Oracle® Fusion Middleware

Programming JCOM for Oracle WebLogic Server

11*g* Release 1 (10.3.6)

E13725-06

November 2011

This document describes developing and configuring applications that include WebLogic Server Java Component Object (jCOM).

ORACLE®

Oracle Fusion Middleware Programming JCOM for Oracle WebLogic Server, 11*g* Release 1 (10.3.6)

E13725-06

Contents

4   Calling into a COM Application from WebLogic Server

5   A Closer Look at the jCOM Tools

Preface

This preface describes the document accessibility features and conventions used in this guide—*Programming JCOM for Oracle WebLogic Server*.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction and Roadmap

This chapter describes the contents and organization of this guide—*Programming JCOM for Oracle WebLogic Server*.

- Section 1.1, "Document Scope and Audience"
- Section 1.2, "Guide to This Document"
- Section 1.3, "Related Documentation"
- Section 1.4, "New and Changed Features"

1.1 Document Scope and Audience

This document is a resource for software developers who want to develop and configure applications that include WebLogic Server Java Component Object (jCOM). It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server or considering the use of WebLogic Server jCOM for a particular application

The topics in this document are relevant during the design and development phases of a software project. The document also includes topics that are useful in solving application problems that are discovered during test and pre-production phases of a project.

This document does not address production phase administration, monitoring, or performance tuning jCOM topics. For links to WebLogic Server documentation and resources for these topics, see Section 1.3, "Related Documentation."

It is assumed that the reader is familiar with Java EE and jCOM concepts. This document emphasizes the value-added features provided by WebLogic Server jCOM and key information about how to use WebLogic Server features and facilities to get a jCOM application up and running.

1.2 Guide to This Document

- This chapter, Chapter 1, "Introduction and Roadmap," describes the scope and organization of this guide.
- Chapter 2, "Understanding WebLogic jCOM," provides an overview of the Java to COM Service. It also describes WebLogic jCOM components and features.
- Chapter 3, "Calling into WebLogic Server from a COM Client Application," describes how to access WebLogic Server from a COM client application using WebLogic Server jCOM.

- Chapter 4, "Calling into a COM Application from WebLogic Server," describes how to access a COM client application from WebLogic Server using WebLogic Server jCOM.

- Chapter 5, "A Closer Look at the jCOM Tools," describes how to programatically manage your jCOM applications using value-added WebLogic jCOM tools.

1.3 Related Documentation

This document contains information about configuring and managing jCOM resources.

For jCOM information as it relates to WebLogic Server, see the following document:

- Securing WebLogic Resources for information about COM resources.

1.4 New and Changed Features

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.

2

Understanding WebLogic jCOM

This chapter provides an overview of WebLogic jCOM. WebLogic jCOM is a software bridge that allows bidirectional access between Java/Java EE objects deployed in WebLogic Server, and Microsoft ActiveX components available within Microsoft Office family of products, Visual Basic and C++ objects, and other Component Object Model/Distributed Component Object Model (COM/DCOM) environments.

- Section 2.1, "What Is WebLogic jCOM?"
- Section 2.2, "Why Use WebLogic jCOM?"
- Section 2.3, "WebLogic jCOM Features"
- Section 2.4, "Planning Your WebLogic jCOM Application"
- Section 2.5, "jCOM Features and Changes in this Release"

2.1 What Is WebLogic jCOM?

In general, Oracle believes that Web Services are the preferred way to communicate with Microsoft applications. We suggest that customers plan to migrate legacy COM applications to .NET in order to leverage this type of communication. jCOM is provided as a migration path for interim solutions that require Java-to-COM integration. It is suitable for small projects or bridge solutions.

Unlike other Java-to-COM bridges available on the market, jCOM is specifically designed to work with WebLogic Server on the Java side. You cannot use jCOM to make COM objects communicate with any arbitrary Java Virtual Machine (JVM). In addition, jCOM makes direct use of WebLogic Server threads, providing a very robust way to expose services to COM objects.

WebLogic jCOM makes the differences between the object types transparent: to a COM client, WebLogic Server objects appear to be COM objects and to a WebLogic Server application, COM components appear to be Java objects.

WebLogic jCOM is bidirectional because it allows:

- Microsoft COM clients to access objects in WebLogic Server as though they were COM components.
- Applications within WebLogic Server to access COM components as though they were Java objects.

2.1.1 An Important Note on Terminology

Throughout the remainder of this programming guide, we refer to the two types of applications by their directions of access. Thus:

- An application in which a COM client accesses WebLogic Server objects is a "COM-to-WLS" application.

- An application in which WebLogic Server accesses COM objects is a "WLS-to-COM" application.

2.1.2 jCOM Architecture

WebLogic jCOM provides a runtime component that implements both COM/DCOM over Distributed Computing Environment Remote Procedure Call, and Remote Method Invocation (RMI) over Java Remote Method protocol/Internet Inter-ORB Protocol distributed components infrastructures. This makes the objects on the other side look like native objects for each environment.

WebLogic jCOM also provides automated tools to convert between both types of interfaces: it automatically builds COM/DCOM proxies and RMI stubs necessary for each side to be able to communicate through the above mentioned protocols.

WebLogic jCOM does all the necessary translation between DCOM and RMI technologies, and connects to WebLogic Server as an RMI client. It then communicates requests to Enterprise Java Beans (EJBs) deployed in the WebLogic Server as if the request comes from a regular EJB client.

In a similar manner, when a component deployed in WebLogic Server requests services provided by a DCOM object, the request is translated by the jCOM component from a regular RMI client request issued by the WebLogic Server into DCOM compliant request, and communicated to the DCOM environment to the appropriate object.

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.

2.2 Why Use WebLogic jCOM?

The major reasons for using WebLogic jCOM are:

- To gain interoperability among distributed applications that span diverse hardware and software platforms

- To aid those with a significant investment in Microsoft development tools and trained development staff who don't want to write Java client software in order for their client applications to access business logic on WebLogic Server.

- To address the needs of e-business application builders seeking to leverage the skills available for both COM/DCOM, and Java environments to build fully integrated applications and reuse existing components. The specifics of each environment can be completely hidden for developers used to another environment.

WebLogic jCOM follows a software industry trend of making heterogeneous environments and applications interoperate transparently.

2.3 WebLogic jCOM Features

The key features of the WebLogic jCOM subsystem are:

- 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 of object types.

- 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 supports an optional "native mode" which maximizes performance when running on a Windows platform. See Section 2.4.3, "DCOM Versus Native Mode."

- WebLogic jCOM supports event handling. For example, Java events are accessible from Visual Basic using the standard COM event mechanism and Java objects can subscribe to COM component events.

2.4 Planning Your WebLogic jCOM Application

Before designing and building your jCOM application, you must make a few key decisions. Specifically, you must decide:

- Whether to employ a zero-client architecture for your application (COM-to-WLS only)

- Whether to employ an early or late binding model (COM-to-WLS only)

- Whether to run your jCOM application in native or DCOM mode (both COM-to-WLS and WLS-to-COM)

The following sections provide information to help you make these decisions.

2.4.1 Zero-Client Deployment

A jCOM zero client deployment is easy to implement. No WebLogic-jCOM-specific software is required on the client machine.

The WebLogic Server location is coded into the COM client using an object reference moniker (`objref`) moniker string. The `objref` moniker is generated by the user and it encodes the IP address and port of the WebLogic Server. You can obtain the moniker string for the COM client code programmatically—or by copying and pasting—from a WebLogic Server servlet. Once the server connection is established, the COM client can link a COM object to an interface in the Java component.

2.4.1.1 Advantages and Disadvantages of Zero-Client Deployment

The following table summarizes the advantages and disadvantages of a zero-client implementation.

Advantages	Disadvantages
No WebLogic-specific software need be loaded into the client machine registry.	A few jCOM-specific tools must be copied from the *WL_HOME*\bin directory on the WebLogic Server machine
Offers the benefits of the late binding model (see Section 2.4.2, "Early Versus Late Binding") and therefore provides the same flexibility in terms of changes made to the Java component.	Requires that the WebLogic Server location and port number be coded into the COM client, which means that if the server location is changed, this reference has to be regenerated and changed in the source code.
	Deprives your application of the advantages of early binding. (See Section 2.4.2, "Early Versus Late Binding")

The zero-client model programming model is probably a good choice if your WebLogic jCOM deployment requires a large number of COM client machines.

2.4.2 Early Versus Late Binding

Binding substitutes the symbolic addresses of routines or modules with physical addresses. Early binding and late binding both provide access to another application's objects.

Early bound access gives you 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.

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.

2.4.2.1 Advantages and Disadvantages of Each Binding Model

The following table summarizes the pros and cons of the *early* binding model:

Early Binding Pros	Early Binding Cons
■ More reliable than late bound implementation. ■ Compile-time type checking makes debugging easy ■ The application's end user can browse the type library. ■ Improved runtime transaction performance relative to a late bound implementation.	■ Complex to implement, as 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 server are on separate machines the type library and wrappers have to be generated on the same machine and then copied to the systems where they are required. ■ Lacks the flexibility of late bound access, in that any changes made to the Java component require regeneration of the wrappers and the library. ■ Slower initialization at runtime than a late bound implementation.

The following table summarizes the pros and cons of the *late* binding model:

Late Binding Advantages	Late Binding Disadvantages
■ Easy to implement ■ Flexible implementation, since objects referenced are only evaluated at runtime ■ Faster runtime initialization than for an early bound implementation	■ Error prone, as no type checking can be done at compile time ■ It is not until you run the program that you find out if the methods and properties you are accessing actually exist. ■ Runtime transaction performance inferior to early bound implementation

2.4.3 DCOM Versus Native Mode

The DCOM (Distributed Component Object Model) mode uses the Component Object Model (COM) to support communication among objects on different computers. In a WebLogic jCOM application running in DCOM mode, the COM client communicates with WebLogic Server in DCOM protocol.

In native mode, COM clients make native calls to WebLogic Servers (COM-to-WLS) and WebLogic Servers make native calls to COM applications.

For both COM-to-WLS and WLS-to-COM applications, because native mode uses native code dynamically loaded libraries (DLLs)—which are compiled and optimized specifically for the local operating system and CPU—using native mode results in better performance.

Moreover, COM-to-WLS applications operating in native mode use WebLogic's T3/Internet InterORB (IIOP) protocols for communication between the COM client and WebLogic Server. This brings the advantages of:

- Superior performance as compared to using DCOM calls because it results in fewer network calls

  For example, suppose your COM application creates a vector containing 100 data elements whose values are returned by a call to WebLogic Server. In DCOM mode, this would require 100 roundtrip network calls to the server. In native mode, this would require one roundtrip call.

- Access to WebLogic Server's failover and load balancing features

However, for both types of applications, because native libraries have only been created for Windows, implementing native late bound access requires that the WebLogic Server be installed all COM client machines.

Moreover, for WLS-to-COM applications, WebLogic Server must be running on a Windows machine to run in native mode.

2.4.3.1 Advantages and Disadvantages of Native Mode

The following table summarizes the pros and cons of a native mode implementation.

Advantages	Disadvantages
For COM-to-WLS applications, better performance than DCOM mode because calls are not made over the network.	For COM-to-WLS and WLS-to-COM applications, since native libraries have only been created for Windows, implementing native mode requires that the WebLogic Server be installed on all COM machines.
For COM-to-WLS applications, access to WebLogic Server's load balancing and failover features.	For WLS-to-COM applications, WebLogic Server must be running on a Windows machine to run in native mode.
For WLS-to-COM applications, better performance because calls are not made over the network.	

2.5 jCOM Features and Changes in this Release

WebLogic jCOM now supports passing—from COM to Java—two dimensional arrays of the following COM types:

Table 2–1   Two Dimensional Array Support in jCOM

COM Type	Visual Basic Type	Java Type
I1, UI1	Byte	Byte
BOOL	Boolean	Boolean
I2, UI2	Integer	Short
CY, I8, UI8	Currency	Long

Table 2–1   (Cont.) Two Dimensional Array Support in jCOM

COM Type	Visual Basic Type	Java Type
R8	Double	Double
DATE	Date	Date
I4, UI3, INT, UINT	Long	Int

3

Calling into WebLogic Server from a COM Client Application

This chapter describes how to use WebLogic jCOM to call methods on a WebLogic Server object from a COM client.

- Section 3.1, "Special Requirement for Native Mode"
- Section 3.2, "Calling WebLogic Server from a COM Client: Main Steps"
- Section 3.3, "Preparing WebLogic Server"
- Section 3.4, "Preparing the COM Client"
- Section 3.5, "Running COM-to-WLS Applications in Native Mode"

3.1 Special Requirement for Native Mode

Note that WebLogic Server must be installed on COM client machines in order for your COM-to-WLS application to run in native mode.

For more information on native mode, see Section 3.5, "Running COM-to-WLS Applications in Native Mode."

3.2 Calling WebLogic Server from a COM Client: Main Steps

This section summarizes the main steps to call into WebLogic Server from a COM client. Most are described in detail in later sections.

On the WebLogic Server side:

1. If you are using early binding, run the java2com tool to generate Java wrapper classes and an Interface Definition Language (IDL) file and compile the files. See Section 3.3.1, "Generate Java Wrappers and the IDL File—Early Binding Only."

2. Enable COM calls on the server listen port. See Enable jCOM in the *Oracle WebLogic Server Administration Console Help*.

3. Grant access to server classes to COM clients. See Section 3.3.2, "Configuring Access Control."

4. Configure any other relevant console properties. See Servers: Protocols: jCOM in the *Oracle WebLogic Server Administration Console Help*.

On the COM client side:

1. Install the jCOM tools files and, for native mode only, WebLogic Server class files. See Section 3.4.1, "Install Necessary Files."

2. If this is a zero-client installation:

   ■ Obtain an object reference moniker (ORM) from the WebLogic Server ORM servlet, either progammatically or by pasting into your application. See Section 3.4.2, "Obtain an Object Reference Moniker from the WebLogic Server Servlet—Zero Client Only."

3. If you are using early binding:

   ■ Obtain the IDL file generated on the WebLogic Server machine and compile it into a type library.

   ■ Register the type library and the WebLogic Server instance serviced.

   For both of these steps, see Section 3.4.3, "Generate Java Wrappers and the IDL File—Early Binding Only."

4. Register the WebLogic Server JVM in the registry. If want to communicate with the WebLogic Server in native mode, set that in this step. See Section 3.4.4, "Register the WebLogic Server JVM in the Client Machine Registry."

5. Code the COM client application. See Section 3.4.5, "Code the COM Client Application."

6. Start the COM client. See Section 3.4.6, "Start the COM Client."

3.3 Preparing WebLogic Server

The following sections discuss how to prepare WebLogic Server so that COM clients can call methods on WebLogic Server objects:

3.3.1 Generate Java Wrappers and the IDL File—Early Binding Only

1. Add the path to JDK libraries and `weblogic.jar` to your `CLASSPATH`. For example:

   ```
   set CLASSPATH=%JAVA_HOME%\lib\tools.jar;
   %WL_HOME%\server\lib\weblogic.jar;%CLASSPATH%
   ```

   Where *JAVA_HOME* is the root folder where the JDK is installed (`c:\Oracle\Middleware\jdk160` or `c:\Oracle\Middleware\jrockit_160` by default) and *WL_HOME* is the root directory where WebLogic Platform software is installed (`c:\Oracle\Middleware\wlserver_10.3` by default).

2. Generate java wrappers and an IDL file with the `java2com` tool:

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

   The `java2com` GUI is displayed:
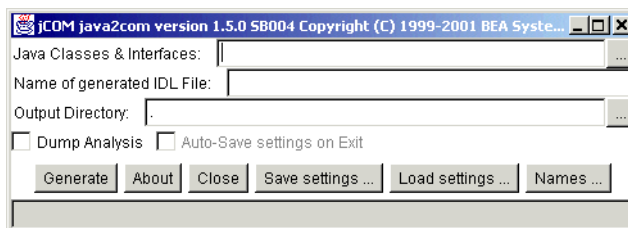
3. Input the following:

   Java Classes & Interfaces: *list of the wrapper classes to be converted*

Name of generated IDL File: *name of the IDL file*

Output Directory: *drive letter and root directory\TLB*

where *TLB* signifies OLE Type Library.

The `java2com` tool looks at the class specified, and at all other classes that it uses in the method parameters. It does this recursively. You can specify more than one class or interface here, separated by spaces.

All Java classes that are public, not abstract, and have a no-parameter constructor are rendered accessible as COM Classes. Other public classes, and all public interfaces are rendered accessible as COM interfaces.

If you click the "Generate" button and produce wrappers and the IDL at this point, errors are generated. This is because certain classes are omitted by default in the `java2com` tool. By looking at the errors generated during compilation, you would be able to determine which classes were causing problems.

To fix the problem, click the "Names" button in the `java2com` tool and remove any references to the class files you require. In this example we must remove the following references:

```
*.toString > ''''
class java.lang.Class > ''''
```
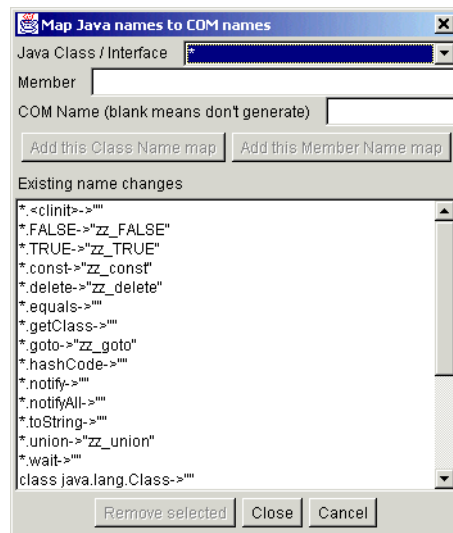

4. Once these references have been removed, you can generate your wrappers and IDL. Click Generate in the `java2com` GUI.

The `java2com` tool generates Java classes containing DCOM marshalling code used to access Java objects. These generated classes are used behind the scenes by the WebLogic jCOM runtime. You simply need to compile them, and make sure that they are in your CLASSPATH.

3.3.2 Configuring Access Control

Grant the COM client user access to the classes that the COM client application needs to access. Your particular application dictates which classes to expose.

For example, assume that the COM client needs access to the following three classes:

- `java.util.Collection`

- `java.util.Iterator`

- `ejb20.basic.beanManaged`

3.3.2.1 Granting Access to java.util.Collection and java.util.Iterator

1. In the left-hand pane of the WebLogic Server Administration Console, click the Services node and then click the JCOM node underneath it.

2. In the right-hand pane, enter:

   `java.util.*`

3. Click Define Security Policy.

4. In the Policy Condition box, double-click "Caller is a member of the group".

5. In the "Enter group name:" field, enter the name of the group of users to whom you're granting access.

6. Click Add.

7. Click OK.

8. In the bottom right-hand corner of the window, click Apply.

3.3.2.2 Granting Access to ejb20.basic.beanManaged

To grant access to `ejb20.basic.beanManaged`, repeat the steps in Section 3.3.2.1, "Granting Access to java.util.Collection and java.util.Iterator," replacing "java.util.*" with "ejb20.basic.beanManaged" in step 3.

3.4 Preparing the COM Client

The following sections describe how to prepare a COM client to call methods on WebLogic Server objects:

3.4.1 Install Necessary Files

There are a number of files that must be installed on your client machine in order to call methods on WebLogic Server objects. As noted below, some of these are only necessary if you are making method calls in native mode.

3.4.1.1 jCOM Tools Files

There are five files and three folders (including all subfolders and files) necessary for running the jCOM tools. These tools are located in the *WL_HOME*\server\bin directory on the machine where you installed WebLogic Server. They are:

- `JintMk.dll`

- `ntvinv.dll`

- `regjvm.exe`

- `regjvmcmd.exe`

- `regtlb.exe`

- `regjvm` (including all subfolders and files)

- `regjvmcmd` (including all subfolders and files)

- `regtlb` (including all subfolders and files)

For more information on the jCOM tools, see Chapter 5, "A Closer Look at the jCOM Tools."

3.4.1.2 WebLogic Server Class Files—Native Mode Only

In order to run a COM-to-WLS application in native mode, a COM client machine must have access to certain WebLogic Server class files. To obtain these files, install WebLogic Server on each COM client machine.

3.4.2 Obtain an Object Reference Moniker from the WebLogic Server Servlet—Zero Client Only

You can obtain an object reference moniker (ORM) from WebLogic Server. The moniker can be used from the COM client application, obviating the need to run `regjvmcmd`. The moniker remains valid for new incarnations of the server as long as the host and port of the server remain the same.

There are two ways to obtain an ORM for your COM client code:

- Obtain it through a servlet running on WebLogic Server. Open a Web browser on WebLogic Server to `http://[wlshost]:[wlsport]/bea_wls_ internal/com`

  where *wlshost* is the WebLogic Server machine and *wlsport* is the server's port number.

- Run the `com.bea.jcom.GetJvmMoniker` Java class, specifying as parameters the full name or TCP/IP address of the WebLogic Server machine and port number:

  ```
  java com.bea.jcom.GetJvmMoniker [wlshost] [wlsport]
  ```

  A long message is displayed which shows the objref moniker and explains how to use it. The text displayed is also automatically copied to the clipboard, so it can be pasted directly into your source. The objref moniker returned can access the WebLogic Server instance on the machine and port you have specified.

3.4.3 Generate Java Wrappers and the IDL File—Early Binding Only

Perform the client-side portion of the wrapper and Interface Definition Language (IDL) file generation:

1. Copy the IDL to the client machine:

   If the `java2com` tool successfully executes on the WebLogic Server machine (see Section 3.3, "Preparing WebLogic Server"), an IDL file is produced on the server machine. Copy this IDL file to the client machine, and place it in this COM application's `\TLB` subdirectory.

   > **Note:** If the client and the server are on the same machine, this step is not necessary. The `java2com` tool outputs to the sample's `\TLB` subdirectory.

2. Compile the IDL file into a type library:

   ```
   midl containerManagedTLB.idl
   ```

This command calls the Microsoft IDL compiler `MIDL.EXE` to carry out the compilation. The result of the compilation is a type library called `containerManagedTLB.tlb`.

3. Register the type library and set the JVM it services:

```
regtlb /unregisterall
regtlb containerManagedTLB.tlb registered_jvm
```

The first line above calls the `regtlb.exe` in order to un-register any previously registered type library versions. The second line then registers the newly compiled type library.

The second parameter `registered_jvm` passed to `regtlb` is important. It specifies the name of the JVM linked with the type library. The WebLogic jCOM runtime requires this information for linking type library defined object calls to the appropriate wrapper classes.

The WebLogic Server JVM is registered in the client machine registry through the `regjvm` tool. For details, see Section 3.4.4, "Register the WebLogic Server JVM in the Client Machine Registry."

3.4.3.1 Some Notes about Wrapper Files

- In general, wrapper files must be placed on the server and compiled. The IDL file must be placed on the client and compiled. If the server and client are on separate machines, and you created the wrappers and IDL on the *client* side, you must distribute the wrapper files you have just compiled to the server. If you created the wrappers and IDL on the *server* side, then you must move the IDL file to the client, where it can be compiled to a type library.

- The wrapper files and IDL file must be created by a *single execution of the* `java2com` *tool*. If you attempt to run the `java2com` tool separately on both the server and the client, the wrappers and IDL file created would not be able to communicate. The IDL and wrappers have unique stamps on them for identification; wrappers can only communicate with IDL files created by a common invocation of the `java2com` tool, and vice versa. As a result, the `java2com` tool must be run once, and the files it creates distributed afterward. If you make a mistake or a change in your Java source code and you need to run the `java2com` tool again, you must delete all of your wrapper files, your IDL file, and your TLB file, and redo all the steps.

- When you use the `java2com` tool to create wrappers for classes that contain (or reference) deprecated methods, you see deprecation warnings at compile time. disregard these warnings; WebLogic jCOM renders the methods accessible from COM.

- The generated wrapper classes must be in your CLASSPATH. They cannot be just located in your EJB jar.

3.4.4 Register the WebLogic Server JVM in the Client Machine Registry

Register with the local Java Virtual Machine by adding the server name to the Windows registry and associating it with the TCP/IP address and client-to-server communications port where the WebLogic Server instance listens for incoming COM requests. By default, this is localhost:7001.

1. Invoke the `regjvm` GUI tool, which displays this screen.
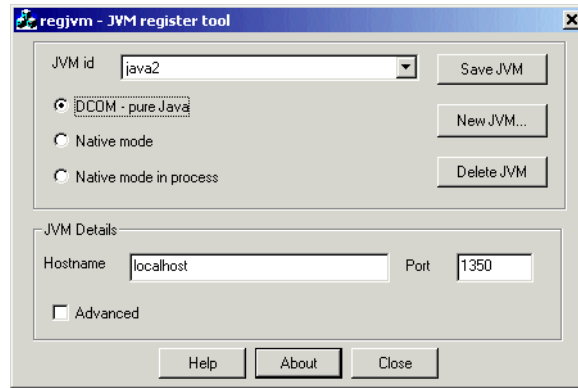
2. If WebLogic Server is running on something other than localhost and listening on a port other than 7001, then fill in the hostname (or IP address) and port number

If you prefer, use the command-line version of `regjvm`:

```
regjvmcmd servername localhost[7001]
```

3.4.4.1 Unregistering JVMs

The `regjvm` (or `regjvmcmd`) 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, unregister the old entry, and then create a new one.
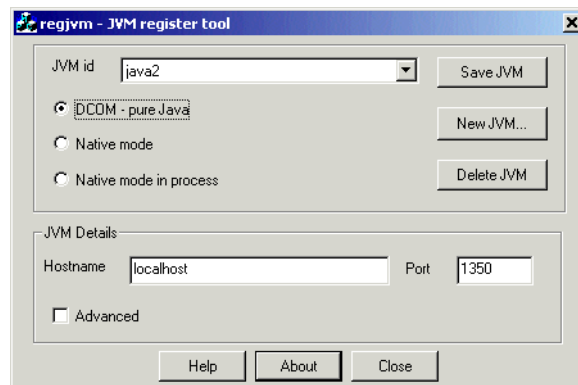
To unregister a JVM in the `regjvm` tool window, select the JVM you wish to unregister and click Delete.

Alternatively, unregister the JVM with the command line tool `regjvmcmd`:

```
regjvmcmd /unregister servername
```

3.4.4.2 Select Native Mode, If Applicable

If your COM client is running in native mode, check the "Native Mode" or "Native Mode Out-of-Process" radio button in the `regjvm` window or invoke `regjvmcmd` with the `/native` parameter. For details on this step, see Section 3.5, "Running COM-to-WLS Applications in Native Mode."

3.4.5 Code the COM Client Application

You can now invoke methods on the WebLogic Server objects. How you code this naturally depends on whether you chose late binding or early binding.

3.4.5.1 Late Bound Applications

In the following sample Visual Basic Application, notice the declaration of the COM version of the `Account` EJB's home interface `mobjHome`. This COM object is linked to an instance of the `AccountHome` interface on the server side.

```
Dim mobjHome As Object
Private Sub Form_Load()
'Handle errors
On Error GoTo ErrOut '
Bind the EJB AccountHome object through JNDI
Set mobjHome =
CreateObject("examplesServer:jndi:ejb20-containerManaged-AccountHome")
```

3.4.5.1.1 Known Problem and Workaround for Late Bound Clients WebLogic jCOM has problems handling methods that are overloaded but have the same number of parameters. There is no such problem if the number of parameters in the overloaded methods are different.

When they're the same, calls fail.

Unfortunately, the method `InitialContext.lookup` is overloaded:

```
public Object lookup(String)
public Object lookup(javax.naming.Name)
```

To perform a lookup, you must use the special JNDI moniker to create an object:

```
Set o = CreateObject("servername:jndi:objectname")
```

3.4.5.2 Early Bound Applications

The most obvious distinguishing feature of early bound code is that fewer variables are declared `As Object`. Objects can now be declared by using the type library you generated previously:

Declare objects using the type library generated in Section 3.4.3, "Generate Java Wrappers and the IDL File—Early Binding Only." In this Visual Basic code fragment, the IDL file is called `containerManagedTLB` and the EJB is called `ExamplesEjb20BasicContainerManagedAccountHome`:

```
Dim objNarrow As New containerManagedTLB.JCOMHelper
```

Now, you can call a method on the object:

```
Set mobjHome = objNarrow.narrow(objTemp,
"examples.ejb20.basic.containerManaged.AccountHome")
```

3.4.6 Start the COM Client

Start up the COM client application.

3.5 Running COM-to-WLS Applications in Native Mode

For COM-to-WLS applications, there's a distinction in native mode between "in-process" and "out-of-process":

- Out-of-process: The JVM is created in its own process; inter-process communication occurs between the COM process and the WebLogic Server JVM process.

- In-process: The entire WebLogic Server JVM is brought into the COM process; in effect, it's loaded into the address space of the COM client. The WebLogic Server client-side classes reside inside this JVM.

You determine which process your application uses by selecting the native-mode-in-process or native mode radio button in the `regjvm` GUI tool interface.

3.5.1 Native Mode with the JVM Running Out-of-Process

If you want your JVM to run out of process (but allow COM client access to the Java objects contained therein using native code), follow these steps:

1. Invoke the `regjvm` GUI tools to register your JVM as being native. The `regjvm` sets up various registry entries to facilitate WebLogic jCOM's COM-to-WLS mechanism. When you register the JVM you must provide the name of the server in the JVM id field. For example, if you enabled JCOM native mode on `exampleServer` then when you register with `regjvm` enter `exampleServer` in the JVM id box.

2. If your JVM is not already running, click the Advanced radio button and type its path in the "Launch Command" field.

   For detailed information on the `regjvm` tool, see Chapter 5, "A Closer Look at the jCOM Tools."

3. Insert the following code into the `main` section of your application code, to tell the WebLogic jCOM runtime that the JVM is ready to receive calls:

   ```
   com.bea.jcom.Jvm.register("MyJvm"):
   public class MyJvm {
   public static void main(String[] args) throws Exception {
   // Register the JVM with the name "firstjvm"
   com.bea.jcom.Jvm.register("firstjvm");
   Thread.sleep(6000000); // Sleep for an hour
   }
   ```

4. From Visual Basic you can now use late binding to instantiate instances of any Java class that can be loaded in that JVM:

   ```
   Set acctEJB = CreateObject("firstjvm.jndi.ejb20.beanManaged.AccountHome")
   ```

5. Having registered the JVM, use the standard WebLogic jCOM `regtlb` command to allow early bound access to Java objects (`regtlb` takes as parameters the name of a type library, and a JVM name, and registers all the COM objects defined in that type library as being located in that JVM).

   You can also control the instantiation of Java objects on behalf of COM clients by associating your own instantiator with a JVM (additional parameter to `com.bea.jcom.Jvm.register(...)`)—a kind of object factory.

3.5.2 Native Mode with the JVM Running In-Process

Use this technique to actually load the JVM into the COM client's address space.

Again, use the `regjvm` command, but this time specify additional parameters.

> **Note:** When you register the JVM you must provide the name of the server in the JVM id field. For example, if you enabled JCOM native mode on `exampleServer` then when you register with `regjvm` enter `exampleServer` in the JV id box.

The simplest example would be to use Visual Basic to perform late bound access to Java objects. First register the JVM. If you are using Sun's JDK 1.3.1, which is installed under `c:\Oracle\Middleware\jdk160` by default, and WebLogic Server is installed in `c:\Oracle\Middleware\wlserver_ 10.3\server\lib\weblogic.jar`, and your Java classes are in `c:\pure`, you would complete the `regjvm` tools screen as follows:
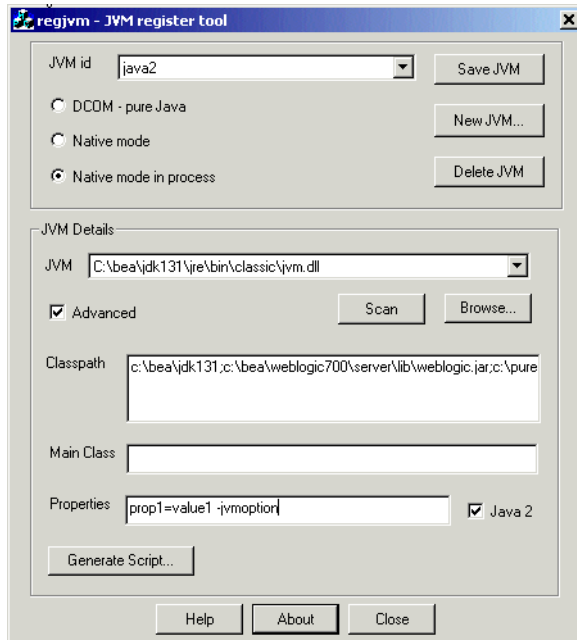
As you can see, you specify the JVM name, the CLASSPATH, and the JVM bin directory path.

From Visual Basic, you should now be able to call the GetObject method:

```
MessageBox GetObject("MyJVM.jndi.ejb20.beanManaged.AccountHome")
```

For detailed information on the `regjvm` tool, see Chapter 5, "A Closer Look at the jCOM Tools."

4

Calling into a COM Application from WebLogic Server

This chapter describes how to prepare and deploy a WLS-to-COM application: an application that uses WebLogic jCOM to call methods on a COM object from WebLogic Server.

4.1 Special Requirements for Native Mode

Note these two special requirements for WLS-to-COM applications that use native mode:

- In order for a COM application to run in native mode, WebLogic Server must be installed on the COM application machine.

- In order to run in native mode, WebLogic Server must be running on a Windows machine.

4.2 Calling a COM Application from WebLogic Server: Main Steps

This section summarizes the main steps to call into a COM application from a WebLogic Server. Most are described in detail in later sections.

On the COM side:

1. Code the COM application. See Section 4.3.1, "Code the COM Application."

2. Generate Java classes from the COM objects with the com2java tool. See Section 4.3.2, "Generate Java Classes with the com2java GUI Tool."

3. Package the classes for use by WebLogic Server. See Section 4.3.3, "Package the Java Classes for WebLogic Server."

4. Start the COM application. See Section 4.3.4, "Start the COM Application."

On the WebLogic Server side:

1. Enable COM calls on the server listen port. See Enable jCOM in the *Oracle WebLogic Server Administration Console Help*.

2. Configure any other relevant console properties. See Servers: Protocols: jCOM in the *Oracle WebLogic Server Administration Console Help*.

   If you have chosen to have WebLogic Server and the COM application communicate in native mode, enable it in the Administration Console. See the Chapter 2.4.3, "DCOM Versus Native Mode" for help deciding whether to use native mode.

3. Use the COM objects as you would any other Java object.

4.3 Preparing the COM Application

The following sections describe how to prepare a COM client so that WebLogic Server can call methods on its objects:

4.3.1 Code the COM Application

Code your COM application as desired.

4.3.2 Generate Java Classes with the com2java GUI Tool

Running the `com2java` GUI tool against a COM type library generates a collection of Java class files corresponding to the classes and interfaces in the COM type library.

Here we demonstrate Java class generation with the GUI tool. To read more about the WebLogic jCOM tools in general, see Chapter 5, "A Closer Look at the jCOM Tools."

1. To run the `com2java` GUI tool, perform the following steps:

   a. Change to the *WEBLOGIC_HOME*/`server/bin` directory (or add this directory to your CLASSPATH)

   b. Open a command shell on the COM machine and invoke the `com2java.exe` file:

   ```
   > com2java
   ```

2. Select the appropriate type library in the top field, and fill in the Java package text box with the name of the package to contain the generated files. The `com2java` tool remembers the last package name you specified for a particular type library.

3. Click Generate Proxies to generate Java class files.

4.3.3 Package the Java Classes for WebLogic Server

If you call a COM object from an EJB, you must package the class files generated by `com2java` into your EJB `.jar` in order for WebLogic Server to find them. Place the generated files in a specific package. For example you may want to put all the files for the Excel type library in a Java package called *excel*.

For more information on packaging EJB `.jar` files, see "Implementing EJBs" in *Programming WebLogic Enterprise JavaBeans for Oracle WebLogic Server*.

4.3.4 Start the COM Application

Once you have generated the Java class files and packaged them appropriately, simply start your COM application, so that the COM objects you want to expose to WebLogic Server are instantiated and running.

4.4 Using Java Classes Generated by com2java

For each COM class that the `com2java` tool finds in a type library, it generates a Java class which you use to access the COM class. These generated Java classes have several constructors:

- The default constructor, which creates an instance of the COM class on the local host, with no authentication

- A second constructor, which creates an instance of the COM class on a specific host, with no authentication

- A third constructor, which creates an instance of the COM class on the local host, with specific authentication

- A fourth constructor, which creates an instance of the COM class on a specified host, with specific authentication

- A final constructor, which can wrap a returned object reference which is known to reference an instance of the COM class

Here are sample constructors generated from the DataLabelProxy class:

```
public DataLabelProxy() {}

public DataLabelProxy(Object obj) throws java.io.IOException {
super(obj, DataLabel.IID);
}
protected DataLabelProxy(Object obj, String iid) throws
java.io.IOException
{
super(obj, iid);
}
public DataLabelProxy(String CLSID, String host, boolean
deferred) throws java.net.UnknownHostException,
java.io.IOException{ super(CLSID, DataLabel.IID, host, null);
}
protected DataLabelProxy(String CLSID, String iid, String host,
AuthInfo authInfo) throws java.io.IOException { super(CLSID,
iid, host, authInfo);
}
```

4.5 Using Java Interfaces Generated from COM interfaces by com2java

A method in a COM interface may return a reference to an object through a specific interface.

For example the Excel type library (`Excel8.olb`) defines the `_Application` COM Interface, with the method `Add` which is defined like this in COM IDL:

```
[id(0x0000023c), propget, helpcontext(0x0001023c)]
HRESULT Workbooks([out, retval] Workbooks** RHS);
```

The method returns a reference to an object that implements the `Workbooks` COM interface. Because the `Workbooks` interface is defined in the same type library as the `_Application` interface, the com2java tool generates the following method in the `_Application` Java interface it creates:

```
/** * getWorkbooks.
*
* @return return value. An reference to a Workbooks
```

```
* @exception java.io.IOException If there are communications problems.
* @exception com.bea.jcom.AutomationException If the remote server throws an exception. */
public Workbooks getWorkbooks () throws java.io.IOException, com.bea.jcom.AutomationException;
It is revealing to look at the implementation of the method in the generated _ApplicationProxy Java
class:
/**
* getWorkbooks.
*
* @return return value. An reference to a Workbooks
* @exception java.io.IOException If there are communications
problems.
* @exception com.bea.jcom.AutomationException If the remote
server throws an exception.
*/
public Workbooks getWorkbooks () throws java.io.IOException,
com.bea.jcom.AutomationException{ com.bea.jcom.MarshalStream
marshalStream = newMarshalStream("getWorkbooks");
marshalStream = invoke("getWorkbooks", 52, marshalStream);
Object res = marshalStream.readDISPATCH("return value");
Workbooks returnValue = res == null ? null : new
WorkbooksProxy(res);
checkException(marshalStream,
marshalStream.readERROR("HRESULT"));
return returnValue;
}
```

As you can see, the `getWorkbooks` method internally uses the generated
`WorkbooksProxy` Java class. As mentioned above, the com2java tool generates the
method with the `Workbooks` return type because the `Workbooks` interface is defined
in the same type library as `_Application`.

If the `Workbooks` interface were defined in a different type library, WebLogic jCOM
would have generated the following code:

```
/**
* getWorkbooks.
*
* @return return value. An reference to a Workbooks
* @exception java.io.IOException If there are communications
problems.
* @exception com.bea.jcom.AutomationException If the remote server
throws an exception.
*/
public Object getWorkbooks () throws java.io.IOException,
com.bea.jcom.AutomationException;
In this case, you would have to explicitly use the generated proxy class to access the returned
Workbooks:
Object wbksObj = app.getWorkbooks();
Workbooks workbooks = new WorkbooksProxy(wbObj);
```

5

A Closer Look at the jCOM Tools

This chapter provides detailed information tools used by jCOM applications. These tools include `com2java`, `java2com`, `regjvm`, `regjvmcmd`, and `regtlb`.

- Section 5.1, "com2java"

- Section 5.2, "java2com"

- Section 5.3, "regjvm"

- Section 5.4, "regjvmcmd"

- Section 5.5, "regtlb"

5.1 com2java

WebLogic jCOM's `com2java` tool reads information from a type library, and generates Java files that you use to access the COM classes and interfaces defined in that type library.

Type libraries contain information on COM classes, interfaces, and other constructs. They are typically generated by development tools such as Visual C++ and Visual BASIC.

Some type libraries are readily identifiable as such. Files that end with the extension `olb` or `tlb` are definitely type libraries. What can be a little confusing is that type libraries can also be stored inside other files, such as executables. Visual BASIC puts a type library in the executable that it generates.

5.1.1 Using com2java

Start `com2java` by typing it in a command shell or double clicking its icon.

When you start `com2java`, this is the dialog that is displayed:
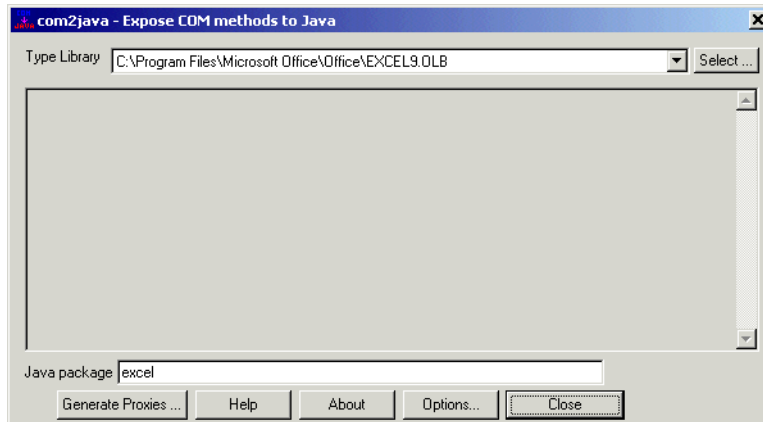
5.1.1.1 Selecting the Type Library

Click the Select button to select the type library that the tool should process.

Remember that type libraries can sometimes be hidden inside executable files, such as the executable or dynamic link library (DLL) containing your COM component.

The com2java tool remembers a list of the last type libraries you successfully opened and generated proxies for.

5.1.1.2 Specifying the Java Package Name

The com2java tool generates a set of Java source files corresponding to the COM classes and interfaces in the type library. Place the generated files in a specific package. For example you may want to put all the files for the Excel type library in a Java package called *excel*.

In the Java package text box, specify the name of the package to which the generated files to belong.

The com2java tool remembers the last package name you specified for a particular type library.

5.1.1.3 Options

Click the Options button to display a dialog box with com2java options described below. Note that these options are saved automatically between sessions of com2java. If you only require an option for one particular generation of proxies, then reset the option after generating the proxies.

Option	Description
Clash Prefix	If methods in the COM interfaces defined in the type library clash with methods that are already used by Java (for example the getClass() method), com2java prefixes the generated method name with a string, which is zz_ by default.
Lower case method names	The convention for Java method names is that they start with a lower-case letter. By default the com2java tool enforces this convention, changing method names accordingly. To have com2java ignore the convention, deselect the Lowercase method names check box in the Options dialog box.
Only generate IDispatch	WebLogic jCOM supports calling COM objects using IDispatch and vtable access. Selecting this option ensures that all calls are made using the IDispatch interface.
Generate retry code on '0x80010001 - Call was rejected by callee'	If a COM server is busy, you may receive an error code. Selecting this option ensures that the generated code retries each time this error code is received.

Option	Description
Generate Arrays as Objects	Parameters that are SAFEARRAYS have a corresponding Java parameter of type java.lang.Object generated. This is required if you are passing two dimensional arrays outside of Variants to/from COM objects from Java.
	This option doesn't change what is actually passed over the wire —it is still arrays—it is just that in the generated Java interface, rather than having the generated method prototype specify the type of the array, it specifies "Object". This is useful in situations where you want to pass a 2D array —in the COM IDL the number of dimensions is not specified for SAFEARRAYS, and if you don't check the "generate arrays as objects" option, WebLogic jCOM assumes you are passing a single element array and generate a corresponding prototype.
	By setting the option, and having com2java generate "Object" instead of "String[]", for example, you are free to actually pass a 2D string array.
Prompt for names for imported tlbs	Sometimes a type library imports another type library. If you are also generating proxies for imported type libraries, using this option prompts you for the package name of the those proxies.
Don't generate dispinterfaces	Selecting this option disables the generation of proxies for interfaces defined as dispinterfaces.
Generate deprecated constructors	Generated proxies contain some constructors which are now deprecated. If you do wish to generate these deprecated constructors select this option.
Don't rename methods with same names	If a name conflict is detected in a COM class, com2java automatically renames one of the methods. Selecting this option overrides this automatic renaming.
Ignore conflicting interfaces	If a COM class implements multiple interfaces which define methods with the same names, selecting this option prevents the corresponding generated Java classes from implementing the additional interfaces. You can still access the interfaces using the getAsXXX method that is generated. See the generated comments.
Generate Java Abstract Window Toolkit (AWT) classes	Generates Java Classes as GUI classes. To be used for embedding ActiveX controls in Java Frames.

5.1.1.4 Generate the Proxies

Click the Generate Proxies button to select the directory in which the com2java tool should generate the Java files.

Once you select the directory, com2java analyzes.the type library and output the corresponding files in the directory you specify. If the directory already contains Java source files, WebLogic jCOM issues a warning and allows you to cancel the operation.

5.1.2 Files Generated by com2java

The com2java tool processes three kinds of constructs in a type library:

- "Enumerations"
- "COM Interfaces"
- "COM Classes"

Refer to documentation about the COM objects that you are accessing to understand how to use generated Java files to manipulate the COM objects. For example when you run com2java on the Excel type library the generated Java files you are seeing correspond to the Microsoft Excel COM API, and you should refer to the Microsoft Excel programming documentation for more information.

5.1.2.1 Enumerations

An enumeration is a list; in Java it is represented by `java.util.Enumeration`. If a type library contains an enumeration, WebLogic jCOM generates a Java interface containing constant definitions for each element in the enumeration.

5.1.2.2 COM Interfaces

WebLogic jCOM handles two types of interfaces. It handles Dispatch interfaces, whose methods can only be accessed using the COM IDispatch mechanism, and dual interfaces, whose methods can be invoked directly (vtbl access).

For each COM interface defined in a type library, the `com2java` tool generates two Java files: a Java interface, and a Java class.

The name of the generated Java interface is the same as the name of the COM interface. For example if the COM interface is called `IMyInterface`, the `com2java` tool generates a Java interface called `IMyInterface` in the file `IMyInterface.java`.

The second file that `com2java` generates is a Java class, which contains code used to access COM objects that implement the interface, and also code to allow COM objects to invoke methods in Java classes that implement the interface. The name of the generated Java class is the name of the interface with 'Proxy' appended to it. Using the example from the previous paragraph, WebLogic jCOM would generate a Java class called *IMyInterfaceProxy* in the file `IMyInterfaceProxy.java`.

For each method in the COM interface, WebLogic jCOM generates a corresponding method in the Java interface. In addition it generates some constants in the interface which, as the generated comments indicate, you can safely ignore.

Once again, WebLogic jCOM picks up comments from the type library describing the interface and its methods, and uses them in the generated javadoc comments.

5.1.2.3 COM Classes

A COM class implements one or more COM interfaces, in the same way that a Java class can implement one or more Java interfaces.

For each COM class in a type library, the `com2java` tool generates a corresponding Java class, with the same name as the COM class. WebLogic jCOM also supports a class implementing multiple interfaces.

The Java class which WebLogic jCOM generates can access the corresponding COM class.

5.1.2.3.1 Special Case—Source Interfaces (Events) A COM class can specify that an interface is a *source* interface. This means that it allows instances of COM classes that implement the interface to subscribe to the events defined in the interface. It invokes the methods defined in the interface on the objects that have subscribed.

> **Note:** In order for the `com2java` tool to treat an interface in a type library as an Event interface, there must be at least one COM class in the type library that uses the interface as a source interface.

Although COM events work using connection points, and source interfaces, Java has a different event mechanism. The `com2java` tool hides the COM mechanism from the Java programmer, and presents the events using the standard Java techniques.

What this means in real terms is that `com2java` adds two methods to the Java class that it generates for accessing the COM Class.

When the `com2java` tool detects that a class uses an interface as a source interface, it generates special code for that interface. It derives the interface from the `java.util.EventListener` Java interface, as is the convention for Java events.

Another Java event convention is that each of the methods in the interface should have a single parameter, which is an instance of a class derived from `java.util.EventObject` Java class.

One final Java event related convention is the use of an *Adapter* class, which implements the event interface, and provides empty default implementations for the methods in the interface. That way, developers that wish to create a class subscribed to the event need not implement all of the methods in the interface, which can be especially painful with large interfaces.

For each event interface, WebLogic jCOM generates an adapter class.

5.2 java2com

You can run `java2com` on any platform. Make sure that the WebLogic jCOM runtime `weblogic.jar` is in your `CLASSPATH` environment variable.

The `java2com` tool analyzes Java classes (using the Java 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.

After you generate these files, compile the IDL file using Microsoft's `MIDL` tool.

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

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

The `java2com` tool displays the following dialog box:
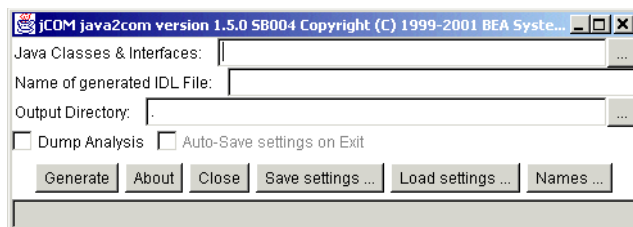
The dialog box has the following fields (any changes to the configuration are automatically saved when you exit the dialog box).

Field	Description
Java Classes and Interfaces	These are the 'root' Java classes and interfaces for 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 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 the ... button to display a dialog that lists the classes and lets you add/remove from the list.
Name of Generated IDL File	This is the name of the COM Interface Definition Language (IDL) file generated. If you specify myjvm, then myjvm.idl is generated. This name is also used for the name of the type library generated when you compile myjvm.idl using Microsoft's MIDL compiler.
Output Directory	The directory to which java2com should output the files it generates. The default is the current directory (".").
Dump Analysis	Displays the classes that the java2com discovers, as it discovers them.
Save Settings/Load Settings	Click the Save Settings button to save the current java2com settings. Do this before you click Generate.
	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.
Names	Clicking the Names button displays the following dialog box:
	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.
Generate button	Click this button to generate 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 generates Java events, then the generated COM class has 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 Microsoft Web site. The command
	`midl procdServ.idl`
	produces a type library called prodServ.tlb, which you must register, as described in Section 5.5, "regtlb."

5.3 regjvm

In order for WebLogic jCOM to allow languages supporting COM to access Java objects as though they were COM objects, you must 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, unregister the old entry and then reregister the 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 *WL_HOME*\server\bin directory).

5.3.1 JVM Modes

You can access a JVM from COM clients in one of three different modes:

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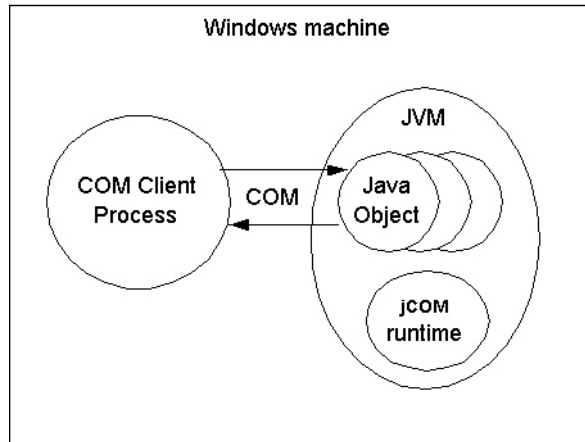- Native mode in process

5.3.1.1 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.
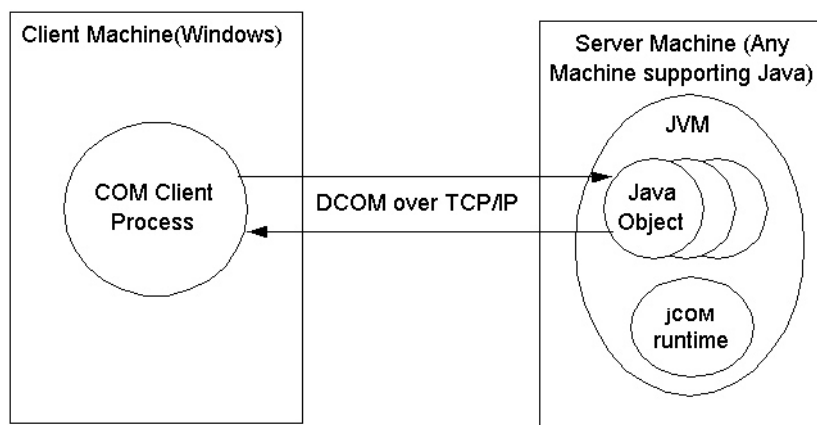
The Java code in the JVM must call com.bea.jcom.Jvm.register(<jvm id>), where <jvm id> is the id of the JVM as defined in regjvm.

#### 5.3.1.2 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`.

### 5.3.1.3 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.
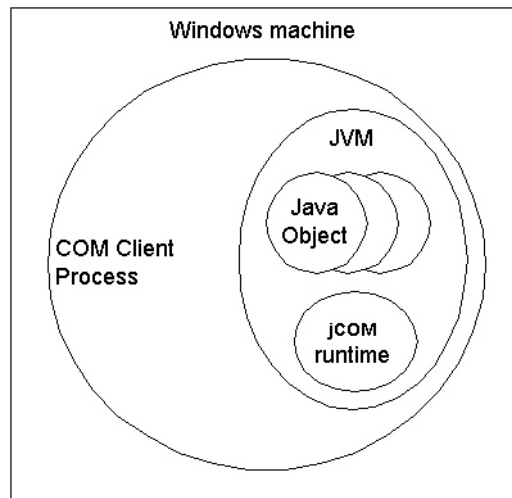
The JVM need not call com.bea.jcom.Jvm.register() or be started as an extra process to the client.

## 5.3.2 The User Interface of the regjvm GUI Tool

Run the `regjvm` tool to display the following dialog box.
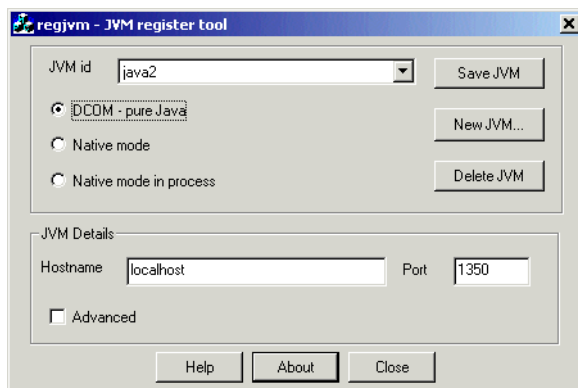
- The top part is for selection and management of all JVMs on the current machine. You can change, add or delete JVMs. Before switching to a different JVM, you must save changes made to the currently selected JVM. The JVM mode you select dictates the information required in the lower half of the screen.

- 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 check box which when selected displays advanced options for each JVM mode.
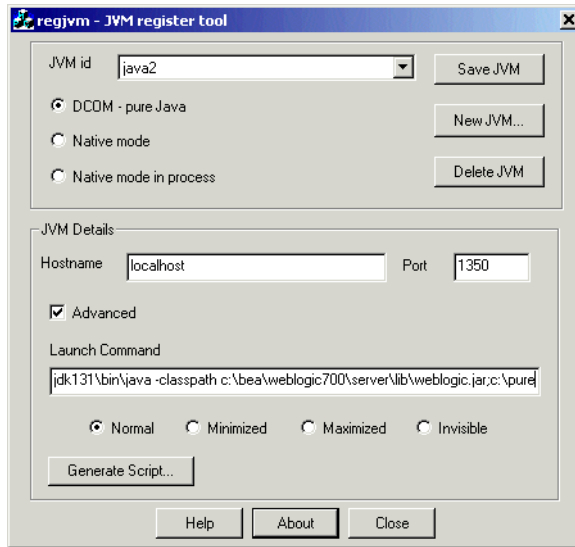
    These options are discussed in the following sections.

5.3.2.1 DCOM Mode Options for the regjvm GUI Tool
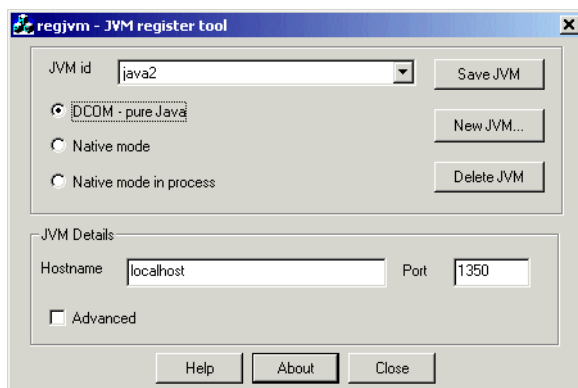
Standard Options

- JVM id (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.

- Hostname—The hostname or IP address where the JVM is located.

- Port—The port number used to initiate contact with the JVM.

Advanced Options

■ Launch command (required)—The command to be used if the JVM is to be automatically launched. Typically this would be something like:

```
c:\Oracle\Middleware\jdk160\bin\java -classpath c:\Oracle\Middleware\wlserver_
10.3\server\lib\weblogic.jar;c:\pure MyMainClass
```

The important thing is that `weblogic.jar` and the appropriate `jdk` files be in your `CLASSPATH`.

■ Generate Script (optional) —Allows the user to generate a registry script selecting the settings of the JVM.

5.3.2.2 Native Mode Options for the regjvm GUI Tool
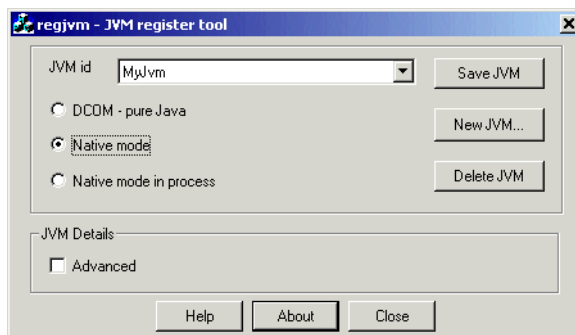
Standard Options

■ JVM id (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.
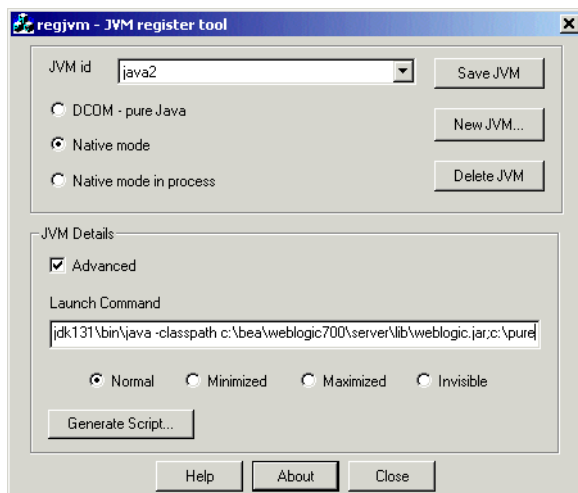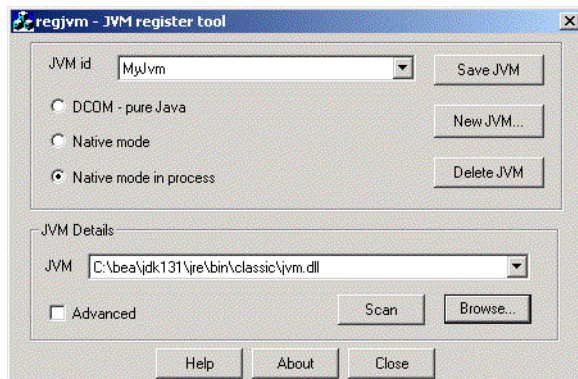
Advanced Options
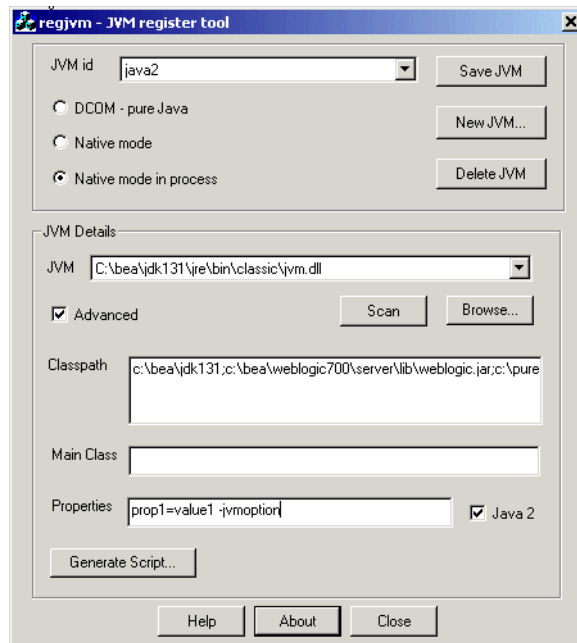
The advanced options are identical to those of DCOM mode. See Section 5.3.2.1, "DCOM Mode Options for the regjvm GUI Tool."

5.3.2.3 Native Mode in Process Options for the regjvm GUI Tool

Standard Options

■   JVM id (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

- 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)—Identical to that of DCOM mode. See Section 5.3.2.1, "DCOM Mode Options for the regjvm GUI Tool."

5.4 regjvmcmd

regjvmcmd is the command line version of the GUI tool, regjvm, discussed in Section 5.3, "regjvm." To get a summary of its parameters, run it without parameters:

```
regjvmcmd
```
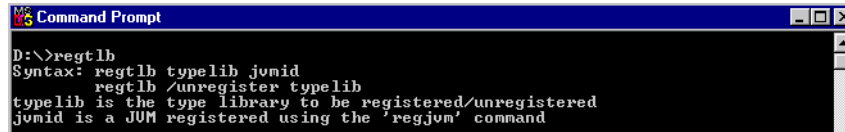
In regjvmcmd's simplest form, you specify the following:

- A jvm ID (corresponding to the name used in com.bea.jcom.Jvm.register("*JvmId*")),

- The binding can 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 WebLogic Server.

If you no longer need to have the JVM registered, or if you wish to change its registration, you must first un-register it with this command:

```
regjvmcmd /unregister JvmId
```

5.5 regtlb

WebLogic jCOM's `regtlb` tool registers a type library on a COM Windows client that needs 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 (through jCOM))
```

If the type library was not generated from a `java2com` generated IDL file, you are 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 instantiates the class 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`.