app.startApplication(currentSchema.name, "TurtleWorker");
end;
```

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

8. Create a method called **sendTurtle** on the **AsyncForm** class, coded as follows.

```
sendTurtle(turtle : Turtle input);

vars
    context : JadeMethodContext;
begin
    create context transient;
    context.workerAppName := "TurtleWorker";
    context.invoke(turtle, moveSlowly);
epilog
    delete context;
end;
```

9. Modify the **click** method of the **btnSendAllTurtles** button control, as follows.

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

vars
    turtle : Turtle;
begin
    foreach turtle in Turtle.instances do
        self.sendTurtle(turtle);
    endforeach;
end;
```

10. Run the form using the JadeScript **createAsyncForm** method. Try creating various numbers of turtles and workers. How long does it take for the message boxes to be displayed?

Turtles	Workers	Time Taken
1	1	
2	1	
5	1	
5	2	
10	2	
10	5	

Exercise 5 – Waiting for Asynchronous Operations

Currently, whenever the **btnSendAllTurtles** button is clicked, the **Turtle** class will pop up a message box but remain in the database. By using the **waitForMethods** method, you can have the **click** method of the **btnSendAllTurtles** button control delete the **Turtle** class from the database when it has finished its **moveSlowly** method.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

In this exercise, you will use the **Process** class **waitForMethods** method to wait for the completion of the asynchronous methods.

1. Modify the **moveSlowly** method of the **Turtle** class, as follows.

```
moveSlowly() : Turtle;

begin
    app.doWindowEvents(5000);
    app.msgBox("I have finally arrived...", "A Turtle's Journey", MsgBox_OK_Only);
    return self;
end;
```

2. Modify the **sendTurtle** method on the **AsyncForm** class, as follows

```
sendTurtle(turtle : Turtle input) : JadeMethodContext;

vars
    context : JadeMethodContext;
begin
    create context transient;
    context.workerAppName := "TurtleWorker";
    context.invoke(turtle, moveSlowly);
    return context;
end;
```

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

3. Modify the **click** method of the **BtnSendAllTurtles** button control, as follows.

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

vars
    turtle      : Turtle;
    contexts    : ObjectArray;
    context     : JadeMethodContext;
begin
    create contexts transient;
    foreach turtle in Turtle.instances do
        contexts.add(self.sendTurtle(turtle));
    endforeach;

    while true do
        context := process.waitForMethods(contexts);
        if context = null then
            break;
        else
            beginTransaction;
            delete context.getReturnValue().Turtle;
            commitTransaction;
        endif;
    endwhile;

epilog
    delete contexts;
end;
```

4. Run the form using the JadeScript **createAsyncForm** method.

   Try creating a few turtles by clicking **BtnCreateTurtle**, using **BtnSendAllTurtles**, and then inspecting the instances of **Turtle** in the Schema Collection Inspector.

   You should see that the turtle objects are deleted only after you close the message boxes.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved