Building Internet Applications with Oracle Forms 6*i* and Oracle8*i*

An Oracle Technical White Paper

March 2000

ORACLE

INTRODUCTION

You've probably heard about the wealth of new features the Oracle8*i* platform offers to turbo-charge your applications and Web solutions. But if you use Oracle Forms, you may mistakenly believe that many of these features are not available to you. This paper will describe and demonstrate how you can take advantage of some of these powerful new Oracle8*i* features in your Oracle Forms applications.

WHAT IS POSSIBLE?

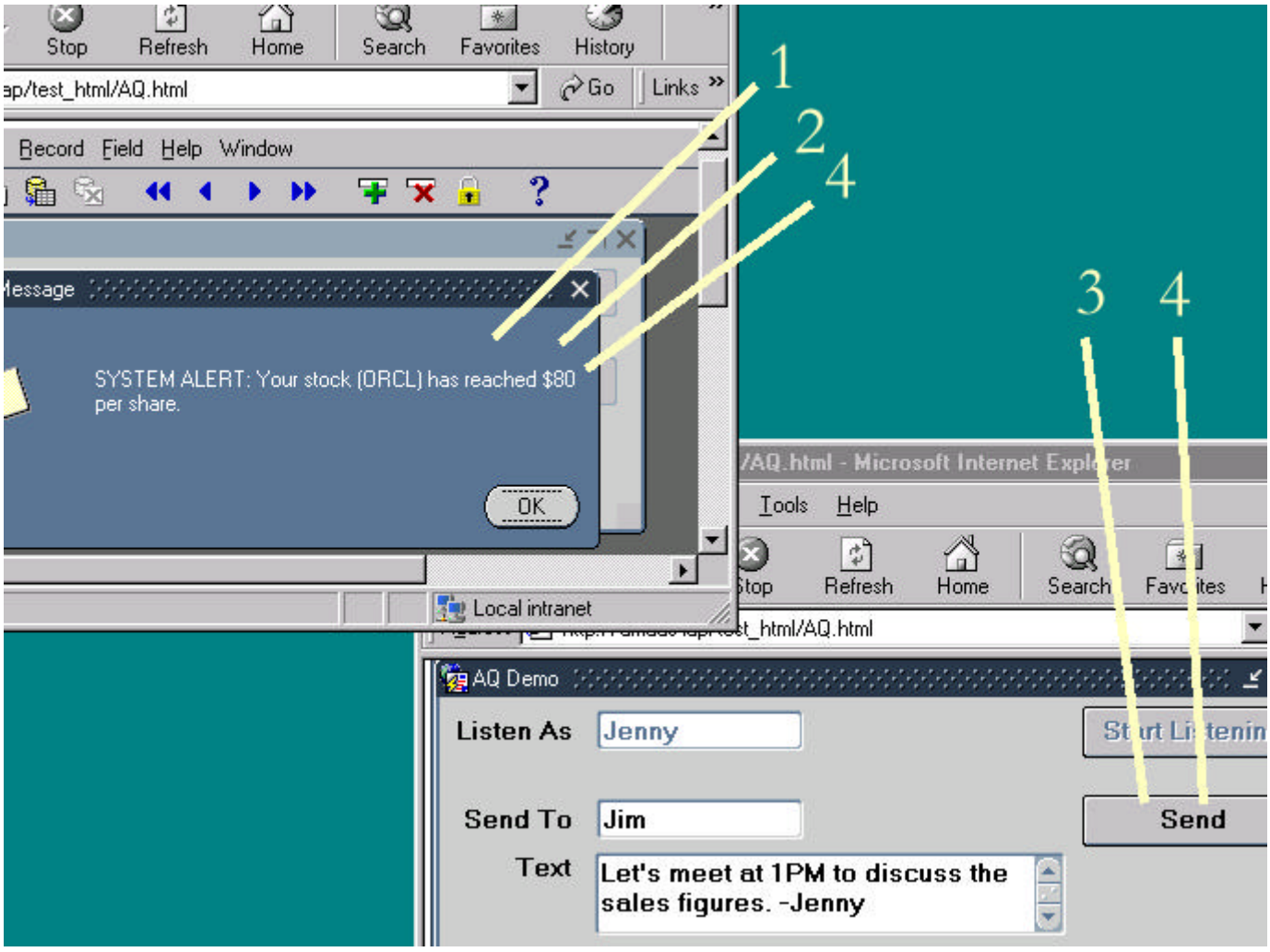

Figure 1. Oracle Forms 6*i* taking advantage of several features of Oracle8*i*.

Figure 1 illustrates Oracle Forms making use of several Oracle8*i* features. In this example, an Oracle 6*i* Form is running in two browser windows. In the upper window, a user receives an asynchronous automated alert about one of his investments. In the lower window, two users engage in a chat session using the Oracle Forms application.

WHAT TECHNOLOGIES DOES THIS SOLUTION EMPLOY?

1. The application receives messages asynchronously using the Oracle Forms 6*i* Pluggable Java Component architecture. The object receiving the messages is a JavaBean embedded in an Oracle Forms Developer Form.

2. Messages are retrieved and sent to the Pluggable Java Component through a callback from an Enterprise JavaBean that has been deployed to an Oracle8*i* database.

3. Oracle Forms sends messages by calling a PL/SQL wrapper for an Oracle8*i* Java Stored Procedure.

4. The Oracle8*i* Advanced Queuing Option is the messaging infrastructure for all messages.

REQUIREMENTS

WHAT YOU SHOULD KNOW

To get value from this paper, the following two statements should be true for you:

- You have built Forms of at least moderate complexity, in which you have added your own triggers and written your own PL/SQL to control Form behavior.

- You have at least some familiarity with Java.

However, to fully explore the technologies discussed here, it will be helpful if the following statements are also true for you:

- You have access to a working 1.1.x JDK (Java Development Kit, available from `http://java.sun.com/`).

- You know how to compile `.JAVA` files into `.CLASS` and `.JAR` files. (`javac –g [path\file.java]`)

- You know how to run Java classes from `.CLASS` and `.JAR` files. (`java [classfile.class]`)

- You have access to an `ORACLE_HOME` that contains a working installation of the Oracle8*i* database.

- You have `SYSTEM` access to an Oracle8*i* database and know how to create users, grant access to objects, and create and run stored procedures from SQL*Plus. (`create user [username] identified by [password]; grant [privilege] to [username]; create or replace procedure [prototype] as ...; exec [procedure name]([argument list]);`)

- You have access to Oracle Forms Server 6.0 or Oracle Forms Server 6*i*.

- You know how to get an Oracle Form that you have built to run in your Web browser.

SOFTWARE TO HAVE ON HAND

I developed everything in this paper using the following components:

- Oracle Forms Server Release 6.0 (Forms 6.0.5.0.2: Developer, Listener and Web Runtime)
- Oracle8*i* Enterprise Edition Release 8.1.5 for Windows NT
- Oracle SQL*Plus Release 8.1.5.0.0
- Sun š JDK Version 1.1.8 for Windows NT
- Microsoft Peer Web Server 3.0 (provided with Windows NT, go to Control Panel, Network, Services, Add...)
- Microsoft Internet Explorer 5.0 and the Microsoft VM.
- Microsoft Windows NT Workstation 4.0 Service Pack 3
- Helios Textpad 4.0 (http://www.textpad.com)

PART 1 OF 3: JAVA STORED PROCEDURES

Oracle Forms Program Units have long been able to call PL/SQL functions and procedures stored in an Oracle database. So the easiest way to begin using Java in your Oracle Forms applications is to put a PL/SQL wrapper around an Oracle8*i* Java Stored Procedure.

A SIMPLE EXAMPLE

Here is an example of a very simple Java class that adds two numbers together and returns a result.

```
//Simple.java
public class Simple
{
  public static int add(int first, int second)
  {
```

```
        return first + second;
      }
    }
```

This class (Simple) contains one method (add). This method accepts two integer arguments and returns an integer that is the sum of these arguments. If you've installed a JDK, you can easily follow along with this example. Using a text editor of your choice, such as `NOTEPAD.EXE`, type or copy and paste this example. If you type it, do so very carefully, as Java is case-sensitive. Save the file as `Simple.java`.

Now you can compile and run this example. To do so, open a command prompt window. From the command prompt window, you must set two environment variables before you can compile or run Java classes. First, the `PATH` environment variable must include your `[JDK]\bin` directory. Second, there must be an environment variable called `CLASSPATH` that is equal to your `[JDK]\lib\classes.zip`. You can check and set these environment variables two ways:

- From your command prompt window, issue the `SET PATH` and `SET CLASSPATH` commands respectively. This is the method we recommend.

OR

- Close your command prompt window; and set the environment variables from Control Panel -> System -> Environment. Then open a new command prompt window. Be aware, however, that setting the `CLASSPATH` environment variable from the Control Panel can prevent Oracle Forms Developer applications from running properly in a Web browser.

Once you have set these environment variables and you are in a command prompt window, change your directory to the one where you saved Simple.java. From this directory, issue a `javac –g Simple.java` command to compile your class. If you have entered it correctly and set the variables correctly, the command prompt window should return quickly to a command prompt with no messages displayed. If a message appears, then an error has occurred.

If all has proceeded well, a new file, `Simple.class`, should appear in the current directory. This file is the compiled version of your class.

In many situations, the next step might be to run the class from the command line using the `java [file.class]` command. However, the Simple class is so simple that it would produce no visible result. Instead, the next step is to load the class into your Oracle8*i* database. If your Oracle8*i* database is not started, start it now. Next, make certain that the `[ORACLE_HOME]\bin` directory for your Oracle8*i* database is still included in your `PATH` environment variable. Finally, issue a `loadjava –user scott/tiger Simple.class` command from the directory where you compiled your class. The `loadjava` utility is provided with Oracle8*i* for importing class files into the database. If your Oracle8*i* database is functioning properly, the command prompt window should return quickly to a command prompt with no messages displayed. If a message appears, then an error has occurred.

Now you may start SQL*Plus and log in to your Oracle8*i* database as `SCOTT/TIGER`. From your SQL*Plus window issue a `SELECT object_name, object_type FROM user_objects WHERE object_type = 'JAVA CLASS';` query. Your query should produce a result similar to the following:

```
OBJECT_NAME         OBJECT_TYPE
------------------- ------------------
Simple              JAVA CLASS
```

What Just Happened?

Oracle8*i* has the capability to assimilate certain Java classes directly into the database when you use the `loadjava` command. You have just loaded a Java class into your database!

So What?

You may be wondering what's useful about this since you have yet to run your class. So let's demonstrate Oracle8*i*'s ability to run Java next.

You must next create a PL/SQL wrapper for your class to make it accessible via PL/SQL and SQL*Plus. To do so, type or copy and paste the following command into your SQL*Plus window:

```
CREATE OR REPLACE FUNCTION simple_wrapper(first NUMBER, second NUMBER) RETURN
NUMBER AS

LANGUAGE JAVA NAME 'Simple.add(int,int) return int';

/
```

This PL/SQL wrapper function is know as a *Java Call Spec*. Notice that the parameters for the wrapper function are of the same number and type as the parameters for the Java method. Oracle8*i* will translate argument values given to the wrapper function and pass them along as arguments to the Java object.

After you have issued the `CREATE OR REPLACE FUNCTION` command, SQL*Plus should respond with a `Function created` statement. If SQL*Plus responds with the statement: `Warning: Function created with compilation errors`, then an error has occurred.

If everything has run smoothly, and you have not ended up with any error statements, you are ready to execute your Java class. To execute your Java class, issue the following commands in your SQL*Plus window:

```
VARIABLE result NUMBER;
CALL simple_wrapper(3, 4) INTO :result;
PRINT result;
```

SQL*Plus should respond with the following:

```
RESULT
----------
         7
```

If you get this result, you have successfully created a Java class, compiled it, loaded it into an Oracle8*i* database, written a Call Spec for it, and executed it through the Call Spec. Congratulations!

CALLING SIMPLE FROM ORACLE FORMS DEVELOPER

Now it's easy to use your Java Stored Procedure from an Oracle Forms Developer application. Follow these steps if you are unsure how to do this:

> **Note:** *For this exercise to be successful, your Oracle8*i *database and its listener must be running. Additionally, you must have previously created a Connect Identifier for your Oracle8*i *database in your* `TNSNAMES.ORA` *file for Oracle Forms Developer.*

1. Launch Forms Developer.
2. If prompted, either choose to Build a New Form manually or cancel the prompt.
3. In the Object Navigator window, open the Canvases node.
4. Double-click on `CANVAS2`. The Layout Editor window should appear.
5. Using the Layout Editor Item Tool Palette, select the Text Item tool and add three new Text Items to the Canvas.
6. Select the Button Item tool, and add a new Button to the Canvas.
7. Select all three Text Items, and press F4 to invoke the Property Palette.

8. From the Property Palette, set the Data Type property for these Items to Number.

9. Right-click on the Button Item from either the Layout Editor window or the Object Navigator window and select the SmartTriggers popup menu option.

10. From the SmartTriggers runoff menu, choose the `WHEN-BUTTON-PRESSED` trigger. The PL/SQL Editor window should appear.

11. In this PL/SQL Editor window, type:
    ```
    :text_item6 := simple_wrapper(:text_item4, :text_item5);
    ```
    If your Text Item names vary from these, adjust the entered PL/SQL appropriately.

12. To Compile your Form, choose the File -> Administration -> Compile File menu option, or press Ctrl-T.

13. When prompted to log on, choose Yes.

14. Log on as `SCOTT/TIGER@[Your Oracle8i Database Connect Identifier]`.
    After completing these steps, your Form module should be built successfully.

15. Run your Form. You should be able to enter numeric values in the first two Text Items, press the Button, and have the added result appear in the third Text Item.

Congratulations! You are now using your Java Stored Procedure from Oracle Forms Developer!

AQSERVERSEND: A MORE COMPLICATED EXAMPLE

Our next example of a Java Stored Procedure is the AQServerSend class. This is a class that allows addressed text messages to be deposited in an Oracle8*i* Advanced Queue for later retrieval. It demonstrates how Java in the database can be used to perform more sophisticated tasks. It also showcases Java's robust datatype conversion capabilities by accepting message text as a VARCHAR2 argument and then Enqueuing it into an Advanced Queue of type RAW, a conversion that can prove challenging in PL/SQL.

At this point, you should be familiar with the requirements and processes for using Oracle8*i* to work with Java, so I will provide progressively fewer instructions.

Setup Steps

1. To prevent the `SCOTT` schema from becoming too cluttered, put the Advanced Queuing components in a fresh schema.

2. Create the recommended schema:

   - Log in to your Oracle8*i* database from SQL*Plus using the `SYS` account and issue the following commands:

   - `DROP USER aqdemo CASCADE;`

- `GRANT CONNECT, RESOURCE TO aqdemo IDENTIFIED BY aqdemo;`
- `GRANT EXECUTE ON sys.dbms_aqadm TO aqdemo;`
- `GRANT EXECUTE ON sys.dbms_aq TO aqdemo;`
- `GRANT EXECUTE ON sys.dbms_aqin TO aqdemo;`

3. Create and start an Advanced Queuing Queue Table and Queue in the new schema.
   - Log in to SQL*Plus for your Oracle8*i* database using the new `aqdemo` account. You can do this directly from the previous step (2.E.) by typing `CONNECT aqdemo/aqdemo`. Then issue the following commands:
   - `EXEC dbms_aqadm.create_queue_table('demo_queue_table', 'RAW');`
   - `EXEC dbms_aqadm.create_queue('demo_queue', 'demo_queue_table');`
   - `EXEC dbms_aqadm.start_queue('demo_queue');`
4. Expand your `CLASSPATH` environment variable.
   - Open a command prompt window from which you can compile and load Java and issue the following command:
   - `SET CLASSPATH=%CLASSPATH%;[ORACLE8i_HOME]\rdbms\jlib\aqapi.jar;[ORACLE8i_HOME]\jdbc\lib\classes102.zip`
5. Load the Advanced Queuing classes into the new schema:
   - From a command prompt window, issue the following command:
   - `loadjava –user aqdemo/aqdemo [ORACLE8i_HOME]\rdbms\jlib\aqapi.jar`

If you completed the preceding steps without error statements, you have successfully created and started an Oracle8*i* Advanced Queue named `demo_queue` in a Queue Table named `demo_queue_table`.

AQServerSend

The code for `AQServerSend` is well documented. Comment lines begin with a double forward slash (`//`). If you wish to make use of it, you must compile it with the `javac` command and load it with the `loadjava` command. Refer to the section entitled A Simple Example for instructions on these commands.

```
//AQServerSend.java
```

```
//This is a Java Stored Procedure to add new messages
//to an Oracle8i Advanced Queue. It has one method:
//sendMessage. When called, this method connects to an
//Oracle8i Advanced Queue and  Enqueues the supplied
//message to the supplied recipient. Note that because
//no transaction is created or committed by this class,
//the caller must explicitly commit the  sendMessage
//after it completes.

import java.sql.*;
import oracle.AQ.*;

public class AQServerSend
{
  public static void sendMessage(String addressee, String payload)
  {
    Connection              myConnection;
    //A bunch of Oracle8i Advanced  Queueing objects.
    AQSession               myAQSession          = null;
    AQQueueTable            myAQQueueTable;
    AQQueue                 myAQQueue;
    AQEnqueueOption         myAQEnqueueOption    = new AQEnqueueOption();
    AQMessage               myAQMessage;
    AQMessageProperty       myAQMessageProperty  = new AQMessageProperty();
```

```
AQRawPayload             myAQRawPayload;
//The message will need to be translated to bytes from a string.
byte[]                   payloadBytes;

try
{
  if(addressee == null || payload == null)
    return;

  //We need to connect to the Oracle8i database before
  //we send the message. Fortunately, since the SP is inside
  //the database, a method is provided by which we can
  //implicitly connect to the database we're in.
  myConnection = new oracle.jdbc.driver.OracleDriver().defaultConnection();

  //Load the Oracle8i AQ driver.
  Class.forName("oracle.AQ.AQOracleDriver");

  //Each Oracle8i Advanced Queue is stored in Queue Table,
  //which is a normal database table. Each Queue Table may
  //contain one or more Queues. Here we use the implicit
  //database connection that we retrieved above to create a
  //new AQ Session and get handles to our Queue Table and Queue.
  //aqdemo is the user schema in the Oracle8i database in
```

```
  //which our Queue Table resides. If you chose to use a
  //different schema, Queue Table name or Queue name, you
  //would need to change the values below.
  myAQSession = AQDriverManager.createAQSession( myConnection);
  myAQQueueTable = myAQSession.getQueueTable("aqdemo", "demo_queue_table");
  myAQQueue      = myAQSession.getQueue("aqdemo", "demo_queue");

  //Create a message.
  myAQMessage = myAQQueue.createMessage();
  //Convert the payload string to bytes.
  payloadBytes = payload.getBytes();
  //Populate the message with the text bytes.
  myAQRawPayload = myAQMessage.getRawPayload();
  myAQRawPayload.setStream( payloadBytes, payloadBytes.length);

  //Address the message.
  myAQMessageProperty.setCorrelation(addressee);
  myAQMessage.setMessageProperty( myAQMessageProperty);

  //Enqueue (send) the message.
  myAQQueue.enqueue( myAQEnqueueOption, myAQMessage);
}
catch(Exception myException)
{
```

```
        System.out.println(myException);
      }
      return;
    }
  }
```

Once you have loaded the `AQServerSend` class, you must create a Call Spec so that it can be accessed from PL/SQL. Below is a Call Spec for the `AQServerSend` class:

```
CREATE OR REPLACE PROCEDURE aqserversend_wrapper(addressee VARCHAR2, payload
VARCHAR2) AS
  LANGUAGE JAVA NAME 'AQServerSend.sendMessage(java.lang.String,
java.lang.String)';
/
```

Once you have compiled and loaded the class and Call Spec without errors, you can put new messages in the Queue from SQL*Plus as follows:

```
EXEC aqserversend_wrapper('Jim', 'What are you doing? –Phil');
COMMIT;
```

Note that you must manually commit the transaction after performing a send, since `AQServerSend` does not do this for you.

To check if your send operation has successfully deposited a message in the queue, you can issue the following query from SQL*Plus:

```
SELECT corrid FROM demo_queue_table;
```

This query will display a list of addressees for whom there are currently undelivered messages in your Queue Table. You can clear a message from your Queue Table simply by deleting its row.

Build A Send Form

As with `simple_wrapper()` in the previous example, `aqserversend_wrapper()` may now be called from a Form. You can take the first steps toward the completed messaging application by creating a Form with two Text Items and a Button Item. A `WHEN-BUTTON-PRESSED` trigger for the Button Item could call `aqserver_wrapper()` using the values of the Text Items as the arguments to the procedure. Once completed, this allows users to deposit messages from Oracle Forms Developer to an Oracle8*i* Advanced Queue.

CONCLUSION OF PART 1

You ll find more information about and examples of Java Stored Procedures in *Oracle8i Java Developer s Guide* and *Oracle8i Java Stored Procedure Developer s Guide*. You can get further information about the Oracle8*i* Advanced Queuing Option in *Oracle8i Application Developer s Guide – Advanced Queuing*. Oracle supplies all three of these volumes with your Oracle8*i* database in both HTML and PDF formats.

PART 2 OF 3: ENTERPRISE JAVABEAN

Oracle8*i* and Oracle Forms' support of Java does not end with Java Stored Procedures. You can also access Enterprise JavaBeans in your Oracle Forms and Oracle8*i* applications.

WHAT ARE ENTERPRISE JAVABEANS?

Enterprise JavaBeans (EJB) is a specification from Sun that allows Java backend logic to be encapsulated and distributed across many servers in an N-Tiered application. In most systems, it is more efficient for certain logic to be hosted near the data, and certain logic to reside nearer the user or client. Enterprise JavaBeans provide a mechanism that allows developers to distribute this logic. Unlike regular JavaBeans, Enterprise JavaBeans do not perform user interface functions, such as receiving user input or implementing GUI widgets for display. Instead, they typically perform backend functions, such as managing data and transactions.

Communications to and among Enterprise JavaBeans are mediated by an Object Request Broker (ORB). Oracle8*i* provides an ORB and a container environment for deploying Enterprise JavaBeans. The `listener.ora` file provided in your `[Oracle8i_HOME]\network\admin` directory defines the port on which the Oracle8*i* ORB will listen for requests. This defaults to port 2481.

WHERE CAN WE USE AN ENTERPRISE JAVABEAN?

In the last part, you created a Java Stored Procedure that allowed you to add messages to an Oracle8*i* Advanced Queue. But you still have no way to receive these messages. Fortunately, the task of locating and extracting messages from a server queue is ideally suited to an Enterprise JavaBean.

But where will the messages go once the EJB has retrieved them? Since an EJB does not have a user interface, a client-side Bean must deliver the messages to the user. The EJB specification provides a means to look up and call an EJB. But what if you do not want the client Bean to continuously call the server EJB to poll for new messages?

An analogy for the desired roles of the client Bean and the EJB is to think of the client Bean as a busy manager and the EJB as a helpful administrative assistant. The busy manager will probably not have time to keep calling the administrative assistant to ask if there are any new messages. It would be more efficient for the busy manager to tell the administrative assistant to start listening for messages and call back when a message arrives.

The EJB architecture provides such a mechanism; it is known as a *callback*. When you construct your client Bean to perform a callback, it does what the busy manager would do. The client bean tells the EJB on the server to start listening for messages and gives the EJB a callback handle. This callback handle is like a phone number that the EJB then uses to call the client Bean

whenever it receives a message for it. The client Bean is free to continue processing without any further polling of the EJB on the server.

The only drawback to using callback is that it introduces a bit of complexity on the client side.

THE THREE PARTS OF AN ENTERPRISE JAVABEAN

To provide security and transaction management, the Enterprise JavaBeans specification requires insulation around an EJB. This insulation comes in the form of two interfaces that are required for each EJB: the Home interface and the Remote interface. The three parts of an EJB are:

- The Home interface, which specifies the mechanism that creates new instances of the EJB.
- The Remote interface, which specifies the methods that the EJB exposes to remote calls.
- The Bean itself, which is where most of the logic of the EJB resides.

In this example the three parts are filled by:

- `AQServerReceiveHome`, the Home interface
- `AQServerReceiveRemote`, the Remote interface
- `AQServerReceiveBean`, the Bean class

Setup 1 of 2: Create the Package Directories

If you are following along with the examples, you should create a directory with some subdirectories for your `.JAVA` and `.CLASS` files before you proceed. This is because the remaining Java source files are part of *Java packages*. A Java package is a way of grouping together Java classes that belong together. It ensures that all the classes in the package will have visibility to one another. Java classes that are part of a package must be put in a subdirectory (or series of nested subdirectories) that matches their package name. Once you have created the subdirectories for packages used in the example classes, you should issue commands from the example parent directory. Here is the directory structure and the list of files that should be put in each:

- [Your parent directory for the examples, recommended name: AQ] (Run commands from here.)
- `AQServerSend` (should contain)
    - Simple.java
    - Simple.class

    - AQServerSend.java
    - AQServerSend.class
- `AQServerReceive` (also add to your `CLASSPATH`, will contain)
    - `AQServerReceiveHome.java`
    - `AQServerReceiveHome.class`
    - `AQServerReceiveRemote.java`
    - `AQServerReceiveRemote.class`
    - `AQServerReceiveBean.java`
    - `AQServerReceiveBean.class`
- `AQClientReceive` (also add to your `CLASSPATH`, will contain)
    - `AQCallback.java`
    - `AQCallback.class`
    - `AQCallbackStub.java`
    - `AQCallbackStub.class`
    - `AQClientReceiveListener.java`
    - `AQClientReceiveListener.class`
    - `AQClientReceiveBean.java`
    - `AQClientReceiveBean.class`
    - `AQClientReceiveWrapper.java`
    - `AQClientReceiveWrapper.class`
    - `AQCallbackHelper.java`
    - `AQCallbackHelper.class`
    - `AQCallbackHolder.java`
    - `AQCallbackHolder.class`

- AQCallbackOperations.java
- _AQCallbackImplBase.java
- _AQCallbackImplBase.class
- _example_AQCallback.java
- _st_AQCallback.java
- _st_AQCallback.class
- _tie_AQCallback.java

Setup 2 of 2: Expand your CLASSPATH

Expand your CLASSPATH environment variable from its value after completing part 1, Setup Steps, step 4, to include the following files: [ORACLE8i_HOME]\lib\aurora_client.jar;[ORACLE8i_HOME]\jdbc\lib\classes111.zip; [ORACLE8i_HOME]\sqlj\lib\translator.zip;[ORACLE8i_HOME]\ lib\vbjorb.jar; [ORACLE8i_HOME]\lib\vbjapp.jar;[ORANT_HOME]\forms60\java\f60all.jar

AQServerReceiveHome

Below is the source for the AQServerReceiveHome interface of the AQServerReceive EJB. It is an example Home interface for an EJB.

```
//AQServerReceiveHome.java


//Every EJB uses two helper-interfaces:
//A Home interface, which creates new instances
//                  of the EJB on demand,
//and a Remote interface, which exposes methods
//                        on the EJB to any clients
//                        who call it.
//This is the Home interface for the EJB.
```

```
//See also: AQServerReceiveRemote.java, AQServerReceiveBean.java,
//and AQClientReceiveBean.java

package AQServerReceive;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface AQServerReceiveHome
       extends EJBHome
{
  public AQServerReceiveRemote create()
    throws RemoteException, CreateException;
}
```

AQServerReceiveRemote

Following is the source for the `AQServerReceiveRemote` interface for the `AQServerReceive` EJB. It is an example Remote interface for an EJB.

```
//AQServerReceiveRemote.java

//Every EJB uses two helper-interfaces:
//A Home interface, which creates new instances
//                  of the EJB on demand,
```

```
//and a Remote interface, which exposes methods
//                       on the EJB to any clients
//                       who call it.
//This is the Remote interface for the EJB.
//See also: AQServerReceiveHome.java, AQServerReceiveBean.java,
//and AQClientReceiveBean.java

package AQServerReceive;

import AQClientReceive.AQCallback;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.sql.SQLException;

public interface AQServerReceiveRemote
       extends EJBObject
{
  public void waitForMessage(AQCallback callbackClient, String messageRecipient)
    throws RemoteException, SQLException, ClassNotFoundException;
}
```

AQServerReceiveBean

Last is the Bean itself. Below is the source for `AQServerReceiveBean` class. It is an example Bean class for an EJB.

Note: *If you are compiling the source files as you proceed, you should be aware that* `AQServerReceiveBean` *has a dependency on one client class:* `AQCallback`. `AQCallback` *is the "phone number" that* `AQServerReceiveBean` *uses to call back to the client when a new message arrives, so that the client Bean does not have to continuously poll it for new messages. (See the section entitled Where can we use an Enterprise JavaBean?) If you wish to compile* `AQServerReceiveBean`, *you will first have to create, compile, and make* `AQCallback` *available from your* `CLASSPATH`. *Notice that once called, the* `waitForMessage()` *method of the* `AQServerReceiveBean` *will run forever unless it receives a message with a content of* `!!!STOP!!!`. *The source for* `AQCallback` *can be located in the section entitled The Client JavaBean below.*

```
//AQServerReceiveBean.java


//This is the server EJB.

//It has one important method: waitForMessage.

//When called, this method connects to an Oracle8i

//Advanced Queue and begins a loop where it  Dequeues

//messages for the caller and calls back down with

//these messages.

//See also: AQClientReceiveBean.java, AQCallback.java,

//AQServerReceiveRemote.java, and AQServerReceiveHome.java


package AQServerReceive;


import AQClientReceive.AQCallback;


import javax.ejb.*;

import java.rmi.RemoteException;

import java.sql.*;

import oracle.AQ.*;
```

```
public class AQServerReceiveBean
       implements SessionBean
{
  private SessionContext mySessionContext;

  //Here is where a call that began from  AQClientReceiveWrapper or
  //AQClientReceiveBean finally ends up. We will need to be able to
  //call back down to the client (through  AQCallback) and we need
  //to know who to get messages for ( messageRecipient).
  public void waitForMessage(AQCallback callbackClient, String messageRecipient)
    throws RemoteException, SQLException, ClassNotFoundException
  {
    Connection              myConnection;
    //A bunch of Oracle8i Advanced  Queueing objects.
    AQSession               myAQSession          = null;
    AQQueueTable            myAQQueueTable;
    AQQueue                 myAQQueue;
    AQDequeueOption         myAQDequeueOption    = new AQDequeueOption();
    AQMessage               myAQMessage;
    AQMessageProperty       myAQMessageProperty;
    AQRawPayload            myAQRawPayload;
    //What message should cause us to stop listening for messages.
    final String            stopCondition        = "!!!STOP!!!";
```

```
//The message will arrive as bytes and be translated to a string.
byte[]                   payloadBytes;
String                   payloadString         = null;

if(messageRecipient == null)
  return;

//We need to connect to the Oracle8i database before
//we begin looking for messages. Fortunately, since
//the EJB is inside the database, a method is provided
//by which we can implicitly connect to the database
//we're in.
myConnection = new oracle.jdbc.driver.OracleDriver().defaultConnection();

//Load the Oracle8i AQ driver.
Class.forName("oracle.AQ.AQOracleDriver");

//Start a transaction to record our database operations
//so that we may later perform a commit or rollback on
//them.
mySessionContext.getUserTransaction().begin();

//Each Oracle8i Advanced Queue is stored in Queue Table,
//which is a normal database table. Each Queue Table may
```

```
//contain one or more Queues. Here we use the implicit
//database connection that we retrieved above to create a
//new AQ Session and get handles to our Queue Table and Queue.
//aqdemo is the user schema in the Oracle8i database in
//which our Queue Table resides. If you chose to use a
//different schema, Queue Table name or Queue name, you
//would need to change the values below.
myAQSession = AQDriverManager.createAQSession( myConnection);
myAQQueueTable = myAQSession.getQueueTable("aqdemo", "demo_queue_table");
myAQQueue      = myAQSession.getQueue("aqdemo", "demo_queue");

//Get the Dequeue Options ready for our  Dequeue below.
//A set of  Dequeue Options determines how a  Dequeue operation
//is performed. In this case, we want any messages we  Dequeue
//to be removed from the Queue. We also want to  Dequeue
//only messages that are addressed to our caller. Third,
//we want our  Dequeue operation to wait an unlimited amount
//of time for a message that is addressed to our caller.
myAQDequeueOption.setDequeueMode( AQDequeueOption.DEQUEUE_REMOVE);
myAQDequeueOption.setCorrelation( messageRecipient);
myAQDequeueOption.setWaitTime( AQDequeueOption.WAIT_FOREVER);

//Commit any database operations in our transaction
//so far.
```

```
    mySessionContext.getUserTransaction().commit();

    //Continue to pass through this loop,  Dequeuing messages
    //addressed to our caller, until we receive a stop message.
    do
    {
      //Start a new transaction.
      mySessionContext.getUserTransaction().begin();
      //Dequeue the next message addressed to our caller.
      myAQMessage = myAQQueue.dequeue(myAQDequeueOption);
      //Extract the payload of the message.
      myAQRawPayload = myAQMessage.getRawPayload();
      //Get the bytes that make up this payload.
      payloadBytes = myAQRawPayload.getBytes();
      //Convert the bytes of the payload to a string.
      payloadString = new String(payloadBytes);
      //Commit any database operations in this transaction.
      mySessionContext.getUserTransaction().commit();
      //Call back to the client and deliver the payload
      //of the message in string format.
      callbackClient.receiveMessage(payloadString);
    }
    while(!payloadString.equals(stopCondition));
  }
```

```
    //Some other methods required in a SessionBean.
    public void ejbCreate () throws RemoteException, CreateException {}
    public void ejbRemove() {}

    public void setSessionContext (SessionContext newSessionContext)
    {
      this.mySessionContext = newSessionContext;
    }

    public void ejbActivate () {}
    public void ejbPassivate () {}
  }
```

You may observe that `AQServerReceive` bears some resemblance to the Java Stored Procedure `AQServerSend` that you created earlier. This is because both classes perform similar but inverted functions. `AQServerSend` enqueues messages into the Queue and `AQServerReceiveBean` dequeues those messages from the Queue.

THE CLIENT JAVABEAN

In the example, the client JavaBean is slightly more complicated than the EJB, primarily because the definition of the callback resides in the client. A client Bean could consist of only one class, but in the example it consists of the following four interfaces and classes (for now):

- `AQCallback` is the parent interface for the callback.
- `AQCallbackStub` is the class, derived from `AQCallback`, that actually receives the callback from `AQServerReceiveBean`.
- `AQClientReceiveListener` will be required to communicate with Oracle Forms Server in the next part. It is used by `AQClientReceiveBean`, but don't worry about it for now.
- `AQClientReceiveBean` is the main logic for the client JavaBean.

AQCallback

`AQCallback` is an example parent interface for a callback. If you look ahead to `AQCallbackStub`, you see that `AQCallbackStub` does not extend `AQCallback` directly. Instead, `AQCallbackStub` extends a class called `_AQCallbackImplBase`. Since I haven't described `_AQCallbackImplBase` as part of the client, you may wonder where it came from and when it will appear. The answer is that `_AQCallbackImplBase.java` and the source for six other `.JAVA` files are automatically generated by a utility you run on the `AQCallback` class. The utility is `java2rmi_iiop`. It accepts as its argument the name of a class for which it should generate Java RMI (Remote Method Invocation) helper classes. If you completed the package subdirectory setup steps earlier, and you have successfully compiled `AQCallback`, issue the `java2rmi_iiop AQClientReceive.AQCallback` command from the example parent directory. This command may take several seconds to complete. It should generate the following new files in the `AQClientReceive` directory:

- AQCallbackHolder.java
- AQCallbackHelper.java
- _st_AQCallback.java
- _AQCallbackImplBase.java
- AQCallbackOperations.java
- _tie_AQCallback.java
- _example_AQCallback.java

These files include source files for the classes that make the callback from the server EJB to the client JavaBean work. You may examine them, but further explanation of callback is beyond the scope of this paper.

Below is the source for `AQCallback`:

```
//AQCallback.java


//This is the EJB callback parent interface.
//It is used to build the EJB callback classes
//when the following command is issued:
```

```
//java2rmi_iiop AQClientReceive.AQCallback
//These classes will be used to allow the server
//EJB to call back down to the client each time
//a new message arrives for it.
//See also: AQCallbackStub.java

package AQClientReceive;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface AQCallback
       extends Remote
{
  //Here the AQClientReceive interface defines the
  //method by which callbacks to the client bean
  //will occur in any callback stubs derived from it.
  public void receiveMessage(String payload)
    throws RemoteException;
}
```

AQCallbackStub

`AQCallbackStub` is an example callback class. It is derived, using the `java2rmi_iiop` utility, from the `AQCallback` interface. `AQCallbackStub` is instantiated by `AQClientReceiveBean` and passed to `AQServerReceiveBean`. When

AQServerReceiveBean pulls a message out of the Queue, it calls AQCallbackStub. AQCallbackStub in turn calls AQClientReceiveBean. Following is the source for AQCallbackStub.

```
//AQCallbackStub.java

//This is the EJB callback class. It is like a phone number
//by which the server EJB can call the client back when
//something that we care about happens. It is derived from
//AQCallback by way of the java2rmi_iiop command.
//_AQCallbackImplBase is one of the classes generated by the
//java2rmi_iiop utility. AQCallbackStub is instantiated
//by AQClientReceiveBean and passed to the server EJB so that
//the server can call back down to AQClientBean each time a new
//message arrives for it.
//See also: AQCallbackStub.java and AQClientReceiveBean.java

package AQClientReceive;

import java.rmi.RemoteException;
import org.omg.CORBA.Object;

import oracle.aurora.AuroraServices.ActivatableObject;

public class AQCallbackStub
       extends _AQCallbackImplBase
```

```
        implements ActivatableObject
{
  private AQClientReceiveBean parentAQClientReceiveBean;

  //Constructor. Receives a handle to its calling object:
  //AQClientReceiveBean so that it can let  AQClientReceiveBean
  //know each time a new message arrives from the server EJB for it.
  public AQCallbackStub(AQClientReceiveBean parentAQClientReceiveBean)
  {
    this.parentAQClientReceiveBean =  parentAQClientReceiveBean;
  }

  //Here we implement the required  AQCallback method, which
  //will be the initial return point for any callbacks from
  //the server EJB. It simply passes the message back into
  //the AQClientReceiveBean object that started it.
  public void receiveMessage(String messageText)
    throws RemoteException
  {
    parentAQClientReceiveBean.receiveMessage( messageText);
  }

  //Required for EJB callbacks.
  public Object _initializeAuroraObject () {
```

```
    return this;
  }
}
```

AQClientReceiveListener

`AQClientReceiveListener` is an example of a listener interface that can be used by one Java object to communicate with another Java object. It is required to communicate with Oracle Forms Server in the next part. It is used by `AQClientReceiveBean`, but you don't worry about it for now.

```
//AQClientReceiveListener.java


//This is the interface implemented by
//AQClientReceiveWrapper that allows it to receive
//messages from AQClientReceiveBean so it can
//pass those messages back to the Form. It is used
//only when the bean is run as part of a Form. It
//is not used when the bean is run from the
//command-line.
//See also: AQClientReceiveWrapper and AQClientReceiveBean


package AQClientReceive;


public interface AQClientReceiveListener
{
  public void receiveMessage(String messageText);
```

```
}
```

AQClientReceiveBean

`AQClientReceiveBean` is an example of a client JavaBean that calls an EJB. It uses the Java Naming and Directory Interface (JNDI) to look up `AQServerReceiveHome` on the server. It then instructs `AQServerReceiveHome` to create an instance of `AQServerReceiveBean`. It calls `AQServerReceiveBean` through its remote interface: `AQServerReceiveRemote`. Using this interface it requests `AQServerReceiveBean` to begin listening for Queue messages addressed to it. Because of the insulation around EJBs (described in the section entitled The Three Parts of an Enterprise JavaBean), `AQClientReceiveBean` doesn't call methods on `AQServerReceiveBean` directly.

One of the ways `AQClientReceiveBean` differs from previous example classes is its `main()` method. None of the previous classes, including `Simple`, have implemented a `main()` method. This is the reason why none of the previous classes have been runnable from a command prompt using the `java` command. `AQClientReceiveBean` is the first example class that you can run from a command prompt. You will do so in an upcoming section.

> **Note**: *To make* `AQClientReceiveBean` *work in your environment, change the following line in the source code:*
>
> ```
> String serviceURL = "sess_iiop://dmaas-lap.us.oracle.com:2481:ora815";
> ```

Alter it in the following three ways:

- Replace `dmaas-lap.us.oracle.com` with the name or address of your Oracle8*i* database server.
- Replace `2481` with the port number for your Oracle8*i* GIOP listener.
- Replace `ora815` with the SID for your Oracle8*i* database.

You can find the your settings for these values in the `[Oracle8i_HOME]\network\admin\listener.ora` file provided with your Oracle8*i* database.

Following is the source for `AQClientReceiveBean`.

```
//AQClientReceiveBean.java


//This the main client class.
//It has two important methods:
```

```
//listen, in which it calls up to the server EJB
//        and requests that it listen for new messages
//and
//receiveMessage, which is called by the server EJB
//                (by way of AQCallbackStub) when a
//                new message arrives for it.
//The rest of the code is mostly overhead associated
//with JNDI context lookup and threading.
//See also: AQClientReceiveWrapper.java and AQCallbackStub.java

package AQClientReceive;

//These are imported so we know what methods we can call on the server.
import AQServerReceive.AQServerReceiveRemote;
import AQServerReceive.AQServerReceiveHome;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.ewt.lwAWT.*;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import java.util.Vector;
import java.sql.SQLException;
```

```
//We extend LWComponent so that it is runnable from Forms.
//We implement Runnable to allow it to thread.
public class AQClientReceiveBean
       extends oracle.ewt.lwAWT.LWComponent
       implements Runnable
{
  private String               me                 = null;
  private String               message            = null;
  private Vector               listeners          = new Vector();
  private AQServerReceiveRemote myAQServerReceiveRemote;
  private AQCallbackStub       myAQCallbackStub   = null;
  private Thread               selfThread         = null;

  //Its constructor. It instantiates an AQCallbackStub (phone number)
  //which we will send up to the server so it can call us back when
  //something that we care about happens. It also looks up the server
  //EJB, asks the server to create an instance of it and then saves
  //a handle to it for use by the other methods.
  public AQClientReceiveBean()
  {
    super();
    myAQCallbackStub = new AQCallbackStub(this);
```

```
try
{
  //These values must be changed if you want to try this example
  //for yourself. user and password are the database username and
  //password where the server EJB has been deployed using the
  //deployejb utility (see also AQServerReceive.ejb). serviceURL
  //contains the machine name:port number:sid of the Oracle8i
  //database where the server EJB has been deployed.  objectName
  //is the virtual home directory where the server EJB has been
  //exposed using deployejb. When combined, the serviceURL and
  //objectName form a complete reference to the location of the
  //server EJB, in a format similar to a standard Web URL.
  String user       = "aqdemo";
  String password   = "aqdemo";
  String serviceURL = "sess_iiop://dmaas-lap.us.oracle.com:2481:ora815";
  String objectName = "/test/AQReceive";

  //Create a JNDI context with these values so we can attempt to
  //find the server EJB.
  Hashtable environment   = new Hashtable();
  environment.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
  environment.put(Context.SECURITY_PRINCIPAL, user);
  environment.put(Context.SECURITY_CREDENTIALS, password);
  environment.put(Context.SECURITY_AUTHENTICATION,
ServiceCtx.NON_SSL_LOGIN);
```

```
      Context myContext = new InitialContext(environment);


      //Using the context, lookup the Home interface for the server EJB.
      AQServerReceiveHome myAQServerReceiveHome =
(AQServerReceiveHome)myContext.lookup(serviceURL + objectName);
      //Instruct the server EJB's Home interface to create an instance
      //of it for us. This yields a handle to its Remote interface,
      //which is the means by which we communicate with it.
      myAQServerReceiveRemote = myAQServerReceiveHome.create();
    }
    catch(Exception myException)
    {
      myException.printStackTrace();
    }
  }


  //This method exists so we can run the bean from the
  //command-line. It does not function when the bean is
  //embedded in a Form. It accepts a command-line argument
  //and listens for that name.
  public static void main(String[] args)
    throws Exception
  {
    if(args.length < 1)
    {
```

```
    System.out.println("usage=AQClientReceiveBean [listen for name]");
    System.exit(1);
  }
  AQClientReceiveBean myAQClientReceiveBean = new AQClientReceiveBean();

  myAQClientReceiveBean.me = args[0];
  myAQClientReceiveBean.listen(myAQClientReceiveBean.me);
}

//This method allows the bean's wrapper to tell it
//that it wants to know about any messages that the
//bean receives. It is used only when the bean is
//run from a Form. It does not function when the bean
//is run from the command-line.
public void addAQClientReceiveListener(AQClientReceiveListener newListener)
{
  listeners.addElement(newListener);
}

//This is the method that indirectly calls to the
//server EJB when we are ready to start listening
//for messages. It creates a separate thread to do
//this. Its logic is actually resolved in the run()
//method (next).
```

```
public void listen(String messageRecipient)
{
  if(messageRecipient == null)
    return;

  me = messageRecipient;
  if(selfThread == null)
  {
    selfThread = new Thread(AQClientReceiveBean.this);
    selfThread.start();
  }
}

//The run method is called when a new thread is
//started by the listen() method (above). It calls
//to the server EJB using the handle to it that we
//established in the constructor. It passes it a
//reference to our CallbackStub, which is our phone
//number, so the server EJB can call us back, and
//our name (the person whose messages we want it
//to listen for and call us back with).
public void run()
{
  try
```

```
  {
    myAQServerReceiveRemote.waitForMessage( myAQCallbackStub, me);
  }
  catch(Exception myException)
  {
    myException.printStackTrace();
  }
}

//When the server EJB does receive a message for
//us, it calls first to AQCallbackStub using the
//handle to it that we gave it in the run() method;
//then AQCallbackStub calls to us in this  mehtod.
//If there are any bean wrapper objects that also
//want to know about the message received, we loop
//through and call to each of them as well. Note
//that we only would have wrapper objects when the
//bean is being run from a Form.
public void receiveMessage(String messageText)
{
  Vector copyOfListeners;

  System.out.println(messageText);
```

```
    message = messageText;

    synchronized(this)

    {

      copyOfListeners = (Vector)listeners.clone();

    }

    for(int i=0; i < copyOfListeners.size(); i++)

((AQClientReceiveListener)(copyOfListeners.elementAt(i))).receiveMessage(message
Text);

  }

}
```

DEPLOYING THE EJB

Once you have successfully compiled all the example EJB and client JavaBean classes above, you can migrate the EJB into your Oracle8*i* database. You migrate it using the `deployejb` utility provided with Oracle8*i*. Note that this is *not* the same as the `loadjava` utility that you used in part 1. The `loadjava` utility allows you to import classes and sets of classes into an Oracle8*i* database where you may wrap and access them from PL/SQL. The `deployejb` utility allows you to import the classes that comprise EJBs and CORBA objects. These classes are then *published* into a *session namespace* where they may be accessed by objects running outside the database via the Oracle8*i* ORB. You may not use CORBA to access classes imported into the database using the `loadjava` utility.

You must complete the following two steps before you can publish your EJB with the `deployejb` utility:

- You must bundle the class files required by the EJB into a `.JAR` file.
- You must provide a *deployment descriptor* file for the EJB.

Creating the .JAR File

A group of class files can be bundled together into a single `.JAR` (Java ARchive) file using the `jar` command line utility provided with your JDK.

To create the `.JAR` file:

1. Verify that you have, as suggested, saved and compiled the files that comprise the EJB in a separate subdirectory with the same name as the EJB package (`AQServerReceive`). If you haven't, do so now. (See the section in part 2 entitled Setup Steps.)

2. Verify that you have, as suggested, saved and compiled the files that comprise the client JavaBean in a separate subdirectory with the same name as the client package (`AQClientReceive`). This entails having run the `java2rmi_iiop` utility as described in the section entitled CallbackStub. If you haven't completed these steps, do so now. (Also see the section in part 2 entitled Setup Steps.)

3. Run the following command from a command prompt window while in your example parent directory:
```
jar cf0 AQServerReceive.jar AQClientReceive/AQCallback.class
AQClientReceive/AQCallbackHolder.class AQClientReceive/AQCallbackHelper.class
AQClientReceive/_st_AQCallback.class AQServerReceive/AQServerReceiveRemote.class
AQServerReceive/AQServerReceiveHome.class AQServerReceive/AQServerReceiveBean.class
```

If all these steps were completed successfully, you should have a new file, `AQServerReceive.jar`, in your example parent directory.

You could not complete this step earlier because some of the files required to publish the EJB are part of the client JavaBean. You can see these included in the `.JAR` file in step 3 above.

The Deployment Descriptor

Below is a deployment descriptor that will instruct the `deployejb` utility how to publish your EJB. The deployment descriptor is not a Java class. It is actually more like an `.INI` file. You should create this file in your example parent directory.

```
//AQServerReceive.ejb


//This is the deployment descriptor for the server EJB
//classes. It is read by the deployejb utility. It is
//not java or code. It's more like a .INI file.


//The name of the EJB.
SessionBean AQServerReceive.AQServerReceiveBean
{
  //The virtual directory in the Oracle8i server
```

```
  //in which to expose the EJB.
  BeanHomeName = "test/AQReceive";
  //The name of the EJB's Remote interface.
  RemoteInterfaceClassName = AQServerReceive.AQServerReceiveRemote;
  //The name of the EJB's Home interface.
  HomeInterfaceClassName = AQServerReceive.AQServerReceiveHome;

  //How long to wait before timing out the EJB
  //when it is not in use.
  SessionTimeout = 5;

  //Who should have access to the EJB.
  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;

  //Where/how should any transactions for the EJB be managed.
  TransactionAttribute = TX_BEAN_MANAGED;
}
```

Running the deployejb utility

Now you may run the `deployejb` command.

> ***Note 1**: Your Oracle8i database and listener must be running for the* `deployejb` *utility to work.*
>
> ***Note 2**: To make the* `deployejb` *utility work in your environment, you must change the* `-s` *argument in the provided command line example as follows:*

```
-s sess_iiop//dmaas-lap.us.oracle.com:2481:ora815
```

Alter the argument in the following three ways:

- Replace `dmaas-lap.us.oracle.com` with the name or address of your Oracle8*i* database server.
- Replace `2481` with the port number for your Oracle8*i* GIOP listener.
- Replace `ora815` with the SID for your Oracle8*i* database.

You can find the your settings for these values in the `[Oracle8i_HOME]\network\admin\listener.ora` file provided with your Oracle8*i* database.

Following is the example `deployejb` command:

```
deployejb -republish -temp temp -u aqdemo -p aqdemo -s sess_iiop://dmaas-
lap.us.oracle.com:2481:ora815 -descriptor AQServerReceive.ejb
AQServerReceive.jar
```

Once started, the `deployejb` utility may run as long as a of couple minutes depending on your database server platform. If it completes successfully, it will produce *only* the following messages before returning to a command prompt:

```
Reading Deployment Descriptor...done

Verifying Deployment Descriptor...done

Gathering users...done

Generating Comm Stubs.....................................done

Compiling Stubs...done

Generating Jar File...done

Loading EJB Jar file and Comm Stubs Jar file...done

Generating EJBHome and EJBObject on the server...done

Publishing EJBHome...done
```

If any other messages appear, an error has occurred. If it completes successfully, the `deployejb` utility will create an `AQServerReceive_generated.jar` file in your example parent directory. Add this file to your `CLASSPATH`.

TESTING THE ENTERPRISE JAVABEAN

You can now run and test your published Enterprise JavaBean from your Oracle8*i* database.

Adding Messages to the Queue

If you completed the AQServerSend sections of part 1, add messages to the Queue from a SQL*Plus window using the following commands:

```
EXEC aqserversend_wrapper('Jim', 'What are you doing? –Phil');
COMMIT;
```

In this example, Jim is the addressee (intended recipient) of the message, and the second argument is the content of the message to be delivered to Jim.

Starting the EJB to Retrieve Messages from the Queue

Start your client JavaBean, which should start your EJB, by issuing the following command:

```
java AQClientReceive.AQClientReceiveBean Jim
```

In this example, Jim is a command line argument to `AQClientReceiveBean` that specifies the addressee whose messages you would like to retrieve. After a few seconds of processing, the client Bean should display, in the command prompt window, any messages addressed to its argument that have been committed to the Queue. If it does not display known messages, ensure that you have spelled the name of the addressee exactly the same when sending the message and when starting the listener. Remember that Java is sensitive to case. If it still does not display know messages after carefully checking the argument values, you may have a bug. If so, follow the steps for stopping listener below. Then carefully check and recompile you source code.

Stopping the EJB and the Client JavaBean

Unless you have changed the source code for `AQServerReceiveBean`, the EJB will continue to wait for new messages until it receives a message, addressed to its argument, with a message content of `!!!STOP!!!` (case-sensitive). You can send such a message from a SQL*Plus window using the following commands:

```
EXEC aqserversend_wrapper('Jim', '!!!STOP!!!');

COMMIT;
```

In this example, Jim is the addressee whose EJB you would like to stop.

Once the EJB is stopped, you may stop the client JavaBean by pressing the `CTRL-C` keyboard combination from the command prompt window in which you started it. If you do not stop your EJB, as the result of a bug or the omission of this step, you may have to forcibly shut down and restart your Oracle8*i* database before you are able to rerun or republish the EJB.

CONCLUSION OF PART 2

You ll find more information and examples of Enterprise JavaBeans in *Oracle8i Enterprise JavaBeans and CORBA Developer s Guide*. You can get further information about the Oracle8*i* Advanced Queuing Option in *Oracle8i Application Developer s Guide – Advanced Queuing*. Oracle supplies both of these volumes with your Oracle8*i* database in both HTML and PDF formats.

PART 3 OF 3: INTEGRATING ORACLE FORMS SERVER

Oracle Forms 6*i* combines the functionality of Java with the power of a Rapid Application Development (RAD) tool. It does this through two key features:

- Pluggable Java Components, which allow you to use Java to tailor your own functionality into existing Forms widgets
- The Bean Area, which allows you to embed JavaBeans in your Forms Server applications

THE BEAN WRAPPER CLASS

You may embed a JavaBean in an Oracle Form by creating Bean Wrapper class for that JavaBean. The Bean Wrapper class must extend the `oracle.forms.ui.VBean` class that is provided with Forms under your `[ORACLE_HOME]\forms60\java` directory.

When you create the Bean Wrapper class, it registers and handles any properties, methods, or events in the JavaBean that you would like to make available to the Form. The Bean Wrapper also instantiates the JavaBean when the Form requests it.

AQClientReceiveWrapper

`AQClientReceiveWrapper` is an example JavaBean Wrapper class. It is a wrapper for the `AQClientReceiveBean` that you created in part 2. It registers the following three Bean components with its Form:

- `messageText`, a property in which the wrapper gives the Form the content of a message that has been sent down from the Queue
- `listen`, a method in which the wrapper tells the Bean to call the EJB, so the EJB will start retrieving messages
- `receiveMessage`, an event that the wrapper fires in the Form to tell the Form when a new message has arrived

When `AQClientReceiveWrapper` instantiates `AQClientReceiveBean` on behalf of the Form, it also instantiates and passes an `AQClientReceiveListener` to `AQClientReceiveBean`. In much the same way as `AQCallbackStub`, `AQClientReceiveListener` allows `AQClientReceiveBean` to call back to `AQClientReceiveWrapper` when a new message arrives without `AQClientReceiveWrapper` having to continuously poll `AQClientReceiveBean` for new messages. The difference is that `AQCallback` uses RMI to allow an object to call from one machine or environment to another, while the `AQClientReceiveListener` can only be used to call to objects on the same machine or environment. This is the purpose of `AQClientReceiveListener`.

`AQClientReceiveWrapper` is part of the `AQClientReceive` package. Save and compile it to that subdirectory.

Following is the source for `AQClientReceiveWrapper`:

```
//AQClientReceiveWrapper.java

//This is the mandatory wrapper class that allows
//Forms to communicate with a bean. That means that
//all communication from the Form to the bean is
//handled and passed along here and all communication
//from the bean back to the Form is handled and
//passed along here. It is used only when the bean is
//run as part of a Form. It is not used when the bean
//is run from the command-line. Since it is called first
//by Forms, it creates the bean and registers any
//properties, methods, or events in the bean that we
//want to allow the Form to communicate with. It
//receives incoming messages from the bean by way of
//the AQClientReceiveListener interface.
//See also AQClientReceiveBean.java and AQClientReceiveListener.java

package AQClientReceive;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
```

```
import java.io.*;


import oracle.forms.properties.ID;
import oracle.forms.handler.IHandler;
import oracle.forms.ui.CustomEvent;
import oracle.forms.ui.VBean;


//It must be a VBean so it can be a part of a Form.
//It implements AQClientReceiveListener so that the
//bean can call back to it with incoming messages.
public class AQClientReceiveWrapper
       extends VBean
       implements AQClientReceiveListener
{


  private Component myBeanComponent;
  private IHandler  myIHandler;


  //Forms treats properties, methods, and events as properties.
  //This registers the property of  messageText with
  //Forms so that the Form can get the content of any
  //messages we receive.
  private static final ID MESSAGETEXT    = ID.registerProperty("messageText");
  //This registers the listen method, which will tell
```

```
  //the bean to start listening when the Form calls it.
  private static final ID LISTEN         = ID.registerProperty("listen");
  //This registers the receiveMessage event, which farther
  //down we will flesh out so that it will tell the Form
  //to fire a WHEN-CUSTOM-ITEM-EVENT for.
  private static final ID RECEIVEMESSAGE =
ID.registerProperty("receiveMesssage");


  public AQClientReceiveWrapper()
  {
    super();
    try
    {
      //Create an instance of the bean we wish to use.
      ClassLoader myClassLoader = getClass().getClassLoader();
      Object myAQClientReceiveBean = Beans.instantiate(myClassLoader,
"AQClientReceive.AQClientReceiveBean");


      Thread myBeanThread = new Thread((Runnable)myAQClientReceiveBean);
      myBeanThread.start();


      //Make a component representation of the instantiated bean.
      myBeanComponent = (Component)myAQClientReceiveBean;
      myBeanComponent.setVisible(true);
      myBeanComponent.setEnabled(true);
```

```
    //Here we tell the bean to add us to its list of listeners
    //to notify whenever it receives a message.
    ((AQClientReceiveBean)myBeanComponent).addAQClientReceiveListener(this);
    add("Center", myBeanComponent);
  }
  catch(Exception myException)
  {
    myException.printStackTrace();
  }
}

//The initialization method called by the Forms immediately after
//the object is constructed. A handler reference is passed in.
public void init(IHandler myIHandler)
{
  super.init(myIHandler);
  this.myIHandler = myIHandler;
}

//The Form calls this method to elimate us.
//Here we can release any resources we are using prior to destruction.
public void destroy() {}
```

```
//This method is called when the Form issues SET_CUSTOM_ITEM_EVENT()
//on the bean. Keep in mind that this is the means by which
//a bean property is set or a bean method is called.
public boolean setProperty(ID propertyId, Object propertyValue)
{
  try
  {
    //In this case, the only thing we let the Form do is
    //tell us to start listening, in which case we use our
    //handle to the bean to tell it to start listening.
    if(propertyId == LISTEN)
      ((AQClientReceiveBean)myBeanComponent).listen((String)propertyValue);
    else
      return super.setProperty(propertyId, propertyValue);
  }
  catch(Exception myException)
  {
    myException.printStackTrace();
  }
  return true;
}

//Used for getting properties.
public Object getProperty(ID propertyId)
```

```
{
  return null;
}

//This method is required when we implement the
//AQClientReceiveListener interface. It the means
//by which the bean gives us any incoming messages.
public void receiveMessage(String messageText)
{
  try
  {
    //Here we save the text of the incoming message
    //in the MESSAGETEXT property that we registered
    //at the top.
    myIHandler.setProperty(MESSAGETEXT, messageText);
  }
  catch(Exception myException)
  {
    myException.printStackTrace();
  }
  //After we receive the message, we fire the RECEIVEMESSAGE
  //event that we registered at the top. This causes a
  //WHEN-CUSTOM-ITEM-EVENT to fire in the Form.
  CustomEvent myCustomEvent = new CustomEvent(myIHandler, RECEIVEMESSAGE);
```

```
        dispatchCustomEvent(myCustomEvent);
      }
    }
```

ADD THE JAVABEAN TO A FORM

Setup Steps

1. Before starting Oracle Forms Developer, verify that the Oracle JDK has been installed in the `ORACLE_HOME` where Forms resides. You can determine this by invoking the Oracle Installer or by positively determining that there are files and subdirectories under the `[ORACLE_HOME]\JDK` directory in the `ORACLE_HOME` in which you have installed Forms. If the Oracle JDK has *not* been installed, use your Oracle Installer and Forms media to install it.

2. Open a new command prompt window. In this window, set the `PATH` environment variable so that the `[ORACLE_HOME]\JDK\BIN` directory where the Oracle JDK is installed appears before any other `\JDK\BIN` directories. Verify that the `[ORACLE_HOME]\BIN` directory containing the `IFBLD60.EXE` file is included in your `PATH` environment variable. Also verify that your `CLASSPATH` environment variable includes either the `AQClientReceive` subdirectory of your example parent directory or a `.JAR` file into which `AQClientReceiveWrapper.class` has been bundled.

3. Launch Oracle Forms Developer by typing `ifbld60` from this command prompt window.

You can omit these Setup Steps, but it will prevent your JavaBean from being rendered in the Forms Layout Editor. If you did so, you would only be able to edit your Canvas in the Layout Editor while your JavaBean was not assigned to that Canvas. However, the JavaBean should still function at runtime in a Web browser as long as you follow the deployment steps.

Inside Forms Developer

1. When you have started Forms Developer, open or recreate the Form you created in part 1 to call your PL/SQL Call Spec: `aqserversend_wrapper()`. This Form should already be capable of sending messages to Queue.

2. You can now enhance this Form by adding another Text Item and another Button Item. You will add PL/SQL code to the Button later. The Text Item will allow the user to specify the addressee for which to get messages (the addressee to listen for). The Button will start the listener by calling `AQClientReceiveWrapper`.

3. In the Layout Editor, add a Bean Area to your Canvas. Since it will be a non-visual Bean, you may want to make it small and put it out of the way of the other Items. However, it must be present on the same Canvas as the visual Items to function.

4. In the Property Palette, specify the Implementation Class property for the Bean Area. The Implementation Class should be `AQClientReceive.AQClientReceiveWrapper`. If you completed the Setup Steps above correctly, the Bean should appear in the Layout Editor window (if open), or the Layout Editor should be openable (if it is not open). If Forms Developer displays an `FRM-13008: Cannot find JavaBean with name` message, an error has occurred. Even if Forms Developer is not able to locate or display your Bean, the Bean still may function at runtime in a Web browser.

5. Add PL/SQL code to the `WHEN-BUTTON-PRESSED` trigger for the new start listening Button. Following is an example of this PL/SQL:

   ```
   SET_CUSTOM_ITEM_PROPERTY('AQ_RECEIVE_BEAN', 'listen', :LISTEN_AS);
   ```

   In this example, `AQ_RECEIVE_BEAN` is the name of the Bean Area, and `:LISTEN_AS` is the name of the Text Item that will supply the addressee name for which to listen.

6. Add PL/SQL code to the `WHEN-CUSTOM-ITEM-EVENT` trigger for the Bean Area. Following is an example of this PL/SQL:

```
DECLARE
  list_id         PARAMLIST;
  parameter_type  NUMBER;
  parameter_value VARCHAR2(300);
BEGIN
  list_id := get_parameter_list(:SYSTEM.CUSTOM_ITEM_EVENT_PARAMETERS);
  get_parameter_attr(list_id, 'messageText', parameter_type, parameter_value);
  message('Incoming Message: ' || parameter_value);
  message('Incoming Message: ' || parameter_value);
END;
```

RUNNING THE FORM

You have now seen or built every component that is necessary for an Oracle Form that can pop up asynchronous messages and engage in chat sessions with other Forms using an Oracle8*i* Advanced Queue. Now you must simply make the components available in a Web browser.

If you said at the beginning of the paper that, "You know how to get an Oracle Form that you have built to run in your Web browser," I will assume that you have already set up a Web Server with the appropriate virtual directories to run Forms. If you are not able to run Forms that you create in your Web browser, then you will have to resolve those issues before you proceed further. Configuring the virtual directories, registry settings, environment variables and HTML files necessary to run Forms on the Web is outside the scope of this paper.

Deploying the Class Files

You can make your class files available for download in one of two ways:

- Package up your client class files into `AQClientReceive.jar` using the `jar` utility. You can find an example of how to use the jar utility in the section entitled .Then copy this `.JAR` file to your Web Server Forms Java virtual directory. Add `AQClientReceive.jar` to the `archive` tag in the HTML file that loads your Form. This is the method we recommend.

OR

- Copy your example parent directory and all package subdirectories and `.CLASS` files to your Web Server Forms Java virtual directory.

With either approach, the two EJB `.JAR` files (`AQServerReceive.jar` and `AQServerReceive_generated.jar`) will also need to be copied to your Web Server Forms Java virtual directory and added to the `archive` tag in the HTML file that loads your Form.

Resolving JVM Security

You can resolve JVM security issues for your classes in one of two ways:

- Get Microsoft Internet Explorer 5 and the Microsoft VM (must be downloaded and installed separately from `http://www.microsoft.com/java/`). Turn off unsigned content restrictions. You can do this in Internet Explorer from Tools –> Internet Options –> Security –> Local Intranet –> Custom Level... –> Java Custom Settings... –> Edit Permissions –> Run Unsigned Content –> Enable. Run your Form natively in Internet Explorer (without JInitiator) by using the `APPLET` tag and omitting the `OBJECT` tab in the HTML file. This is the easiest solution for testing purposes, but should not be done for production applications.

OR

- Sign the .JAR files containing your classes using the `javakey` utility provided with your JDK. Additional information on the `javakey` utility can be found at `http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javakey.html` .

Runtime Classpath

When running Internet Explorer, your `CLASSPATH` environment variable should point only to `[JDK]\lib\classes.zip`.

Example HTML File

Following is an example HTML file to load the AQ-enabled Form:

```
<HTML>

<HEAD>

  <TITLE>Using Oracle8i from Oracle Forms Developer</TITLE>

  <X-CLARIS-WINDOW TOP=25 BOTTOM=608 LEFT=4 RIGHT=927>
```

```
</HEAD>

<BODY>

<APPLET WIDTH=700

        HEIGHT=550

        <PARAM NAME="codebase"    VALUE="/test_java">

        <PARAM NAME="code"        VALUE="oracle.forms.engine.Main">

        <PARAM NAME="archive"     VALUE="f60all.jar, AQClientReceive.jar,
AQServerReceive.jar, AQServerReceive_generated.jar, vbjorb.jar, vbjapp.jar,
aurora_client.jar">

        <PARAM NAME="lookandfeel" VALUE="oracle">

        <PARAM NAME="colorscheme" VALUE="blue">

        <PARAM NAME="serverhost"  VALUE="dmaas-lap">

        <PARAM NAME="serverport"  VALUE="9000">

        <PARAM NAME="serverargs"  VALUE="AQ.fmx userid=aqdemo/aqdemo@local">

        <PARAM NAME="serverapp"   VALUE="default">

</APPLET>



</BODY>

</HTML>
```

CONCLUSION OF PART 3

You ll find more information about using JavaBeans in Forms 6*i* in the online help files supplied with Oracle Forms 6*i*.