BEA WebLogic jCOM

User Guide

BEA WebLogic jCOM Version 6.1
Document Date: November 1, 2001

Contents

About This Document

This document introduces BEA WebLogic jCOM Bridge features and describes the architecture. The document will concentrate on one direction of the WebLogic jCOM bridge: COM clients making calls to Java objects (COM-to-Java). For more information about Java clients accessing COM components, see the *Reference Guide*.

The document is organized as follows:

- Chapter 1, “Introducing BEA WebLogic jCOM,” is an overview of WebLogic Server jCOM and its architecture.
- Chapter 2, “WebLogic jCOM Programming Models,” describes the different ways in which COM-to-Java communication can be implemented using WebLogic jCOM.
- Chapter 3, “Programming,” describes what the programmer has to do in order to use WebLogic jCOM to access Java components from a COM client.
- Chapter 4, “Deploying Your Application,” provides information about the steps that need to be taken before you can run your application.
- Chapter 5, “WebLogic jCOM Tools” describes the most commonly used WebLogic jCOM tools.

Audience

This document is written for architects, application developers and administrators who are interested in using the WebLogic jCOM feature in WebLogic Server.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page http://www.bea.com, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic jCOM. Please visit http://e-docs.bea.com/wls/docs61.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic jCOM, or if you have problems installing and running BEA WebLogic jCOM, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	**Usage**
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

Convention	Usage

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String` *`CustomerName`*`;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

`[ ]`	Optional items in a syntax line. *Example*:
	`java utils.MulticastTest -n` *`name`* `-a` *`address`*
	`[-p` *`portnumber`*`] [-t` *`timeout`*`] [-s` *`send`*`]`

`|`	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`password {application} {source}`

`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Introducing BEA WebLogic jCOM

The following sections describe how WebLogic jCOM can expand the accessibility of WebLogic Server to COM clients:

- What is BEA WebLogic jCOM?
- The Java COM Conflict
- WebLogic jCOM's Solution to the Conflict
- How WebLogic jCOM Works
- WebLogic jCOM Features

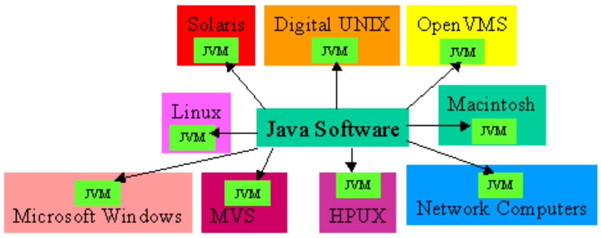
What is BEA WebLogic jCOM?

BEA WebLogic jCOM is a bi-directional COM-Java bridging tool. Using WebLogic jCOM you can access Component Object Model (COM) components as though they were Java objects, and you can access pure Java objects as though they were COM Components.

The Java COM Conflict

In Support of Java

One of the reasons why so many companies are using Java, is that it allows them to dramatically reduce the costs of software development and deployment, because Java programs can execute on any platform that supports a standard Java Virtual Machine (JVM).



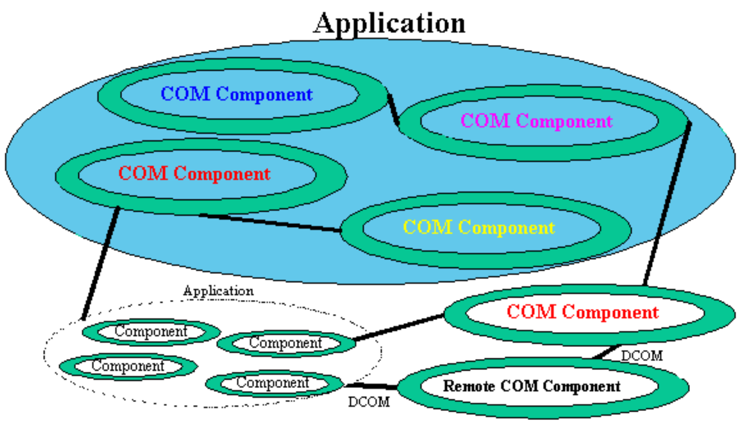
The principle of developing "pure" Java software is a very important one, and many software developers are, correctly, extremely concerned about the use of any software which locks them into a specific platform.

In Support of COM

Software Components are a natural evolution of Object Oriented software development, enabling the isolation of parts of an application into separate components. Such components can be shared between applications, and since components are only accessed through a rigidly defined interface, their implementation can be changed without impacting applications which use them.

Microsoft's widely used Component Object Model (COM) defines a binary standard for component integration, allowing COM components created using Visual BASIC (for example), to be accessed from an application created using Visual C++.

An enormous number of COM Components have been created and are available for purchase. Indeed, any software which has been recently developed under Microsoft Windows will almost certainly have been created using components.



Modern software design under Microsoft Windows practically mandates that an application be designed using a COM Component based approach. One of the benefits of doing so is that parts of an application's functionality can be made accessible to other applications, not only the same host, but also remotely -- using Distributed COM (DCOM).

The Conflict Between Java and COM

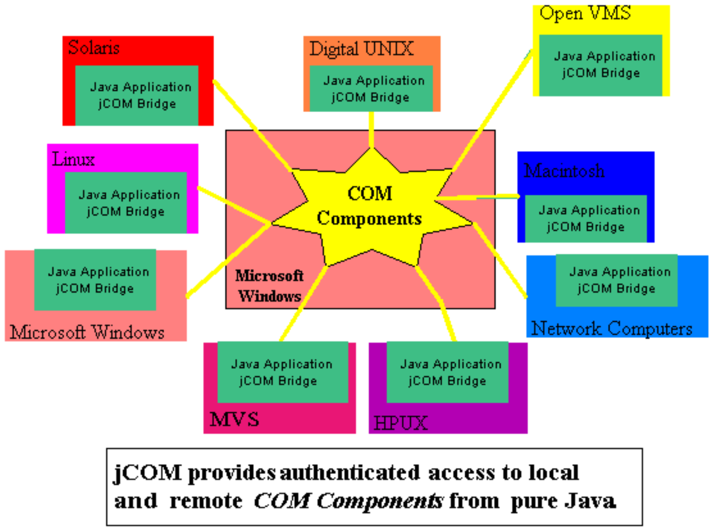
There may be conflict between the desire to create Java software which runs anywhere, and the need to re-use COM software components.

This conflict often manifests itself in companies where there are two clearly divided camps -- the Java developers, who abhor anything which will limit their pure Java software to a specific platform, and the Windows developers, that are urging the use and re-use of COM components.

Nevertheless, there is frequently a business need to access existing applications from new applications developed in pure Java, running in a standard JVM. Many of these existing applications were developed under Microsoft Windows, and because they were well designed, they expose their functionality to other applications by exposing COM Components.

WebLogic jCOM's Solution to the Conflict

BEA WebLogic jCOM offers a solution to the conflict by providing a bridge which allows you the "write once, run anywhere" benefit Java and the benefit of COM component re-use provides.


Solaris
Java Application jCOM Bridge
Digital UNIX
Java Application jCOM Bridge
Open VMS
Java Application jCOM Bridge
Linux
Java Application jCOM Bridge
COM Components
Microsoft Windows
Macintosh
Java Application jCOM Bridge
Java Application jCOM Bridge
Microsoft Windows
Java Application jCOM Bridge
Network Computers
Java Application jCOM Bridge
MVS
Java Application jCOM Bridge
HPUX
jCOM provides authenticated access to local and remote COM Components from pure Java

Internally WebLogic jCOM uses a standard platform independent mechanism which Microsoft has defined for accessing COM Components.

How WebLogic jCOM Works

WebLogic jCOM's Java-COM bridging capabilities allow COM access from and to Java objects that are running on any operating system environment.

COM developers can make callbacks into Java objects. WebLogic jCOM dynamically "remote-enables" any Java object, making all of its public methods and member variables accessible from COM.

To the Java programmer, WebLogic jCOM makes COM components look just like Java objects, presenting COM properties, methods and events as Java properties, methods and events.

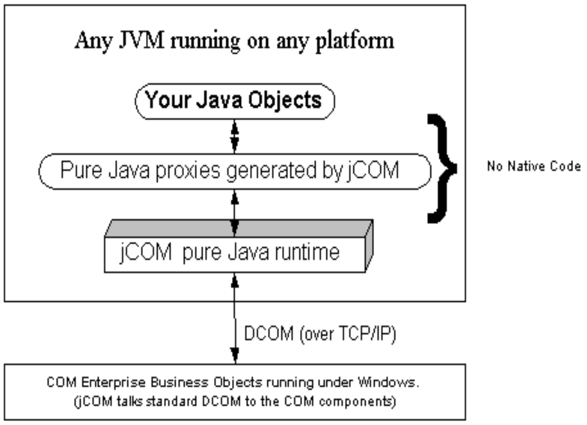

Any JVM running on any platform
Your Java Objects
Pure Java proxies generated by jCOM
jCOM pure Java runtime
No Native Code
DCOM (over TCP/IP)
COM Enterprise Business Objects running under Windows.
(jCOM talks standard DCOM to the COM components)

Figure 1-1 WebLogic jCOM works with any Java Virtual Machine, on any platform, and requires no native code.

WebLogic jCOM's pure Java runtime talks to COM components using Distributed COM layered over Remote Procedure Calls (RPC), which are themselves layered on TCP/IP. So at the lowest level WebLogic jCOM uses the totally standard Java networking classes.

WebLogic jCOM Features

The key features of WebLogic jCOM can be summarized as follows:

- WebLogic jCOM provides a bi-directional bridge between COM components and Java. Enterprise Java Beans (EJBs) and Java objects can be accessed from COM clients; COM components can be accessed from Java clients.

- WebLogic jCOM hides the existence of the data types accessed by the client, dynamically mapping between the most appropriate Java objects and COM components.
- WebLogic jCOM supports both late and early binding.
- All JVMs are supported.
- The machine running the JVM only needs to have the WebLogic jCOM runtime (that is only jCOM.jar) on it. The jCOM.jar file requires no native code, it is pure Java.
- No native code is required on the machine hosting the COM Component. Internally, WebLogic jCOM uses the Windows DCOM network protocol to provide communication between both local and remote COM components and a pure Java environment. WebLogic jCOM does, however, support an optional “native mode” which maximizes performance when running on a Windows platform.
- The WebLogic jCOM Java runtime (jcom.jar) is approximately 595K in size.
- WebLogic jCOM supports event handling. For example, with WebLogic jCOM Java events are accessible from VB using the standard COM event mechanism and also, Java objects can subscribe to COM compnent events.
- WebLogic jCOM lets you access COM Components from Java using no authentication, or with the equivalent of Connect level authentication.

CHAPTER

2 WebLogic jCOM Programming Models

The following sections look at the different ways in which COM-to-Java communication can be implemented using WebLogic jCOM, along with the advantages and disadvantages of each implementation:

- WebLogic WebLogic jCOM Components
- Defining the Terms
- WebLogic WebLogic jCOM Programming Models

WebLogic jCOM Components

BEA WebLogic jCOM allows you to access Java objects (including Java components) running on any Operating System from COM clients running on Microsoft Windows.

In essence, WebLogic jCOM is a kit which provides you with the runtime environment and necessary components to create a Java COM-to-Java bridge. In all the examples we look at, the bridge is on the same machine as the WebLogic Server machine.

The `jCOM.jar` runtime is required to compile and run the bridge. The bridge itself is a Java process, created by the user, which runs in the background to handle the COM to Java communication between the client and the server. The bridge, `JCOMBridge.java,` provided with the examples serves both as a bridge for the examples and a base for creating your own bridge.

In addition to the runtime file, WebLogic jCOM also provides a number of tools and components which are used for configuring the client and server environments.

Note that the Java Virtual Machine always runs as a separate process to the VB process, possibly even on another machine. WebLogic jCOM cannot be used to put a Java bean inside a VB window, since they are in separate processes.You do however have access to all public methods and fields of any public Java class from your Visual Basic programs.

COM-to-Java communication can be achieved using any of the following programming models:

- DCOM Zero Client Programming Model
- DCOM Late Bound Programming Model
- DCOM Early Bound Programming Model
- DCOM Late Bound Encapsulation Programming Model
- Native Late Bound Programming Model
- Native Early Bound Programming Model

Defining the Terms

DCOM Mode

The DCOM (Distributed Component Object Model) mode uses the Component Object Model (COM) to support communication among objects on different computers.

Native Mode

Native mode uses native code (DLLs), which are compiled and optimized specifically for the local operating system and CPU, resulting in better performance.

Zero Client Installation

Zero client installation means that you do not have to install any software on the Windows client machine. There is no deployment overhead.

Late Binding

Late binding is a way of providing access to another application's objects. In late bound access, no information about the object being accessed is available at compile time; the objects being accessed are dynamically evaluated at runtime. This means that it is not until you run the program that you find out if the methods and properties you are accessing actually exist.

Early Binding

Early binding is a way of providing access to another application's objects. Early bound access provides you with information about the object you are accessing while you are compiling your program; all objects accessed are evaluated at compile time. This requires that the server application provide a type library and that the client application identify the library for loading onto the client system.

WebLogic jCOM Programming Models

COM-to-Java communication can be implemented using either DCOM or Native mode. Each of these modes provides a choice of programming model.

In all the implementations, the COM client runs on a Microsoft Windows platform. The Java component runs on a Java™ 2 Platform, Enterprise Edition 1.3 and WebLogic Server 6.1. The client and server/bridge can run on the same system in a one-system implementation.

DCOM Zero Client Programming Model

A DCOM zero client installation is easy to implement. No WebLogic jCOM specific software is required on the client side.

Zero client installation utilizes late binding and therefore provides the same flexibility in terms of changes made to the Java component. However, this implementation requires that the bridge location and port number be hard coded into the COM client, which means that if the bridge location is changed, this reference has to be regenerated and changed in the source code.

The DCOM zero client installation is more error prone than the DCOM early bound implementation.

Relative to the DCOM late bound implementation, DCOM zero client installation provides a gain in performance on initialization at runtime.

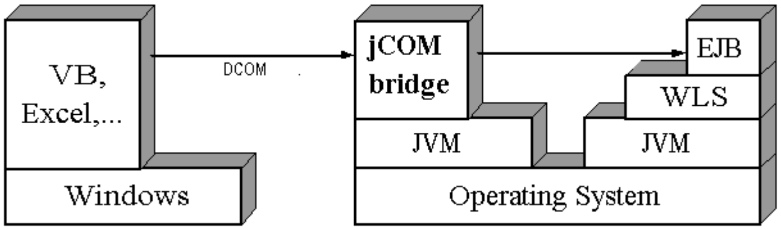


Figure 2-1 Runtime installation for a typical DCOM zero client implementation.

For the client to access the Java component:

1. The WebLogic jCOM bridge location is hardcoded into the client using a `objref` moniker string. The `objref` moniker is generated by the user and it encodes the IP address and port of the WebLogic jCOM bridge. Once the bridge connection is established the client can link a COM object to an interface in the Java component.

2. Any further references in the client to this COM object will use the Java method as if it were a COM method.

For an example of zero client installation implementation see the Zero Client Installation example installed in the `samples/examples` directory of the WebLogic jCOM installation.

DCOM Late Bound Programming Model

DCOM late bound access is easy to implement and it is a flexible implementation since objects referenced are only evaluated at runtime. This however also means that no type checking can be done at compile time, which makes this implementation more error prone.

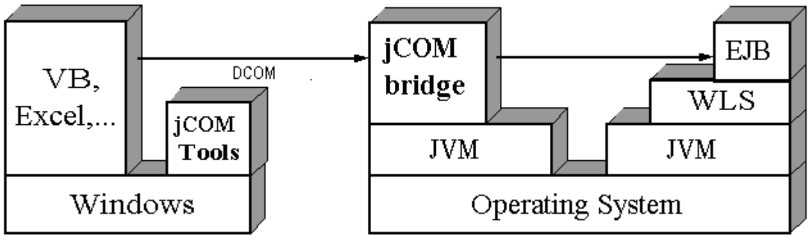


Figure 2-2 Runtime installation for a typical DCOM late bound implementation.

For the client to access the Java component:

1. In the client source code, you link a COM object to an interface in the Java component. For the client to access the interface, you have to use a WebLogic jCOM tool to register the JVM in the Windows registry and associate it with the IP address and port of the WebLogic jCOM bridge.
2. Any further references in the client to this COM object will use the Java method as if it were a COM method.

For an example of a DCOM late bound implementation see the Late Bound Implementation example installed in the `samples/examples` directory of the WebLogic jCOM installation.

DCOM Early Bound Programming Model

DCOM early bound access is complex to implement. It requires the generation of a type library and wrappers. The type library is required on the client side; the wrappers are required on the server side. If the client and user are running on separate machines the type library and wrappers have to be generated on the same machine and then copied to the system where they are required.

Early bound access lacks the flexibility of late bound access in that any changes made to the Java component require regeneration of the wrappers and the library. See Late Bound Encapsulation Programming Model for a hybrid alternative strategy.

Early bound access does provide improved reliability. Compile-time type checking makes debugging easy and the user has the advantage of being able to browse the type library.

Relative to the DCOM late bound implementation, the DCOM early bound implementation provides improved runtime transaction performance, but slower initialization at runtime.

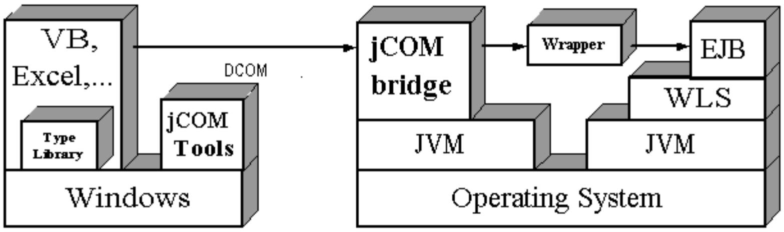


Figure 2-3 Runtime installation for a typical DCOM early bound implementation.

For the client to access the Java component:

1. In the client source code, COM objects are declared using a user generated type library. For the client to access the interface, you have to use a WebLogic jCOM tool to register the JVM in the Windows registry and associate it with the IP address and port of the WebLogic jCOM bridge. The type library is then registered on the client side and linked to the registered JVM name.
2. Any further references in the client to this COM object will use the Java method as if it were a COM method.

For an example of a DCOM early bound implementation see the Early Bound Implementation example installed in the `samples/examples` directory of the WebLogic jCOM installation.

DCOM Late Bound Encapsulation Programming Model

If you have used early binding and plan to alter your Java component interface, or simply wish to be prepared for the future possibility, it is recommended that you employ late bound encapsulation. Any changes made to a Java component requiring a recompilation of the wrappers, will also necessitate creation of a new client-side type library, since wrappers and type library are interdependent and version specific. This situation could arise even if the Java component interface changes are irrelevant to a particular client program. You could therefore be forced to redistribute the new type library to all client systems accessing the Java component.

When using the late bound encapsulation method, you retain the majority of the benefits of early bound programming, while implementing a more flexible late bound design that does not require wrappers or type libraries.

The basic steps described below are for a Visual Basic client but can be applied to any programming language:

1. Create a type library and implement the interface of the objects you want to use inside one or more VB classes.
2. This class can now act as a wrapper for communication with the late bound version of the object.
3. Eliminate the type library once the interface has been established (as long as it still exists the implementation will remain early bound).

Native Late Bound Programming Model

Native late bound access is easy to implement and is a flexible implementation since objects referenced are only evaluated at runtime. This however also means that no type checking can be done at compile time, which makes this implementation more error prone.

Since native libraries have only been created for Windows, implementing native late bound access requires that the WebLogic jCOM bridge be installed on the client machine.

Relative to the DCOM late bound implementation, the native late bound implementation provides a large gain in performance on initialization at runtime.

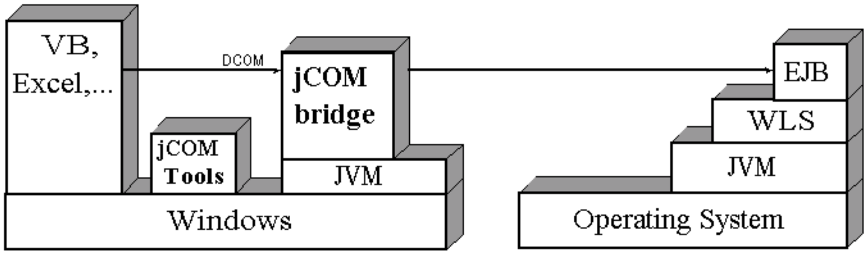


Figure 2-4 Runtime installation for a typical native late bound implementation.

For more on implementing native mode see *Native Mode* in the *Reference Guide*.

Native Early Bound Programming Model

Native early bound access is complex to implement. It requires the generation of a type library and wrappers. The type library is required on the client side; the wrappers are required on the server side. If the client and user are running on separate machines the type library and wrappers have to be generated on the same machine and then copied to the system where they are required.

Early bound access lacks the flexibility of late bound access in that any changes made to the Java component require regeneration of the wrappers and the library.

Early bound access does provide improved reliability. Compile-time type checking makes debugging easy and the user has the advantage of being able to browse the type library.

Since native libraries have only been created for Windows, implementing native early bound access requires that the WebLogic jCOM bridge be installed on the client machine.

Relative to the DCOM early bound implementation; the native early bound implementation provides a large gain in performance on initialization at runtime.

Relative to the native late bound implementation, the native early bound implementation provides slower initialization at runtime, but a slight gain in transaction time performance.

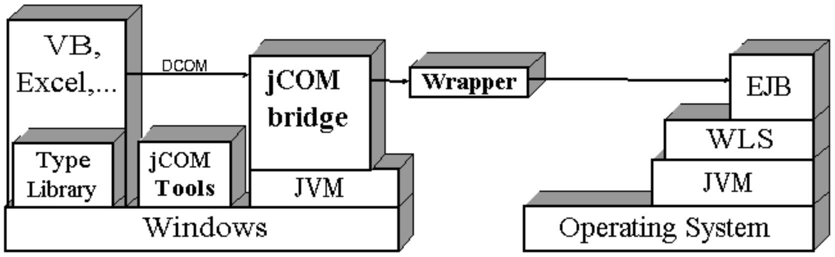


Figure 2-5 Runtime installation for a typical native early bound implementation.

For more on implementing native mode see *Native Mode* in the *Reference Guide*.

3 Programming

WebLogic jCOM uses the Windows Distributed COM (DCOM) network protocol to provide communication between both local and remote COM components and a pure Java environment. The following sections describe what the programmer has to do in order to use WebLogic jCOM to access Java components from a COM client:

- Server-Side Programming Requirements
- Client-Side Programming Requirements
- Intercepting the Instantiation of Java Objects
- Instantiating Java Objects from COM using Constructors
- Implementing a Singleton Java Object

Server-Side Programming Requirements

The programming requirements on the server side are, for most programming models, limited to configuration and compiling and activating the bridge. When implementing an early bound programming model or a late bound encapsulation programming model you have to, in addition, generate wrappers and a type library.

The bridge file, `JCOMBridge.java`, is provided with the WebLogic jCOM installation. This file is used in all the examples provided and can be used as a base for creating your own bridge file, should you wish to do this.

To establish the bridge on the server side, the following needs to be done, irrespective of the programming model being implemented:

1. Start WebLogic Server.

2. Set the PATH environment variables to point to the root directories of your WebLogic Server and JDK 1.3 installations (by running the `setEnv` file).

3. Ensure that the WebLogic Server's listen port (7001 by default) is correctly specified by editing the following line in the bridge file:

   ```
   env.put(Context.PROVIDER_URL, "t3://localhost:7001")
   ```

4. Compile the WebLogic jCOM bridge file by executing the build script `build.xml`. You can do this by running the following command:

   ```
   ant
   ```

5. Configure the WebLogic jCOM environment, by setting the PATH environment variables to point to the root directory of your WebLogic jCOM installation.

6. Invoke the bridge on the server side by using the following statement which runs the bridge file, `JCOMBridge.java`:

   ```
   java -classpath %CLASSPATH% -DJCOM_DCOM_PORT=7050 JCOMBridge
   ```

   where `7050` is the port where the WebLogic jCOM bridge must listen for client-to-server communications.

   The bridge file, `JCOMBridge.java`, defines the WLS TCP/IP address and listen port and registers the JVM name with JNDI.

If you are implementing early binding or late bound encapsulation, the following additional steps are required:

1. Set up the generation environment needed for the execution of the java2com tool and compilation of the bridge and wrappers. The CLASSPATH needs to include a path to the jCOM.jar as well as a path to the Java classes being accessed.

2. Generate Java wrappers and an IDL file for your Java component with the `java2com` tool:

   ```
   java com.bea.java2com.Main
   ```

   When this line is executed, the java2com window will pop up. In the Java Classes & Interfaces field, you must enter the names of the classes you wish to use, including the bridge class.

3. Make sure that the generated classes and JCOMBridge.class file are in your CLASSPATH, then compile the Java wrapper files:

   ```
   javac Output Directory\*.java
   ```

Client-Side Programming Requirements

For the programing steps required on the client system, we will look at how to implement the following programming models:

- DCOM Zero Client Programming Model
- DCOM Late Bound Programming Model
- DCOM Early Bound Programming Model
- DCOM Late Bound Encapsulation Programming Model
- Native Mode Programming Model

DCOM Zero Client Programming Model

The basic client-side programming steps required to implement a DCOM zero client programming model are (described below for a VB client accessing an EJB on the WebLogic Server):

1. Generate a coded reference to the bridge's location (the objref) using the Java class `com.bea.jcom.GetJvmMoniker`. Specify as parameters the full name or TCP/IP address of the server machine and the port where the bridge can be accessed, for example:

   ```
   java com.bea.jcom.GetJvmMoniker mymachine.mycompany.com 7050
   ```

   or

   ```
   java com.bea.jcom.GetJvmMoniker localhost 7050
   ```

2. In the client source code, access the bridge using the following statement which contains the generated objref:

   ```
   Set objBridge = GetObject("objref:generatedobjref")
   ```

3. Following this, all objects can be requested from the server using JNDI, for example:

   ```
   Set objHome = objBridge.get("JVMName:jndi name of ejb")
   ```

DCOM Late Bound Programming Model

The basic client-side programming steps required to implement a DCOM late bound programming model are (described below for a VB client accessing an EJB on the WebLogic Server):

1. In the source code for the VB client, first link a COM object to an interface of the EJB. In this extract from a VB client's source code, notice the declaration of the COM version of the EJB's home interface, `objHome`. This COM object is linked to an instance of the EJB's home interface on the server side.

   ```
   Dim objHome As Object

   Private Sub Form_Load()

   'Handle errors

   On Error GoTo ErrOut

   'Bind the EJB’s HomeInterface object via JNDI

   Set objHome = GetObject("JVMName:jndi name of ejb")
   ```

   `GetObject` is getting an object through JNDI lookup on the WebLogic Server. The JVM ("*`JVMName`*") needs to be registered in the registry, as described in step 3.

2. Any further references to this object appear to be referring to a COM object, but are in fact using the Java methods as if they are COM methods.

3. On the client system use the `regjvm` tool to register the local Java Virtual Machine by adding the name to the Windows registry and associating it with the TCP/IP address and client-to-server communications port where WebLogic jCOM will listen for incoming WebLogic jCOM requests. For example:

   ```
   regjvmcmd JVMName localhost[7050]
   ```

DCOM Early Bound Programming Model

The basic client-side programming steps required to implement a DCOM early bound programming model are (described below for a VB client accessing an EJB on the WebLogic Server):

1. Copy the generated IDL (see *Server-Side Programming Requirements* above) to the client system.

2. Compile the IDL file into a type library using the Microsoft IDL compiler midl.exe:

   `midl` *`generatedIDLFileName`*`.idl`

   The result of the compilation is a type library of the same name, but with the extension `.tlb`.

3. Register the type library and set the JVM it will service, for example:

   `regtlb /unregisterall`

   `regtlb` *`generatedIDLFileName`*`.tlb` *`JVMName`*

   The first line above calls `regtlb.exe` in order to un-register any previously registered type library versions. The second line then registers the newly compiled type library and specifies the name of the JVM ("*`JVMName`*") that will be linked with the type library. The WebLogic jCOM runtime requires this information for linking type library defined object calls to the appropriate wrapper classes.

4. Now the client can access the type library. Load the VB project and in the Projects menu, select Reference. Scroll down until you find the type library and activate its check box. Click OK.

5. Objects are no longer declared "As Object", but rather by using the type library:

   `Dim objCOM As` *`generatedIDLFileName.generated class name`*

   For example, if your fully qualified Java class is *`examples.ejb.basic.containerManaged.AccountHome`*, your generated class name would be *`ExampleEjbBasicContainerManagedAccountHome`*.

6. To access design time information about the various methods and properties of the objects, the following is also required:

   ```
   Dim objTemp As Object

   Dim objBridge As New generatedIDLFileName.COMtoWebLogic

   Set objTemp = GetObject("JVMName:jndi name of ejb")

   Set objHome = objBridge.narrow(objTemp,"fully qualified java
   class")
   ```

Notice the `objTemp` object uses a late bound method to obtain a reference to the EJB object. This late bound object is passed to the bridge's "narrow" method, and is given an early bound object in return.

7. On the client system use the `regjvm` tool to register the local Java Virtual Machine by adding the name to the Windows registry and associating it with the TCP/IP address and client-to-server communications port where WebLogic jCOM will listen for incoming WebLogic jCOM requests:

   ```
   regjvmcmd JVMName localhost[7050]
   ```

DCOM Late Bound Encapsulation Programming Model

Using late bound encapsulation allows you to retain the majority of the benefits of early bound programming, while implementing a more flexible late bound model that does not require wrappers or type libraries.

For example, if you have a Visual Basic client accessing an EJB, you will need to do the following:

1. Copy the generated IDL (see *Server-Side Programming Requirements* above) to the client system.

2. Compile the IDL file into a type library using the Microsoft IDL compiler midl.exe:

   ```
   midl generatedIDLFileName.idl
   ```

   The result of the compilation is a type library of the same name, but with the extension `.tlb`.

3. Register the type library and set the JVM it will service, for example:

   ```
   regtlb /unregisterall
   ```

   ```
   regtlb generatedIDLFileName.tlb JVMName
   ```

   The first line above calls `regtlb.exe` in order to un-register any previously registered type library versions. The second line then registers the newly compiled type library and specifies the name of the JVM (“*`JVMName`*”) that will be linked with the type library. The WebLogic jCOM runtime requires this information for linking type library defined object calls to the appropriate wrapper classes.

4. Reference this type library from within a Visual Basic project, using the Project->References dialog.
5. Add an empty class module to the project and open its source window.
6. Add a module level variable of type "Object" that will be used to reference your EJB.
7. Using the "object" and "procedure" pull-down menus at the top of the class module's source window, add a `Class_Initialize` method, and place within the method the source code required to assign your module level object variable a reference to your EJB. See DCOM Late Bound Programming Model for the necessary initialization source.
8. Use the `Implements` keyword at the top of your class source to implement the interface of any objects you'll need to access. You can reference your EJB objects using the name of your type library, followed by a dot. (i.e. `Implements` *`generatedIDLFileName.generated class name`*)

   For example, if your fully qualified Java class is *`examples.ejb.basic.containerManaged.AccountHome`*, your *`generated class name`* would be `ExampleEjbBasicContainerManagedAccountHome`*.*
9. Use the "object" and "procedure" pull-downs to select all of the methods and properties from the EJB objects that you'll need to access. This should produce the skeleton source code for the selected methods and properties.
10. Within these method and property declarations, insert late bound code that accesses the EJB's methods and properties through the module level object you've created.
11. Delete the "Implements" entries from the top of your class source, and remove the type library reference from the Project->References dialog. Your class no longer depends on the type library (and hence, the wrappers on the server) for access to the EJB.

Once you have done this, you can instantiate an instance of the class you've created, and access all of your EJB functionality as though it were early bound. Any changes to the EJB will not affect your VB project so long as the interface remains static for the methods and properties employed in the VB client source.

Native Mode Programming Model

In native mode a COM client accesses a Java object running on the same machine as the client. WebLogic jCOM uses native code to facilitate the interaction. For more on native mode see *Native Mode* in the *Reference Guide*.

Intercepting the Instantiation of Java Objects

If you wish to control the instantiation of Java objects, create a class which implements the `com.bea.jcom.Instanciator` interface. This interface has one method, which looks like this:

```
public Object instanciate(String javaClass) throws
com.bea.jcom.AutomationException;
```

Pass a reference to your instantiator as a second parameter when calling Jvm.register(...):

```
com.bea.jcom.Jvm.register("MyJvm", myInstanciator);
```

The default instantiator used by WebLogic jCOM looks like this:

```
public final class DefaultInstanciator implements
com.bea.jcom.Instanciator {
public Object instanciate(String javaClass)
throws com.bea.jcom.AutomationException {
try {
return Class.forName(javaClass).newInstance();
} catch(Exception e) {
e.printStackTrace();
throw new AutomationException(e);
}
}
}
```

For example this is a VB to EJB bridge (based on Sun's JNDI Tutorial):

```
import javax.naming.*;
import java.util.Hashtable;
import com.bea.jcom.*;

public class VBtoEJB {
public static void main(String[] args) throws Exception {
Jvm.register("ejb", new EjbInstanciator());
Thread.sleep(10000000);
}
}

class EjbInstanciator implements Instanciator {
Context ctx;

EjbInstanciator() throws NamingException {
Hashtable env = new Hashtable(11);
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://... TBS ...");
ctx = new InitialContext(env);
}

public Object instanciate(String javaClass) throws
AutomationException {
try {
try {
return Class.forName(javaClass).newInstance();
} catch(Exception e) {}
return ctx.lookup(javaClass);
} catch (Throwable t) {
t.printStackTrace();
throw new AutomationException(new Exception("Unexpected: " + t)); }
}
}
```

If you compiled the above, and ran it on a machine (`development.company.com`) like this:

```
java -DJCOM_DCOM_PORT=4321 VBtoEJB
```

Then on a Windows machine you used the WebLogic jCOM `regjvmcmd` command like this:

```
regjvmcmd ejb development.company.com[4321]
```

Then from VB you would then use:

```
Set myEjb = GetObject("ejb:cn=ObjectName")
MsgBox myEjb.someProperty
myEjb.myMethod "a parameter"
```

Instantiating Java Objects from COM using Constructors

COM has no concept of constructors. One method is to define a default constructor, and then define a static member which takes the appropriate parameters and instantiates the object and returns it:

```
public class MyClass {
       public MyClass() {}
       public MyClass(String p1, int p2, double p3) {
       ...
       }

public static MyClass createMyClass(String p1, int p2, double p3) {
       return new MyClass(p1, p2, p3);
}
}
```

Another possibility is to use WebLogic jCOM's instantiation interception capability -- when you register the JVM you can pass a reference to an object whose class implements a special Java jCOM interface which is called when you use `GetObject("MyJvm:MyClass")` -- you get passed everything after the colon, so you could actually do: `GetObject("MyJvm:MyClass(1, 2, three, 4.0")` and then in the interceptor parse the string that is passed in, and invoke the appropriate constructor.

Implementing a Singleton Java Object

In COM terminology an object is a singleton if there exists only one instance of the object at any time. Each time you call CreateInstance you obtain a reference to the same object. This ensures that all clients access the same instance.

By controlling the instantiation of Java objects you can implement a singleton Java object which is accessible from COM clients. Here is an example of what the instantiator would look like for a class called `mySingletonClass`:

```
import java.util.*;
import com.bea.jcom.*;

public class COMtoJava {
public static void main(String[] args) throws Exception {
try {
Jvm.register("MyJvmId", new
SingletonInstanciator("MySingletonClass"));

while (true) { // wait forever
Thread.sleep(100000);
}
} catch (Exception e) {
System.out.println(e.getMessage());
e.printStackTrace();
}
}
}

class SingletonInstanciator implements Instanciator {
String singletonClassname;
static Object singletonObject = null;

SingletonInstanciator(String singletonClassname) {
try {
this.singletonClassname = singletonClassname;
if (singletonObject == null) {
System.out.println("SingletonInstanciator: creating the singleton
[" + singletonClassname + "]");
// initialize the singleton
Class classObject = Class.forName(singletonClassname);
singletonObject = classObject.newInstance();
}
} catch (Exception e) {
System.out.println(e.getMessage());
e.printStackTrace();
}
}

public Object instanciate(String javaClass) throws
AutomationException {
try {
System.out.println("instanciate for " + javaClass);

// if request is to create the singleton, just return the existing
instance.
if (javaClass.equals(singletonClassname)) {
return singletonObject;
} else {
Class classObject = Class.forName(javaClass);
return classObject.newInstance();
}
}
catch (Exception e)
{
System.out.println("Failed to instanciate class " + javaClass);
System.out.println(e.getMessage());
e.printStackTrace();
System.out.println("Throwing exception back to caller.");
throw new AutomationException(e);
}
}
}
```

And here is a sample `MySingletonClass` implementation:

```
public class MySingletonClass {
public MySingletonClass() {
System.out.println("MySingletonClass constructor called.");
}

public int Method1(int val) {
return val + 1;
}
}
```

If you compiled both of the above, and ran COMtoJava on a machine (development.company.com) like this:

```
java -DJCOM_DCOM_PORT=4321 COMtoJava
```

Then on a Windows machine you used the WebLogic jCOM `regjvmcmd` command like this:

```
regjvmcmd MyJvmId development.company.com[4321]
```

Then from VB you would then use:

```
Set objMySingleton1 = GetObject("MyJvmId:mySingletonClass")
Set objMySingleton2 = GetObject("MyJvmId:mySingletonClass")
MsgBox objMySingleton1 & objMySingleton2
```

Which would create two references to the same object.

4 Deploying your Application

The following sections provide information about the steps that need to be taken before you can run your application:

- Deployment Options
- Deploying your Application

Deployment Options

When using WebLogic jCOM to access Java objects from a COM client, there are five different deployment scenarios, depending on the implementation used:

- Deploying a DCOM Zero Client Implementation
- Deploying a DCOM Late Bound Implementation
- Deploying a DCOM Early Bound Implementation
- Deploying a DCOM Late Bound Encapsulation Implementation
- Deploying a Native Mode Implementation

In addition, your deployment may be influenced by the use of several WebLogic Servers in

- Server Clustering

If you chose the appropriate installation for the implementation you are using, all the WebLogic jCOM files required for configuration, deployment and at runtime will already be in the correct location. You may still have to copy additional user-generated files to the correct location.

For more information on deployment when using WebLogic jCOM for accessing COM objects from a Java client, see the WebLogic jCOM Reference Guide.

Deploying your Application

Deploying a DCOM Zero Client Implementation

A DCOM zero client implementation requires zero deployment on the client system. Take note, though, that the WebLogic jCOM bridge is accessed from the client via a hardcoded reference to the server's location (IP and port). This reference is in the client source code. Should the server location change, this `objref` will have to be regenerated and inserted into the client source in order for the COM client to be able to communicate with the server.

The server installation process (typically a *server install*) will have installed the WebLogic jCOM components required on the server at runtime:

- WebLogic jCOM bridge: `jcom.jar`

Once the EJB has been deployed to WebLogic Server and the server has been activated, the compiled bridge can be activated. Any changes made to the server’s TCP/IP address and listen port must be reflected in the bridge file itself.

For more on how to generate the `objref` and activate the bridge, see the Zero Client Installation example in the *Programming* chapter.

Deploying a DCOM Late Bound Implementation

For a DCOM late bound implementation, the client installation process (typically a *client install*) will have installed the WebLogic jCOM components required on the client system at runtime:

- WebLogic jCOM moniker: `JintMk.dll`

as well as those required for configuring the client:

- `regjvm`

Before running the application, the JVM must be registered using `regjvm` or `regjvmcmd`. For more on the tools, `regjvm` and `regjvmcmd,` see *jCOM Tools*.

Note: Should the server location change, you will have to re-register the JVM. Before doing this you have to un-register the old entry because the `regjvmcmd` tool does not overwrite old entries when new entries with identical names are entered. You can un-register the old entry using the command line tool `regjvmcmd`, or by using the GUI tool `regjvm` (both can be found in the `jCOM\bin` directory).

The server installation process (typically a *server install*) will have installed the WebLogic jCOM bridge files required on the server at runtime:

- WebLogic jCOM bridge: `jcom.jar`

Once the EJB has been deployed to WebLogic Server and the server has been activated, the compiled bridge can be activated. Any changes made to the server's TCP/IP address and listen port must be reflected in the bridge file itself.

For more on how to register the JVM and activate the bridge see the Late Bound Implementation example in the *Programming* chapter.

Deploying a DCOM Early Bound Implementation

For a DCOM early bound implementation, the client installation process (typically a *client install*) will install the WebLogic jCOM tools required on the client system at runtime:

- WebLogic jCOM moniker: `JintMk.dll`

as well as those required for configuring the client:

- `regtlb`
- `regjvm`

You must ensure that the type library generated by the `java2com` tool is on the client system at runtime and that the `CLASSPATH` environment variable points to the directory in which it is placed. The type library is needed for early binding of the remote object acquired from the WebLogic jCOM bridge. Use the `regtlb` tool to register the type library on the client. For more about the tools, `java2com` and `regtlb`, see *jCOM Tools*.

Before running the application, the JVM must be registered using `regjvm` or `regjvmcmd`. For more on the tools `regjvm` and `regjvmcmd`, see *jCOM Tools*.

Note: Should the server location change, you will have to re-register the JVM. Before doing this you have to un-register the old entry because the `regjvmcmd` tool does not overwrite old entries when new entries with identical names are entered. You can un-register the old entry using the command line tool `regjvmcmd`, or by using the GUI tool `regjvm` (both can be found in the `jCOM\bin` directory).

The server installation process (typically a *server install*) will install the WebLogic jCOM bridge files required on the server at runtime:

- WebLogic jCOM bridge: `jcom.jar`

You must ensure that the wrapper classes generated by the `java2com` tool are on the server system at runtime and that the `CLASSPATH` environment variable points to the directory in which they are placed. The wrapper classes enable early bound communications with the Java objects they were created to encapsulate.

Once the EJB has been deployed to WebLogic Server and the server has been activated, the compiled bridge can be activated. Any changes made to the server’s TCP/IP address and listen port must be reflected in the bridge file itself.

For more on how to register the type library and the JVM and activate the bridge see the Early Bound Implementation example in the Programming chapter of the WebLogic jCOM User Guide.

Deploying a DCOM Late Bound Encapsulation Implementation

Since late bound encapsulation uses early binding during development and late binding during runtime, deploying a late bound encapsulation implementation is much the same as deploying a late bound implementation. To ensure that late binding is indeed implemented at runtime, you need to be sure that the type library referenced during development is no longer accessible to the client. This will also render the wrappers on the server side inaccessible.

For a late bound encapsulation implementation, the client installation process (typically a *client install*) will install the WebLogic jCOM tools required on the client system at runtime:

- WebLogic jCOM moniker: `JintMk.dll`

as well as those required for configuring the client:

- `regjvm`

You must ensure that the type library referenced during development is not visible to the client at runtime, otherwise the implementation will remain early bound.

Before running the application, the JVM must be registered using `regjvm` or `regjvmcmd`. For more on the tools, `regjvm` and `regjvmcmd,` see *jCOM Tools*.

Note: Should the server location change, you will have to re-register the JVM. Before doing this you have to un-register the old entry because the `regjvmcmd` tool does not overwrite old entries when new entries with identical names are entered. You can un-register the old entry using the command line tool `regjvmcmd`, or by using the GUI tool `regjvm` (both can be found in the `jCOM\bin` directory).

The server installation process (typically a *server install*) will install the WebLogic jCOM bridge files required on the server at runtime:

- WebLogic jCOM bridge: `jcom.jar`

Once the EJB has been deployed to WebLogic Server and the server has been activated, the compiled bridge can be activated. Any changes made to the server's TCP/IP address and listen port must be reflected in the bridge file itself.

Deploying a Native Mode Implementation

In native mode a COM client accesses a Java object running on the same machine as the client. WebLogic jCOM uses native code to facilitate the interaction. For more on native mode see *Native Mode* in the *Reference Guide*.

Server Clustering

If you are using more than one WLS machine in a server cluster, your deployment mechanism will likely be affected. In order to maintain fail-over capabilities, it is recommended that you deploy all WebLogic jCOM files on the client system (similar to native mode deployment above). Placing the WebLogic jCOM bridge on a single WLS machine within your cluster will render your system susceptible to failure; if the bridge machine fails you lose all WebLogic jCOM functionality. Placing the WebLogic jCOM bridge on the client system removes this possibility, allowing your cluster to employ its fail-over capabilities as normal.

5 WebLogic jCOM Tools

The following sections describe the most commonly used WebLogic jCOM tools:

- The regjvm GUI Tool
- The regjvmcmd Command Line Tool
- The java2com Tool
- The regtlb Tool

The regjvm GUI Tool

WebLogic jCOM allows languages supporting COM to access Java objects as though they were COM objects.

To do this you need to register (on the COM client machine) a reference to the JVM in which the Java objects run. The `regjvm` tool enables you to create and manage all the JVM references on a machine.

Note: The `regjvm` tool does not overwrite old entries when new entries with identical names are entered. This means that if you ever need to change the hostname or port of the machine with which you wish to communicate, you have to unregister the old entry. You can do this using the command line tool `regjvmcmd.exe`, or by using the GUI tool `regjvm.exe` (both can be found in the `jCOM\bin` directory).

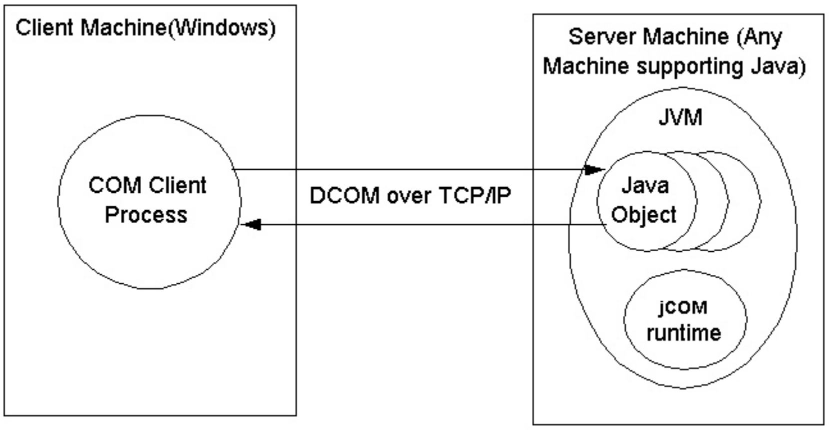
JVM modes

A JVM may be accessed from COM clients using one of three different modes:

- DCOM mode
- Native mode (out of process)
- Native mode in process

DCOM mode

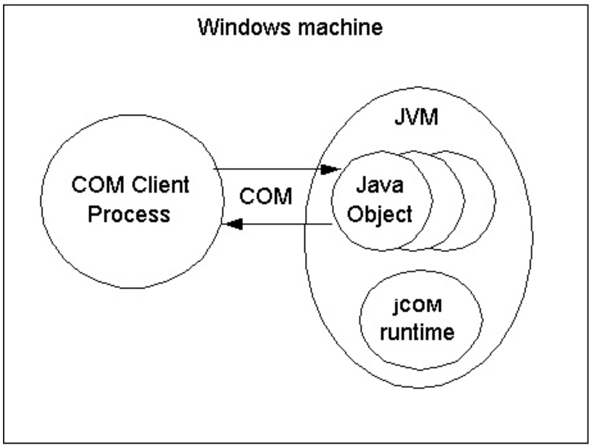
DCOM mode does not require any native code on the Java server side, which means your Java code may be located on a Unix machine or any machine with a Java Virtual Machine installed. When you register the JVM on the Windows client machine you define the name of the server host machine (it may be localhost for local components) and a port number.



The Java code in the JVM must call com.bea.jcom.Jvm.register("<jvm id>"), where <jvm id> is the id of the JVM as defined in regjvm. The JVM must also be started with the JCOM_DCOM_PORT property set to the port defined in regjvm tool for the specified <jvm id>.

Native mode (out of process)

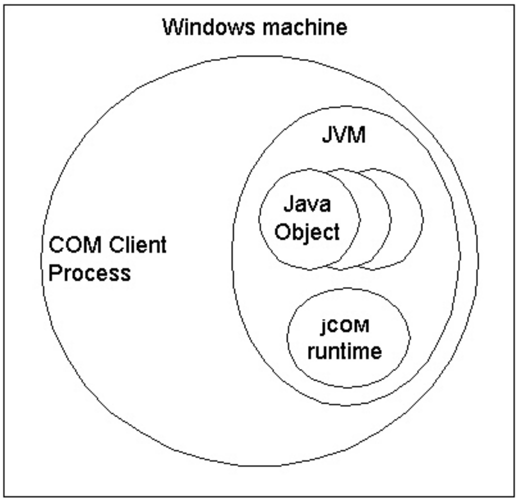
Native mode currently only works on the local machine. Other than the JVM name no additional parameters are necessary.



The JVM must call com.bea.jcom.Jvm.register("<jvm id>"), where <jvm id> is the id of the JVM as defined in regjvm. The JVM must also be started with the JCOM_NATIVE_MODE property set.

Native mode in process

Using native mode in process allows the user to actually load the Java object into the same process as the COM client. Both objects must of course be located on the same machine.



The JVM need not call com.bea.jcom.Jvm.register() or be started as an extra process to the client.

The User Interface of the regjvm GUI Tool

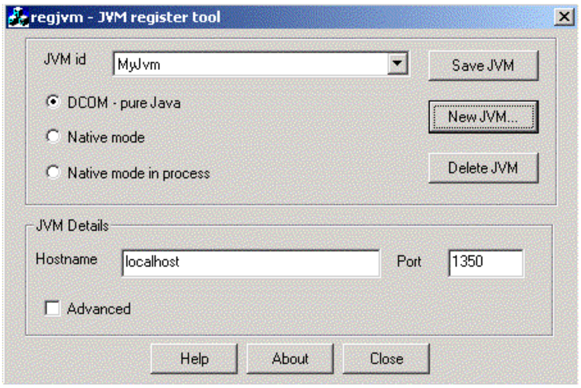
When you run the `regjvm` tool, a dialog is displayed. The dialog is split into two parts:

- The top part is for selection and management of all JVMs on the current machine. JVMs may be added, altered or deleted. Before switching to a different JVM, changes made to the currently selected JVM must be saved. It is also here that the different JVM modes can be selected which then dictates the information required in the lower half of the window.
- The lower half of the windows contains the details required for each JVM, according to the mode of the JVM. In addition to the JVM details there is an advanced checkbox which when selected displays advanced options for each JVM mode.

  The meanings of these options are discussed in the following sections.

DCOM Mode Options for the regjvm GUI Tool

Standard Options



JVM id: MyJvm

[✓ DCOM - pure Java] Native mode / Native mode in process

JVM Details

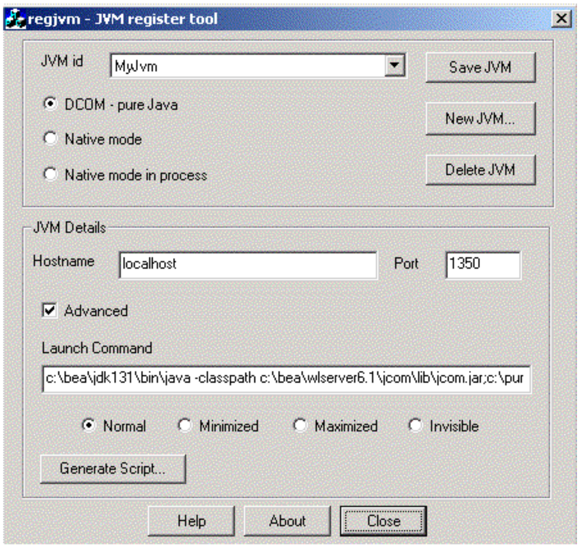
Hostname: localhost

Port: 1350

Advanced: ☐

- **Hostname** (required) - The IP name or IP address where the JVM is located.
- **Port** (required) - The port used to initiate contact with the JVM.

Advanced Options



JVM id: MyJvm

DCOM - pure Java [✓] ; Native mode ; Native mode in process

JVM Details

Hostname: localhost

Port: 1350

Advanced [✓]

Launch Command

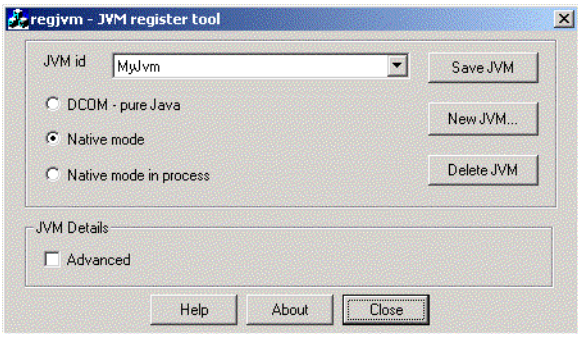
```
c:\bea\jdk131\bin\java -classpath c:\bea\wlserver6.1\jcom\lib\jcom.jar;c:\pur
```

[✓ Normal] Minimized Maximized Invisible

- **Launch command** (optional) - The command to be used if the JVM is to be automatically launched. Typically this would be something like: c:\bea\jdk131\bin\java -classpath c:\bea\wlserver6.1\jcom\lib\jcom.jar; c:\pure MyMainClass.
- **Launch options** (optional) - Allows you to specify the initial window state of the server component.
- **Generate Script...** (optional) - Allows the user to generate a registry script selecting the settings of the JVM.

Native Mode Options for the regjvm GUI Tool

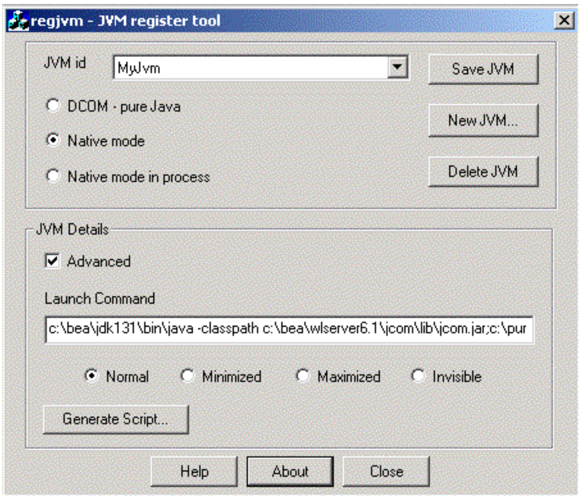
Standard Options



JVM id: MyJvm
[ ] DCOM - pure Java [✓ Native mode] [ ] Native mode in process
[ ] Advanced

There are no standard options for this mode.

Advanced Options



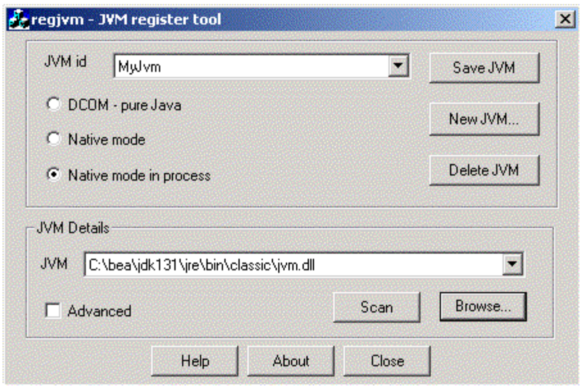
JVM id: MyJvm
[ ] DCOM - pure Java [✓ Native mode] [ ] Native mode in process
[✓ Advanced]
Launch Command: c:\bea\jdk131\bin\java -classpath c:\bea\wlserver6.1\jcom\lib\jcom.jar;c:\pur
[✓ Normal] Minimized Maximized Invisible

- **Launch command** (optional) - see DCOM mode.

- **Launch options** (optional) - see DCOM mode.
- **Generate Script...** (optional) - see DCOM mode.

Native Mode in Process Options for the regjvm GUI Tool

Standard Options



JVM id: MyJvm

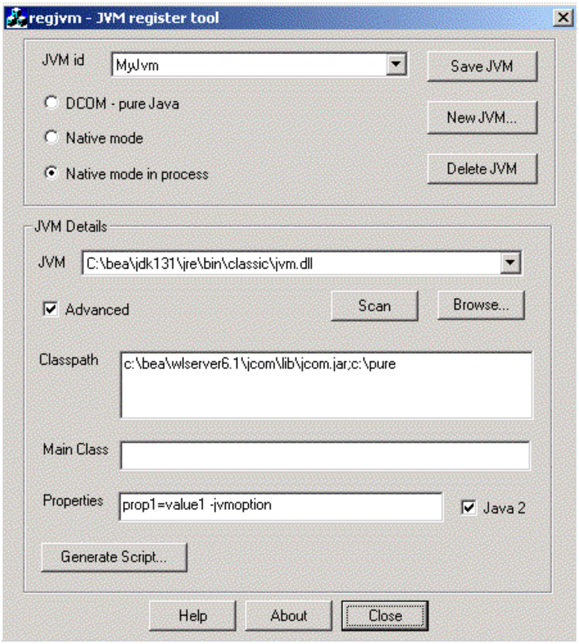
( ) DCOM - pure Java
( ) Native mode
[✓ Native mode in process]

JVM: C:\bea\jdk131\jre\bin\classic\jvm.dll

[ ] Advanced

- **JVM** (required) - The JVM must be specified. Clicking the browse button allows you to select your own JVM, clicking the Scan button scans your local machine for JVMs (this may take a few minutes) and inserts them in the listbox for your selection.

Advanced Options



JVM id: MyJvm

( ) DCOM - pure Java
( ) Native mode
(●) Native mode in process

JVM Details

JVM: C:\bea\jdk131\jre\bin\classic\jvm.dll

☑ Advanced

Classpath: c:\bea\wlserver6.1\jcom\lib\jcom.jar;c:\pure

Main Class:

Properties: prop1=value1 -jvmoption

☑ Java 2

- **Classpath** (optional) - The CLASSPATH for the JVM - if this is left blank the CLASSPATH environment variable at runtime is used. Otherwise the contents are added to the CLASSPATH environment variable.
- **Main class** (optional) - The name of the class containing a Main method which you wish to be called.
- **Properties** (optional) - Any properties which you require to be set. Must have the following syntax: prop1=value1 prop2=value2...
- **Java 2** (optional) - When setting properties this must be set when using Java 2 (JDK 1.2.x, 1.3.x) and cleared when using 1.1.x.
- **Generate Script...** (optional) - see DCOM mode.

The regjvmcmd Command Line Tool

`regjvmcmd` is the command line version of the GUI tool, `regjvm,` discussed above. To get a summary of its parameters, run `regjvmcmd` without parameters.

In its simplest form, you specify:

- a jvm ID (corresponding to the name used in *com.bea.jcom.Jvm.register("JvmId")*),
- and the binding that can be used to access the JVM, in the form *hostname[port]*, where *hostname* is the name of the machine running the JVM, and *port* is the TCP/IP port specified when starting the JVM by setting the JCOM_DCOM_PORT property (e.g. *java -DJCOM_DCOM_PORT=1234 MyMainClass*).

If you no longer need to have the JVM registered, or if you wish to change its registration, you must first un-register it using *regjvmcmd /unregister JvmId*

The java2com Tool

The `java2com` tool analyzes Java classes (using Java's *reflection* mechanism), and outputs:

- a COM Interface Definition Language (IDL) file
- pure Java DCOM marshalling code (wrappers) used by the WebLogic jCOM runtime to facilitate access to the Java objects from COM using vtable (late binding) access.

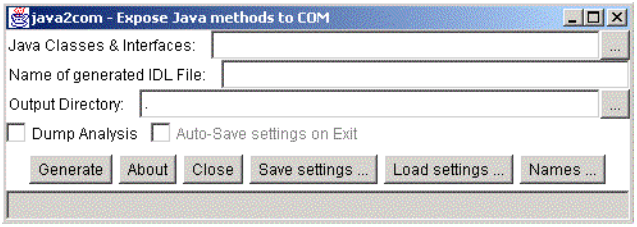
The IDL file has to be compiled using Microsoft's `MIDL` tool.

To generate the IDL file and the wrappers, start the `java2com` tool using the command:

```
java com.bea.java2com.Main
```

You can run this tool on any platform. Make sure that the WebLogic jCOM runtime `jCOM.jar` is in your `CLASSPATH` environment variable.

The `java2com` tool displays the following dialog box:


java2com - Expose Java methods to COM
Java Classes & Interfaces:
Name of generated IDL File:
Output Directory:
.
Dump Analysis
Auto-Save settings on Exit
Generate
About
Close
Save settings ...
Load settings ...
Names ...

The dialog box has the following fields (any changes to the configuration are automatically saved when you exit the dialog box):

1. Java Classes & Interfaces

   These are the 'root' Java classes and interfaces that you want `java2com` to analyze. They must be accessible in your `CLASSPATH`. WebLogic jCOM analyzes these classes, and generates COM IDL definitions and Java DCOM marshalling code which can be used to access the Java class from COM. It then performs the same analysis on any classes or interfaces used in parameters or fields in that class, recursively, until all Java classes and interfaces accessible in this manner have been analyzed.

   Separate the names with spaces. Click on the ... button to display a dialog that lists the classes and lets you add/remove from the list.

2. Name of Generated IDL File

   This is the name of the COM Interface Definition Language (IDL) file which will be generated. If you specify `myjvm`, then `myjvm.idl` will be generated. This name is also used for the name of the type library generated when you compile `myjvm.idl` using Microsoft's MIDL compiler.

3. Output Directory

   The directory to which `java2com` should output the files it generates. The default is the current directory (".").

4. Dump Analysis

   Displays the classes that the `java2com` discovers, as it discovers them.

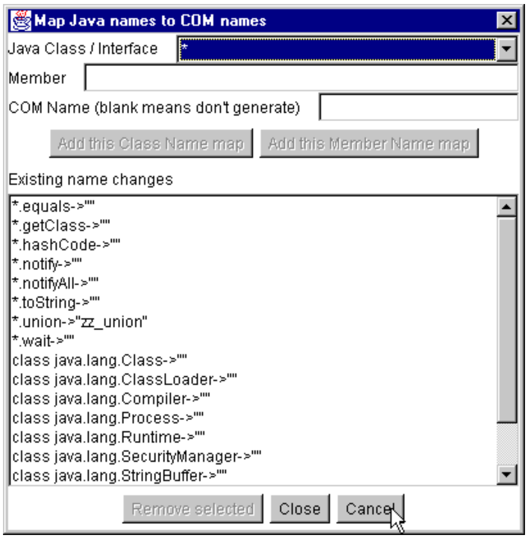
5. Save Settings/Load Settings

   Click on the Save Settings button to save the current `java2com` settings.

When `java2com` starts, it checks to see if there is a `java2com.ser` setting file in the current directory. If present, it loads the settings from that file automatically.

6. Names...

   Clicking the Names button displays the following dialog box:



```
*
*.equals->""
*.getClass->""
*.hashCode->""
*.notify->""
*.notifyAll->""
*.toString->""
*.union->"zz_union"
*.wait->""
class java.lang.Class->""
class java.lang.ClassLoader->""
class java.lang.Compiler->""
class java.lang.Process->""
class java.lang.Runtime->""
class java.lang.SecurityManager->""
class java.lang.StringBuffer->""
```

   When '*' is selected from the class/interfaces names drop-down list, a text box is displayed into which you can type the name of a member (field or class) name. You may specify a corresponding COM name to be used whenever that member name is encountered in any class or interface being generated. If you leave the name blank then that Java member will not have a corresponding member generated in any COM interface.

   When a specific COM class name or interface is selected from the class/interfaces names drop-down list, the set of members in that class or interface is listed below it. You may specify a COM name to be used, and by clicking on *Add this Class Name map* you map the selected class/interface to the specified COM name. By clicking on *Add this Member Name map* you may map the selected member to the specified COM name.

7. Generate button

   Generates the wrappers and IDL file.

For each public Java interface that `java2com` discovers, it creates a corresponding COM interface definition. If the Java interface name were: *com.bea.finance.Bankable*, then the generated COM interface would be named *ComBeaFinanceBankable*, unless you specify a different name using the 'Names ...' dialog.

For each public Java class that `java2com` discovers, it creates a corresponding COM interface definition. If the Java class name were: *com.bea.finance.Account*, then the generated COM interface would be named *IComBeaFinanceAccount*, unless you specify a different name using the 'Names ...' dialog. In addition if the Java class has a public default constructor, then 'java2com' generates a COM class *ComBeaFinanceAccount*, unless you specify a different name using the 'Names ...' dialog.

If a Java class can generate Java events, then the generated COM class will have source interfaces (COM events) corresponding to the events supported by the Java class.

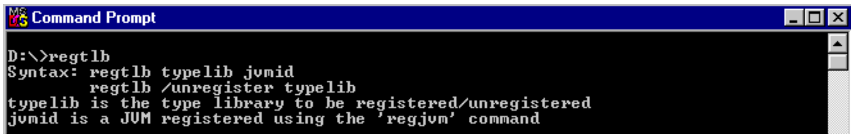
Compile the generated IDL file using Microsoft's MIDL tool. This ships with Visual C++, and can be downloaded from the MS web site. The command

```
midl prodServ.idl
```

will produce a type library called `prodServ.tlb`, which can be registered as described in the following section.

The regtlb Tool

WebLogic jCOM's `regtlb` tool registers a type library on a COM Windows client that wishes to access Java objects using COM's early binding mechanism. `regtlb` takes two parameters. The first is the name of the type library file to be registered. The second is the ID of the JVM in which the COM classes described in the type library are to be found:

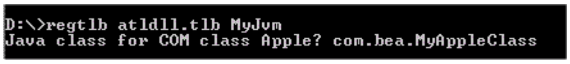


```
D:\>regtlb
Syntax: regtlb typelib jvmid
        regtlb /unregister typelib
typelib is the type library to be registered/unregistered
jvmid is a JVM registered using the 'regjvm' command
```

If the type library was generated from an IDL file that was in turn generated by the WebLogic jCOM `java2com` tool, then the `regtlb` command can automatically determine the Java class name corresponding to each COM class in the type library (the COM class descriptions in the type library are of the form:

```
Java class java.util.Observable (via jCOM))
```

If the type library was not generated from a `java2com` generated IDL file, you will be prompted to give the name of the Java class which is to be instantiated for each COM class:



```
D:\>regtlb atldll.tlb MyJvm
Java class for COM class Apple? com.bea.MyAppleClass
```

This means that when someone attempts to create an instance of `Atldll.Apple`, WebLogic jCOM will instantiate `com.bea.MyAppleClass` in the JVM `MyJvm`. The `MyAppleClass` class should implement the Java interfaces generated by WebLogic jCOM's `java2com` tool from `atldll.tlb` that are implemented by the COM class `Atldll.Apple`.