

Client Side OLE Integration from Oracle9i Forms

Technical White Paper
September 2002

Client Side OLE Integration from Oracle9i Forms

PURPOSE

This document provides a technical overview of how to achieve client side OLE integration from a web browser deployed Forms application.

INTRODUCTION

Integration with desktop OLE servers such as Microsoft Word and Excel is a commonly used function within Microsoft Windows deployed client-server Forms applications. However, the use of that same OLE integration technology ties the application to the Windows platform and can create a significant barrier to the web deployment of those same applications.

When web deployed through Oracle9iAS Forms Services, any Forms modules that perform OLE integration through PL/SQL will execute that OLE integration on the application server tier. This requires that:

1. The Application Server must be running on a Windows platform (with the relevant OLE server, such as Word or Excel, installed)
2. The OLE integration performed is invisible to the user, as it is all taking place on the application server rather than in the client (browser) machine.

If the functionality that is being used is suitable for operation under this set of constraints (for instance the OLE server is being used to perform a mail-merge type operation which outputs batch results to a printer), then web deployment of the application is not a problem. However, in situations where the application requires that the end user have some input or interaction with the OLE server then this solution is not suitable.

So, how do we emulate the OLE integration capabilities of Forms, with an OLE server such as Word, which is deployed into the “client browser” machine rather than the application server tier?

This paper describes how you can leverage the extensibility of the Forms Java client to add OLE integration capabilities on the browser machine.

WHAT WILL THIS PAPER COVER?

For this particular example the integration shown is simple. The code described in this article provides interfaces to carry out the following tasks.

- Open Word and optionally make it visible to the end user
- Create a new document
- Write some text passed from the Form into the document
- Save the document with a specified file name
- Print the document to the default local printer
- Close the document
- Close Word.

Note that this is all external activation of an OLE server. The techniques described here do not emulate the use of an OLE Container item on a Forms canvas.

This paper assumes that you have experience of OLE Automation and are familiar with the concepts of OLE and the interfaces exposed by Microsoft Word. The principles embodied within the paper can be extended to use with any OLE server, but the responsibility of understanding how to use a particular OLE server lies with the programmer writing the automation code.

WHAT TECHNOLOGIES CAN BE USED?

There are several ways that you can talk from Java (the language that we use to extend the Forms client) into OLE. It is possible to use the J/Direct™ interface provided by the Microsoft in their Java virtual machine (JVM). But this does tie you to using Internet Explorer in native JVM mode only¹. So a better solution would be to use an add-on package for Java that can be used with any Java virtual machine, not just the Microsoft proprietary one.

There are several vendors who supply such packages, but for the purposes of this paper I have used JACOB, which is an open source Java to COM (OLE) bridge available from <http://danadler.com/jacob>.

¹ Note that the Microsoft Internet Explorer native JVM is not currently support for the deployment of Oracle9i Forms applications, so this is a further reason not to use J/Direct

CREATING THE JAVA TO OLE INTERFACE.

The mechanism that we use to add functionality to extend the Forms Java client that actually runs in the browser is to either drop in a JavaBean or to use a Pluggable Java Component (PJC). The former technique is only available in Oracle9i Forms, whereas the latter is available in both Forms 6i and Oracle9i Forms.

For simplicity, this article covers the use of a drop in JavaBean for Oracle 9i Forms, as the code simpler and easier to illustrate in a step by step manner.

For more information about Java integration see the white paper “Oracle9i Forms in the Java World” on OTN (<http://otn.oracle.com>).

Defining the Bean in JDeveloper

The JavaBean itself can be created through Oracle9i JDeveloper using the following steps:

1. Create a new Workspace using **File** → **New** from the menu and selecting the **Projects** entry in the **Categories** on the left hand side of the new object gallery dialog. Choose **Workspace** from the options presented.

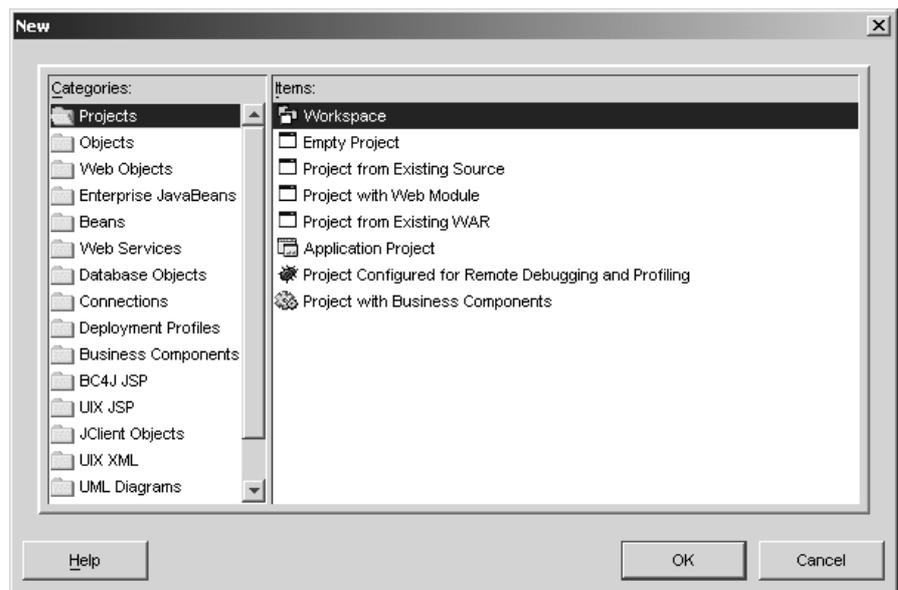


Figure 1 - Creating a new workspace in JDeveloper 9.0.2

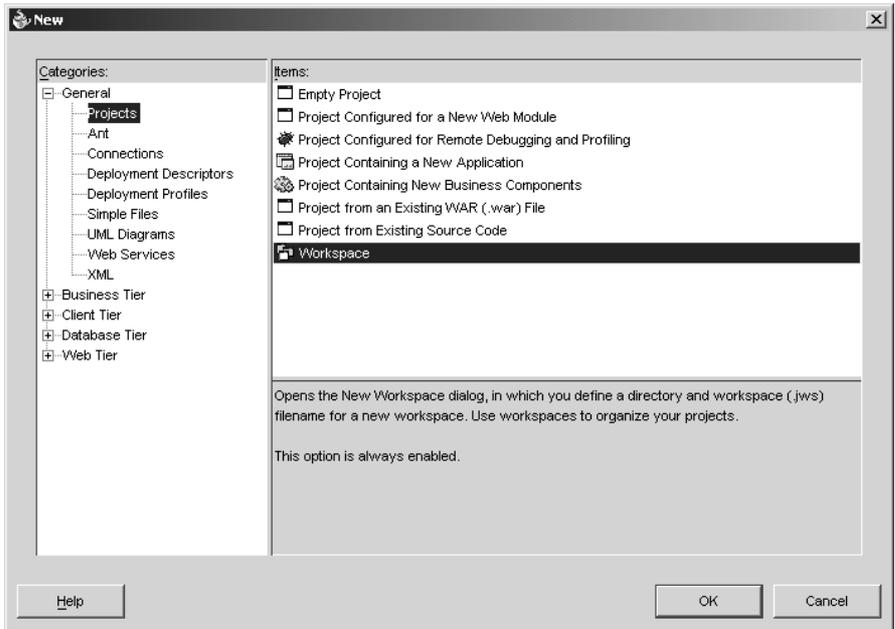


Figure 2 - Creating a new workspace in JDeveloper 9.0.3

2. Create a new empty project in the same way
3. Again choose the **File** → **New** menu option and this time select **Beans** (or **Client Tier** → **JavaBeans** in JDeveloper 9.0.3) in the **Category** pane and **Bean** on the right hand side – you will then be prompted to define the package name (e.g. *oracle.forms.demos.ole*), the class name (e.g. *WordBean*) for your class and to select the type of object that the Bean will extend. Choose *java.awt.Panel* for this value.

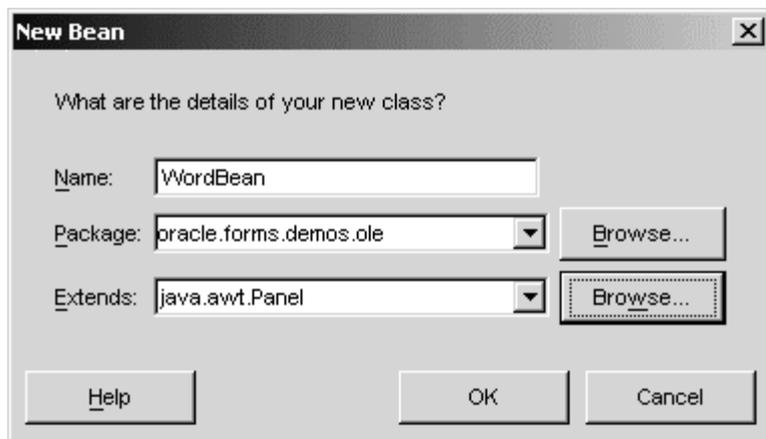


Figure 3 - Creating a new JavaBean

You will now have an empty JavaBean class, which needs to have our OLE integration code added.

4. Right mouse click on the new project that you created and choose **Project Settings** from the context menu. Go to the Libraries settings under **Configurations** → **Development** in the tree control on the left hand side.
5. Select the JACOB Library to be included in the project.

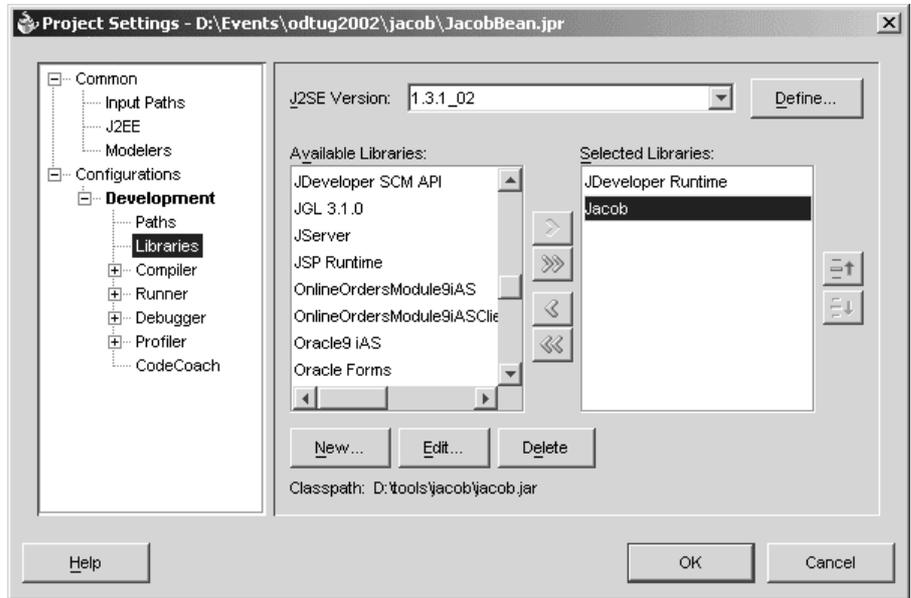


Figure 4 - Adding the JACOB library to the project

6. If the JACOB library is not defined in the list of libraries that are available, select the “**New...**” button and add it. You will have had to download the jacob.jar from the JACOB site before doing this. Once you have defined the classpath of the JACOB library, it should automatically have been added to the list of selected libraries.
7. Add the imports for *com.jacob.com.** and *com.jacobactivex.** to the top of the WordBean code.

You should now have the skeleton of a JavaBean and we can start to add methods. We'll add one method to carry out each of the functions that we listed at the start.

The Bean Methods in Detail

In this section of the paper we drill down onto the various methods required to carry out each of our required actions. The complete code listing of the WordBean class is available for reference at the end of this paper.

1. Open Word and Make it Visible to the End User

First of all create private class variables to hold references to Word and to the document object we'll be creating. These variables will be referenced throughout the session by the various OLE function calls.

```
private ActiveXComponent MsWordApp = null;
private Dispatch document = null;
```

Figure 5 – Class variables required in WordBean

Now, the actual method to create an instance of the Word OLE server. You'll notice that this method takes a boolean argument to define if Word should be made visible to the end user or not. At this stage you also need to know the name of the OLE server. This is the value, which would have been used in the Forms OLE code, and corresponds to the identity that the server has created in the Windows Registry. In this case it is "Word.Application".

```
public void openWord(boolean makeVisible)
{
    //Open Word if we've not done it already
    if (MsWordApp == null)
    {
        MsWordApp = new ActiveXComponent("Word.Application");
    }
    //Set the visible property as required.
    Dispatch.put(MsWordApp, "Visible",
        new Variant(makeVisible));
}
```

Figure 6 – openWord method to start the Word OLE server

Notice how the static method *put()* in the *Dispatch* object is used to set a property which is passed by name. The actual value to set for the property is passed as the type **Variant**, so the boolean value supplied to the method is used to create a new Variant object first, which is then submitted to *put()* rather than the value being set directly.

2. Create a new document

Now that Word is running we can obtain the *Documents* collection (array) from the Word application object and call the *add()* method on that collection to add a new document.

```
public void createNewDocument()  
{  
    //Find the Documents collection object  
    //maintained by Word  
    Dispatch documents =  
        Dispatch.get(MsWordApp, "Documents").toDispatch();  
    //Call the Add method of the Documents collection to  
    //create a new document to edit  
    document = Dispatch.call(  
        documents_collection, "Add").toDispatch();  
}
```

Figure 7 – createNewDocument method to create a new empty document in Word

In this method we both get an OLE property with *Dispatch.get()* and call an OLE method with *Dispatch.call()*. In both cases the object returned is something that we'll want to do OLE automation with later on, so we call the *toDispatch()* method on the return from both the *get()* and *call()* to make sure that the objects we save into the documents and document objects are capable of being passed to the Dispatch methods in their own right.

3. Write some text into the document

The *insertText()* method adds the specified string, at the current default selection point of the document, by setting the *Text* property of the *Selection* object. We can ask Word for the current Selection object using the Selection property.

```
public void insertText(String textToInsert)  
{  
    // Get the current selection within Word at the moment.  
    //If a new document has just been created then this will  
    //at the top of the new document  
    Dispatch selection =Dispatch.get(  
        MsWordApp, "Selection").toDispatch();  
    //Put the specified text at the insertion point  
    Dispatch.put(selection, "Text", textToInsert);  
}
```

Figure 8 – insertText method inserts the supplied text at the default insertion point

Again because the selection object is something we're going to use *Dispatch.put()* on later on we have to convert it to a Dispatch object with *toDispatch()* in the *get()* call.

4. Print the Document

Here, we just print the current document to the default printer, with all the default settings. We could pass Variants to the Word *PrintOut* method to define the printer settings such as number of copies, print to, print from, and so on.

```
public void printFile()  
{  
    //Just print the current document to the default printer  
    Dispatch.call(document, "PrintOut");  
}
```

Figure 9 – Printing to the default printer with all the default settings

5. Save the document with a specified file name

Again, a very simple call to the *SaveAs* method of the Document object.

```
public void saveFileAs(String filename)  
{  
    Dispatch.call(document, "SaveAs", filename);  
}
```

Figure 10 – Saving the document

6. Close the Document

To close the document, call its *close* method. You can pass an optional argument to the close method to define if you want to save the file automatically or not. In this case the code specifies that changes should not be saved. The various valid values for this argument are defined by name in the Visual Basic Help for Word Automation, and the numerical values of each of the specified constants can be found by viewing the Word type library with the OLEView tool from Microsoft.

```
public void closeDocument()  
{  
    // Close the document without saving changes  
    // 0 = wdDoNotSaveChanges  
    // -1 = wdSaveChanges  
    // -2 = wdPromptToSaveChanges  
    Dispatch.call(document, "Close", new Variant(0));  
    Document = null;  
}
```

Figure 11 – Closing the new document without saving

7. Close Word.

Call the *quit* method on the Word OLE server.

```
public void closeWord()  
{  
    Dispatch.call(MsWordApp, "Quit");  
    MsWordApp = null;  
    document = null;  
}
```

Figure 12 – Quitting Word

Packaging the Bean

Once the above Java code has been written and compiled, it'll need to be packaged up into a Java archive (JAR file). To do this, from the File menu or the Context menu in JDeveloper choose **New** and select the **Deployments** category in the object gallery (In JDeveloper 9.0.3 this can be found in the **General** → **Deployment Profiles** node). Select **JAR File – Simple Archive** and accept all the defaults. You will then be able to create and deploy the JAR file by right mouse clicking on the new deployment profile that you have created in the project.

Security and Deployment

The Java Code that has been described here needs to be able to talk out to the operating system to carry out the OLE operations, so it does need rights above and beyond the privileges that a Java applet normally enjoys. In order for the code to be able to do this, it will have to be signed by the developer and then trusted by each client. The signing for Jinitiator 1.3 is done using a JDK tool “jarsigner”, which can be used to sign both the jacob.jar file and the JAR file containing your custom integration code.

Signing can be carried out with a purchased certificate from a third party certificate authority such as Verisign, or if your application is only designed for deployment to a user base that knows and trusts you (for instance an internal company system) then you can self generate a certificate using the “keytool” utility which is also available with the JDK.

More information about signing JAR files can be found in the white paper “*Oracle9i Forms: JAR File Signing for JInitiator 1.3*” available on OTN.

Once the signed JAR files are created you have to add it to the ARCHIVE_JINI tag of the formsweb.cfg entry for the application. In this case the custom OLE integration code has been placed into a (signed) JAR file called WordBean.jar.

A profile in your Formsweb.cfg file used to run an application using the integration code might look something like this.

```
[ole]
pageTitle=Oracle9iAS Forms Services - OLE Integration
archive_jini=f90all_jinit.jar,jacob.jar,WordBean.jar
form= wordsample.fmx
splashScreen=no
```

Figure 13 – Sample formsweb.cfg entry for an OLE enabled application

If you are using a self-created certificate, then the first time that the user runs the application, JInitiator will display a dialog asking the user to trust the code.

In this case both the jacob.jar and the WordBean.jar files have been copied to the Forms90/java directory so that they can be found for download.

Finally JACOB does require a single DLL “jacob.dll” to be copied to each browser client machine that needs to do OLE integration. This file needs to be installed into the Jinitiator /bin directory. For example:

C:\Program Files\Oracle\JInitiator 1.3.1.9\bin\jacob.dll

This installation can be manual or could be accomplished by re-packaging JInitiator with an installation tool such as InstallShield, to include the jacob.dll as well.

RUNNING THE INTEGRATION FROM PL/SQL

So far we've only looked at the Java code required to create an OLE automation JavaBean, so how do we actually use that from within PL/SQL in Forms? The new **FBean2** package introduced in Oracle9i Forms provides a simple way to integrate calls to a JavaBean into PL/SQL. In this case we can write a single trigger to exercise the methods that we created in the integration JavaBean. It will go through the process of creating a new document, adding some text, saving it and finally shutting everything down. In reality all these calls would not normally be made from a single trigger, but it illustrates the point. The first thing to do is to create an empty "Bean Area" item in the Form on a canvas that will be visible to the user when the integration takes place. The Bean Area can be 1x1 with no border and no background and foreground colors to render it effectively hidden (but the visible property must be set to Yes). No special properties are required for this Bean Area to function all the setup is done by the *FBean.Register_Bean* call in the trigger. Next the code to do the automation, this could be for example, a When-Button-Pressed trigger or code on a menu option:

```
declare
  -- CTL.WordBean is the Bean Area Item
  hBean ITEM := Find_Item('CTL.WORDBEAN');

begin
  -- This step would normally be done at
  -- When-New-Form-Instance
  -- It defines what bean to instantiate into the Bean
  -- Area
  FBean.Register_Bean(hBean, 1,
'oracle.forms.demos.ole.WordBean');

  -- Now call the custom methods that we created
  -- in the Bean
  -- Open Word and make it visible
  FBean.Invoke(hBean, 1, 'openWord','true');

  -- Create a new Document
  FBean.Invoke(hBean, 1, 'createNewDocument');

  -- Put some content in
  FBean.Invoke(hBean, 1, 'insertText',
                'Some text to insert');

  -- Save it
  FBean.Invoke(hBean, 1, 'saveFileAs', 'test.doc');

  -- Closedown
  FBean.Invoke(hBean, 1, 'closeDocument');
  FBean.Invoke(hBean, 1, 'closeWord');
end;
```

Figure 14 – Using FBean to call WordBean methods through PL/SQL

² See the Oracle9i Forms on-line help for information about the FBean package.

COMPARING JACOB CODE WITH FORMS OLE2

From the Java code defined in this article, you'll observe that the PL/SQL code you would use to automate an OLE server using the OLE2 package is very similar in structure to the Java code needed to do it with JACOB. This actually makes the translation between the two languages quite simple.

The following table provides a cross reference between the two:

Function	PL/SQL - OLE2	Java – JACOB
Type to hold object which can be automated	OLE2.OBJ_TYPE	Dispatch or ActiveXComponent class
Set a property	OLE2.Set_Property()	Dispatch.set()
Get an object property	OLE2.Get_Obj_Property()	Dispatch.get().toDispatch()
Get a string property	OLE2.Get_Char_Property()	Dispatch.get()
Get a number property	OLE2.Get_Num_Property()	Dispatch.get()
Get a Boolean property	OLE2.Get_BooL_Property()	Dispatch.get()
Invoke a method that returns nothing	OLE2.Invoke()	Dispatch.call()
Invoke a method that returns an object	OLE2.Invoke_Obj()	Dispatch.call().toDispatch()
Invoke a method that returns a string	OLE2.Invoke_Char()	Dispatch.call()
Invoke a method that returns a number	OLE2.Invoke_Num()	Dispatch.call()

Figure 15 – Comparison table for OLE2 and JACOB

Similar comparisons can be made with the other Forms OLE built-ins such as CALL_OLE() which are defined as part of the standard built-ins set rather than be packaged separately like the OLE2 calls.

In either case it should be fairly simple to cross script between the two, to port existing OLE integration functionality into Java code using JACOB.

SUMMARY

Hopefully, this paper has shown that web deployment of Oracle9i Forms applications, and client side OLE integration, do not have to be mutually exclusive options. Using add-on Java packages such as JACOB and the extensibility of the Forms Java client, it is possible, with relatively simple coding, to add OLE integration code to web deployed applications. This can provide those applications with much of the same functionality as is experienced today with the client server edition of Forms on the Windows platform, but with all the benefits of web deployment.

APPENDIX A – WORDBEAN.JAVA

The complete code for the sample JavaBean

```
package oracle.forms.demos.ole;
import com.jacob.activeX.*;
import com.jacob.com.*;

public class WordBean extends java.awt.Panel
{
    private ActiveXComponent MsWordApp = null;
    private Dispatch document = null;

    public WordBean()
    {
        super();
        //output a message to the Java Console so that we can see
        //that something is happening
        System.out.println("Init WordBean");
    }

    public void openWord(boolean makeVisible)
    {
        //Open Word if we've not done it already
        if (MsWordApp == null)
        {
            MsWordApp = new ActiveXComponent("Word.Application");
        }
        //Set the visible property as required.
        Dispatch.put(MsWordApp, "Visible",
            new Variant(makeVisible));
    }

    public void createNewDocument()
    {
        //Find the Documents collection object maintained by Word
        Dispatch documents =
            Dispatch.get(MsWordApp, "Documents").toDispatch();
        //Call the Add method of the Documents collection to create
        //a new document to edit
        document = Dispatch.call(documents, "Add").toDispatch();
    }

    public void insertText(String textToInsert)
    {
        // Get the current selection within Word at the moment. If
        // a new document has just been created then this will be at
        // the top of the new doc
        Dispatch selection =
            Dispatch.get(MsWordApp, "Selection").toDispatch();
        //Put the specified text at the insertion point
        Dispatch.put(selection, "Text", textToInsert);
    }

    public void saveFileAs(String filename)
    {
        Dispatch.call(document, "SaveAs", filename);
    }

    public void printFile()
    {
        //Just print the current document to the default printer
        Dispatch.call(document, "PrintOut");
    }
}
```

```
public void closeDocument()
{
    // Close the document without saving changes
    // 0 = wdDoNotSaveChanges
    // -1 = wdSaveChanges
    // -2 = wdPromptToSaveChanges
    Dispatch.call(document, "Close", new Variant(0));
    document = null;
}

public void closeWord()
{
    Dispatch.call(MsWordApp, "Quit");
    MsWordApp = null;
    document = null;
}
}
```



Client Side OLE Integration from Oracle9i Forms

Version 1.1

September 2002

Author: Duncan Mills

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Oracle is a registered trademark of Oracle Corporation. Various product and service names referenced herein may be trademarks of Oracle Corporation. All other product and service names mentioned may be trademarks of their respective owners.

Copyright © 2002 Oracle Corporation
All rights reserved.