BEA WebLogic jCOM

Reference Guide

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

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

1. The com2java Tool

1 The com2java Tool

WebLogic jCOM's com2java tool reads information from a type library, and generates Java files that can be used 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.

The following sections look at:

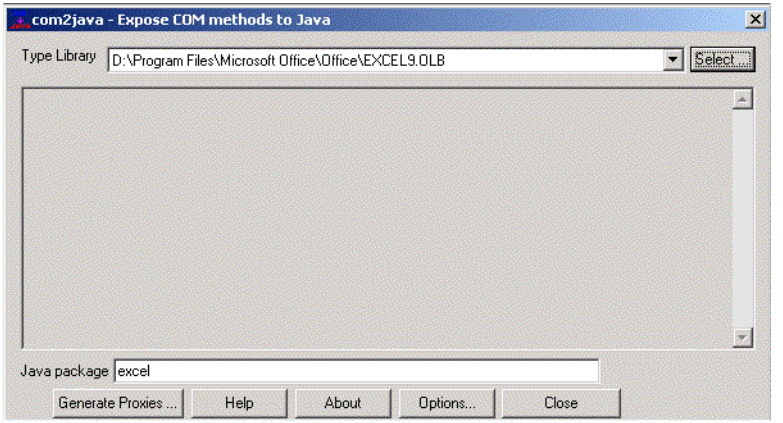
- Using com2java
- Files generated by com2java

Using com2java

The com2java tool is in the *bin* subdirectory under the WebLogic jCOM installation directory. For example if you installed WebLogic jCOM in *c:\bea\wlserver6.1\jcom*, then you will find com2java in *c:\bea\wlserver6.1\jcom\bin\com2java.exe*.

As well as providing the GUI interface described below, a command line interface is provided via the **com2javacmd** tool. Type 'com2javacmd /?' for more information.

When you start com2java, this is the dialog that is displayed:



Selecting the type library

Click on 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 DLL containing your COM component.

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

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. You will probably want to have 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*.

Fill in the `Java package` text box with the name of the package to which you would like the generated files to belong.

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

Options

By clicking on the options button a dialog box with options for com2java appears. Note that these options are saved automatically between sessions of com2java. If you only require an option for one particular generation, then you must remember to reset the option after generating the proxies. The options are:

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 will prefix 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. If you would prefer the tool not to do so, uncheck the `Lowercase method names` checkbox 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 the above error. Selecting this option will ensure that the generated code retries each time this error code is received.

Generate Arrays as Objects

Parameters which 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 will assume 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 will import another type library. If you are also generating proxies for imported type libraries, using this option will prompt 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 depreciated constructors

Generated proxies contain some constructors which are now depreciated. If you do not wish to generate these depreciated constructors at all 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, then selecting this option will cause the corresponding generated Java classes to not implement the additional interfaces. You can still access the interfaces using the getAsXXX method that is generated. See the generated comments.

Generate Java AWT classes

Generates .Java Classes as GUI classes. To be used for embedding ActiveX controls in Java Frames.

Generate the proxies

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

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

Files generated by com2java

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

- Enumerations
- COM Interfaces (including source interfaces - events)
- COM Classes

You will need to refer to documentation associated with the COM objects that you are accessing in order 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, such as the Excel 2000 COM API:

http://msdn.microsoft.com/library/default.asp?URL=/library/officedev/off2000/xltocobjectmodelapplication.htm

Sometimes it is not easy to relate the generated files to the original API. If this is the case please do not hesitate to contact us. Ideally if you can give us a piece of code that works from another COM client (early-bound VB, VC++, etc.) then we will try to help you do the same from Java.

Enumerations

If a type library contains an enumeration, WebLogic jCOM will generate a Java interface containing constant definitions for each element in the enumeration.

COM Interfaces

There are two kinds of COM Interfaces that WebLogic jCOM handles. It handles Dispatch interfaces, whose methods can only be accessed using the COM IDispatch mechanism, and handes 'dual' interfaces, whose methods can also be invoked directly (vtbl access).

For each 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 will generate 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 -- you will never need to know anything about them, or use them.

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.

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 be used to access the corresponding COM class.

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.

Although COM events work using Connection Points, and source interfaces, Java has a different event mechanism. The com2java tool totally 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 notices 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 which will be subscribed to the event need not implement all of the methods in the interface, which can be especially painful with large interfaces.

So for each event interface, WebLogic jCOM generates an adapter class.

Summary - Events

Please note that 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.

2 WebLogic jCOM and COM Event

If a COM Class can generate events (has a `source` interface), then WebLogic jCOM's com2java tool, will generate Java code which allows Java objects to subscribe to the event using standard Java semantics. For information on the generated files, please read the relevant section of the com2java documentation.

In summary, if a COM Class has a source interface that is defined in the same type library as the class, then WebLogic jCOM's com2java tool will do the following:

- it will generate an *addXYZListener* method and a *removeXYZListener* method in the corresponding Java class that it generates to access the COM Class
- it will derive the event interface from *java.util.EventListener*
- for each method in the event interface, it will create a class derived from *java.util.EventObject*, which contains all of the original method parameters, an instance of which is passed as the sole parameter to the method
- it will generate an adapter class which implements the event interface, and provide default empty implementations of each method in the interface

From a Java programmer's perspective there is nothing really else to it. You use the standard Java event techniques -- you implement the event listener interface, and then subscribe to the event using the *addXYZListener* method.

3 Exceptions

WebLogic jCOM transparently maps between COM exceptions and Java exceptions. The following sections look at:

- Exceptions raised in COM components
- Exceptions in Java objects
- Using the Exception Interception mechanism

Exceptions raised in COM components

When a Java client invokes a method in a COM component, that component may raise an exception.

COM Exceptions have an associated error number, a source, and a description.

Raising exceptions in a COM component

In Visual BASIC, an exception can be raised like this:

```
Public Sub errorMethod()
       Err.Raise vbObjectError + 1051, "My program", "This is the
description"
End Sub
```

In Visual C++, an exception can be raised like this:

```
AfxThrowOleDispatchException(0, "I am sorry Dave, I can't do
that...");
```

Catching them in Java

When a COM component throws an exception, an instance of com.bea.jcom.AutomationException is thrown in the Java client. The javadoc documentation associated with that class is here.

The AutomationException class exposes methods to obtain the error number, source and description of the COM exception that was thrown.

```
import com.bea.jcom.AuthInfo;
import java.net.UnknownHostException;
import java.io.IOException;
public class Example {

       public static void main(java.lang.String[] args) throws
IOException, UnknownHostException {

              vbexcep.Class1 c1 = null;

              try {
              c1 = new vbexcep.Class1();
              c1.errorMethod();
              } catch(com.bea.jcom.AutomationException ae) {
              System.out.println("Caught: " + ae);
              System.out.println("Source: " + ae.getSource());
              System.out.println("Description: " +
ae.getDescription());
              System.out.println("Code: " + ae.getCode());
              } finally {
              com.bea.jcom.Cleaner.releaseAll();
              }
       }
}
```

The result of running the example:

```
D:\temp>java Main2
Caught: AutomationException: 0x8004041b - This is the description  in 'My Program '
Source: My Program
Description: This is the description
Code: 2147746843

D:\temp>
```

Use IOException rather than AutomationException?

If you don't like the idea of having to deal with the AutomationException class, then you can catch java.io.IOException instead, since AutomationException derives from IOException. In this way you can reduce the dependancy of your code on COM related classes, at the expense of not being able to access the error code, description and source.

Exceptions raised in Java objects

COM components can invoke methods in Java objects using WebLogic jCOM.

If such a method generates an exception, then WebLogic jCOM will catch the exception, and translate it into an appropriate COM exception.

Take, for example, the following Java code. Note that the 'method1' method is not particularly sound:

```
import java.io.*;
public class Simple {
public static void main(String[] args) throws Exception {

com.bea.jcom.Jvm.register("firstjvm");
Thread.sleep(6000000); // Sleep for an hour
}

public void method1() throws java.io.IOException {
throw new java.io.IOException("A deliberate exception");
}
}
}
```

This is the Visual Basic client code:

```
Public Sub method1(ByVal p1 As Object)
Set p1 = GetObject("firstjvm:Simple")
On Error GoTo ErrorHandler
p1.method1

ErrorHandler: ' Error-handling routine.
MsgBox Err.Source, vbInformation, "Source"
MsgBox Err.Description, vbInformation, "Description"
MsgBox Str(Err.Number), vbInformation, "Code"
```

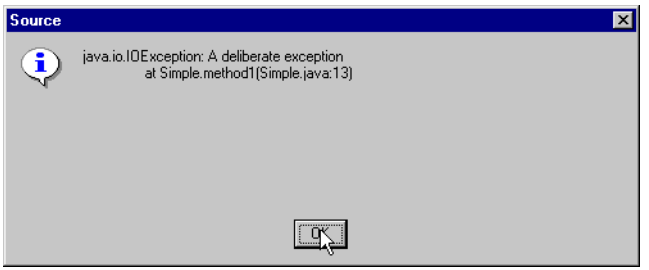
```
End Sub
```

The Visual BASIC code simply establishes an error handler, and then makes a call into the Java object. The Java object in our example deliberately generates an exception which is caught by the Visual BASIC error handler.

The error handler displays a series of message boxes, which give information on the error:

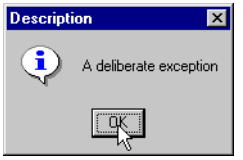
The source

WebLogic jCOM automatically fills in the *source* with the *stack trace of method which generated the error*:



The description

WebLogic jCOM automatically fills in the *description* with the string returned from *Exception.getMessage()*:



We have erased the part of the stack trace showing WebLogic jCOM internal methods.

The code

WebLogic jCOM automatically sets the code to 0x80020009, which is COM for 'Exception occurred.':

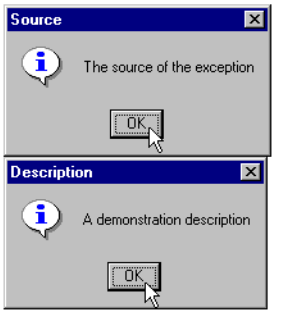
What if you want to specify the source/description/code yourself?

If you wish to explicitly set the error information yourself, rather than accepting WebLogic jCOM's default, then you can throw an instance of com.bea.jcom.AutomationException.

If we change the Java code in the above example:

```
public void method1() throws com.bea.jcom.AutomationException {
long code = 0x80020009;
String source = "The source of the exception";
String description = "A demonstration description";
throw new com.bea.jcom.AutomationException(code, source,
description);
}
```

The following message boxes get displayed, with the information that was explicitly set:



Using the Exception Interception Mechanism

We also provide a hook which allows you to provide code which is called when an exception is about to be returned to a COM client. This lets you change the exception information, or store the exception object for later retrieval by a COM client.

To intercept exceptions generated when calling from COM-to-Java, register an interceptor once, like this:

```
com.bea.jcom.ExceptionInterceptor interceptor = new
YourInterceptor();
com.bea.jcom.AutomationException.setExceptionInterceptor
(interceptor);
```

Create a class which implements the ExceptionInterceptor interface like this:

```
class YourInterceptor implements com.bea.jcom.ExceptionInterceptor
{

       public com.bea.jcom.AutomationException
handleException(Throwable t) {

    ...

  }

}
```

If you want the default error to be returned to the COM client then simply return null in the handler, otherwise you can generate a specific error, for example:

```
public com.bea.jcom.AutomationException handleException(Throwable
t) {

       System.out.println("Intercepting: " + t);
       long code = 0x80020009;
       String source = "The source of the exception";
       String description = "A demonstration description";
       return new com.bea.jcom.AutomationException(code, source,
description);
}
```

4 Garbage Collection and Reference Counting

On COM objects referenced from Java clients

The Java Virtual Machine will perform garbage collection on Java references to a COM object when such references can no longer be accessed.

When such a reference is garbage collected, WebLogic jCOM adds the DCOM object details to an internal list of DCOM object references which should be released. Every ten seconds a WebLogic jCOM daemon thread releases these batched DCOM object references through garbage collection.

If you would prefer to **explicitly release** an object reference yourself, then call the *com.bea.jcom.Cleaner.release(...)* method, passing the object reference as a parameter.

When your JVM is about to shut down you should call *com.bea.jcom.Cleaner.releaseAll()*. This will release any COM object references that have not already been released through garbage collection. Once you have called this method you will no longer be able to make use of any COM object accessed via WebLogic jCOM.

When running in DCOM mode the WebLogic jCOM runtime sends DCOM ping messages per the DCOM protocol to tell the COM server that the client is still alive.

On Java objects referenced from COM clients

When COM clients hold references to a Java object the WebLogic jCOM runtime maintains a reference inside the JVM to that Java object on behalf of the COM clients. It also keeps count of the number of COM references exported to that Java object and releases its reference when the COM reference count reaches zero. When running in DCOM mode WebLogic jCOM receives DCOM ping messages informing it that a COM client is still alive. If no such ping messages are received for six minutes (per the DCOM specification) then the WebLogic jCOM runtime releases all unpinged Java objects.

If you wish to be notified when Java objects are no longer referenced by COM clients, you can call the following method passing in a reference to an instance of a Java class you create that implements the `com.bea.jcom.Unreferenced` interface:

```
com.bea.jcom.Cleaner.addUnreferencedListener(com.bea.jcom.Unrefer
enced listener)
```

The `Unreferenced` interface looks like this:

```
public interface Unreferenced {
public void objectUnreferenced(Object o);
}
```

When you no longer wish to be notified, call:

```
public static void removeUnreferencedListener(Unreferenced
listener)
```

This is a small example:

```
public class MyJvm {

public static void main(String[] args) throws Exception {
com.bea.jcom.Jvm.register("firstjvm");

MyUnreferencedListener l = new MyUnreferencedListener();
com.bea.jcom.Cleaner.addUnreferencedListener(l);

Thread.sleep(6000000); // Sleep for an hour

com.bea.jcom.Cleaner.removeUnreferencedListener(l);
}
}
```

```
class MyUnreferencedListener implements com.bea.jcom.Unreferenced
{
public void objectUnreferenced(Object o) {
System.out.println("** Object no longer referenced: " + o);
}
}
```

5 Native Mode

The Java-COM bridge not only supports the use of network based DCOM to allow Java objects to interact with COM objects, WebLogic jCOM also gives you an alternative: native code (DLLs) can be used to perform the bridging.

Your Java code stays the same whether it is using native mode or DCOM mode.

Note: By default WebLogic jCOM uses DCOM; you have to explicitly enable native mode.

The following sections look at:

- What Is Supported?
- How to run Java clients in native mode
- How to run COM clients in native mode (JVM out-of-process)
- How to run COM clients in native mode (JVM in-process)
- Unsupported features and known issues

What is supported?

Both IDispatch and vtable

The native mode allows both IDispatch and Custom (vtable) method invocations, in both directions. The COM interfaces do not have to be dual (they can derive directly from IUnknown).

In-process and out-of-process

WebLogic jCOM's native mode supports:

- Java clients talking to out-of-process (including remote) COM components
- Java clients talking to in-process COM components, where the COM component's DLL is loaded in-process into the JVM's process
- COM clients talking to Java objects where the JVM is running in a separate process (on the same machine)
- COM clients talking to Java objects where the JVM is loaded in-process, into the address space of the COM client

All JVMs

WebLogic jCOM can be used with any JVM running on any platform.

Microsoft Transaction Server / COM+

We have successfully loaded a Java object into MTS (in-process) and invoked methods on it from a VB base client.

We have made the Java object implement the standard IObjectControl COM interface (which it saw as a normal Java interface, generated by 'com2java'), and MTS invoked the usual methods (activate, canBePooled, etc.).

Finally using a special hook in the WebLogic jCOM runtime we have accessed the IObjectContext MTS/COM+ interface from the Java object and tested some of the attributes -- it correctly detected when it was operating in a transaction, for example.

How to run Java clients in native mode

In order to let a Java client access a COM object using native mode, run the 'com2java' tool on the COM object's type library, and generate the Java proxies as you would when using WebLogic jCOM in DCOM mode.

When running the JVM, define the JCOM_NATIVE_MODE property to enable native mode:
java -DJCOM_NATIVE_MODE JCOMBridge

Pass a host name to the COM object's constructor to create a remote COM component.

The WebLogic jCOM runtime first attempts a CoCreateInstanceEx with flags set to CLSCTX_ALL. If this fails, it retries with flags set to CLSCTX_SERVER.

Threading Models

By default the WebLogic jCOM runtime initializes COM using the COINIT_MULTITHREADED flag. If you wish to have COM initialized using a different flag, set the JCOM_COINIT_VALUE property. For example:

```
java -DJCOM_NATIVE_MODE -DJCOM_COINIT_VALUE=2 JCOMBridge
```

If you get a Class Not Registered Message when using native mode and trying to talk to a COM component hosted in a DLL, try setting the JCOM_COINIT_VALUE to 2 as above.

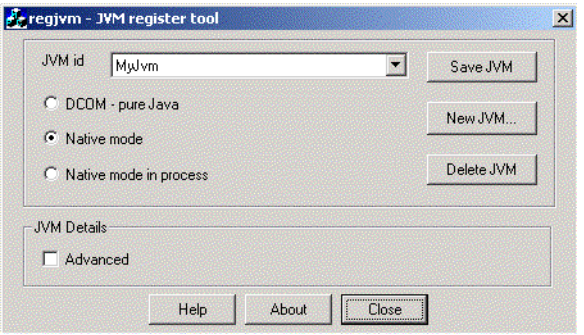
The WebLogic jCOM runtime attempts to put all COM object references into a special COM table called the Global Interface Table (GIT). Whenever a call is made from Java-to-COM the object reference is retrieved from the GIT, which ensures that the call can be made from the current thread.

Unfortunately some COM object references (such as the Frames collection in MSHTML) cannot be placed in the GIT, and so WebLogic jCOM stores a direct pointer reference. In this situation you may have to ensure that only the creating thread makes calls onto that object. This situation is very rare, and a message is logged to the WebLogic jCOM log (if logging is enabled) to let you know that it has happened.

How to run COM clients in native mode (JVM out-of-process)

Prior to reading this section, please read through (and ideally run) the standard DCOM Java examples (VB to Java early and late binding).

If you want your JVM to run out of process (but allow COM client access to the Java objects contained therein using native code), you must use the 'regjvm' command to register it as being native. The regjvm command sets up various registry entries to facilitate WebLogic jCOM's COM-to-Java mechanism:



In your *main* you then tell the WebLogic jCOM runtime that the JVM is ready to receive calls by calling *com.bea.jcom.Jvm.register("MyJvm")*.

You would then start your JVM:

```
java -DJCOM_NATIVE_MODE YourMain
```

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

```
Set aHashtable = GetObject("MyJvm:java.util.Hashtable")
```

"MyJvm" is just a string to identify the JVM -- you can use anything.

This would only work if the JVM were already running. Additional parameters to the regjvm command can specify a command to be used to launch the JVM if it is not already running.

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).

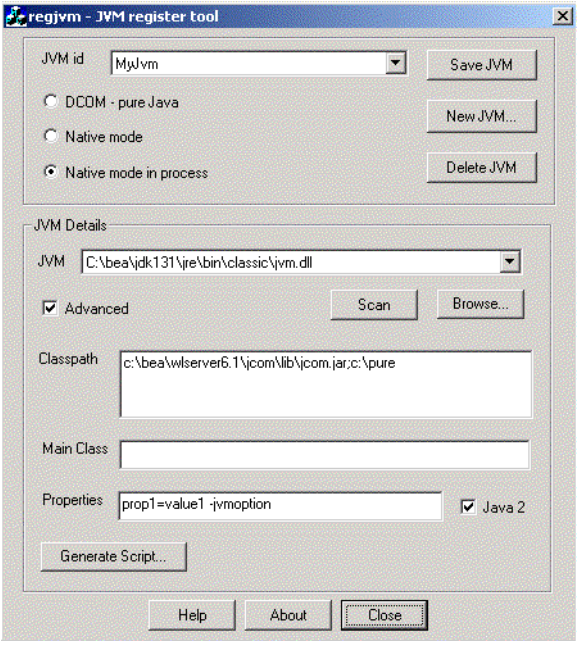
You can also control the instantiation of Java objects on behalf of COM clients by associating your own *Instanciator* with a JVM (additional parameter to com.bea.jcom.Jvm.register(...)) -- a kind of object factory. This is used in most of the standard COM->EJB examples in the WebLogic jCOM documentation.

How to run COM clients in native mode (JVM 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.

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 `d:\bea\jdk131`, and WebLogic jCOM is installed in `d:\bea\wlserver6.1\jcom`, and your Java classes are in `c:\pure`, you would use



As you can see, you specify the JVM name, the CLASSPATH, and the JVM bin directory path.

From VB you should now be able to do:

```
MessageBox GetObject("MyJvm:java.util.Hashtable")
```

If you wish to specify properties to be set for the JVM, add them at the end of the regjvm command line, in the form `name=value`, separated by spaces (don't use `-Dname=value`). For example if you run into problems, enable logging by adding `JCOM_LOG_LEVEL=3` and `JCOM_LOG_FILE=c:\temp\jcom.log` to the end of the 'regjvm' command. The mechanism described above (calling com.bea.jcom.Log.logImmediately(...)) only works when your Java class has had a chance to run -- there may be a problem before then.

If you get the E_NOMONIKER error, please enable logging in the WebLogic jCOM moniker (jintmk.dll) and examine the classpath that is being used to ensure it has any JVM classes that are required, as well as the WebLogic jCOM runtime (jcom.jar) and any classes you may need.

Unsupported features and known issues

The following features are not yet supported in native mode, or are bugs. Tell us if you would like them added/resolved as a matter of priority.

1. Structures as parameters

6 Security/Authentication

The following section is split into four sections:

- the first section covers how you can authenticate clients invoking EJBs hosted by WebLogic Server using JNDI authentication
- the second section covers how you can authenticate Java clients invoking methods on COM components
- the third section tells you how you can identify the identity of COM clients calling your Java code
- the fourth section describes a utility method which can be used to authenticate arbitrary NT Domains/Users/Passwords from pure Java clients
- the final section shows you how you can ask to be notified when new connections come in from COM clients

Authenticating using JNDI with WebLogic Server

If your client application will be providing security credentials (e.g. username and password) which are then used by WLS to authenticate access using JNDI authentication, you need to do the following:

1. Enable JNDI authentication in the jCOM bridge.

   The jCOM bridge is what allows COM clients (such as Excel) to access EJB's hosted on WebLogic Server. To provide JNDI Authentication you need to add the appropriate properties before calling `InitialContext`:

```
Context.SECURITY_AUTHENTICATION
Context.SECURITY_PRINCIPAL
Context.SECURITY_CREDENTIALS
```

Look at the jCOM bridge provided with the WebLogic jCOM examples (e.g. *c:\bea\wlserver6.1\jcom\samples\JCOMBridge.java*). You will notice the above three lines of code, uncomment them in the `login` method.

2. Use or obtain a username and password in the client application.

   For example, for VBA:

```
'Access the jCOM bridge (without accessing any jCOM files)
Set objBridge = GetObject("objref:...:")

'Set the username and password for JNDI Authentication
Dim bridge As Object
Set bridge = objBridge.get("ejb:JCOMBridge")
bridge.login "newUsername", "newPassword"

'Bind the EJB AccountHome object via JNDI
Set mobjHome = objBridge.get("ejb:beanManaged.AccountHome")
```

   For more information on JNDI Authentication with WebLogic Server, see: http://e-docs.bea.com/wls/docs61/security/prog.html#1024165.

Authenticating Java clients accessing COM components

WebLogic jCOM lets you access COM Components from Java using no authentication, or with the equivalent of `Connect` level authentication.

Running your Java code under Windows

If running under Windows, and you wish WebLogic jCOM to pick up your current identity automatically, simply place the WebLogic jCOM *bin* directory in your PATH environment variable.

Running your Java code on non-Windows platforms

If not running under Windows, or if you don't want WebLogic jCOM to use native code to pick up your current identity then call AuthInfo.setDefault(...) at the start of your program to set the authentication to be used on a process-wide basis when creating and using COM components.

You may override this process-wide default using AuthInfo.setThreadDefault(...), which establishes the authentication to be used for the **current thread**. To clear the authentication established for the current thread, call AuthInfo.setThreadDefault(null).

It is strongly recommended that you call AuthInfo.setDefault(...) to establish the authentication to be used on a JVM-wide basis, so that WebLogic jCOM daemon threads can perform authenticated communications (for example when releasing COM object references that have been garbage collected).

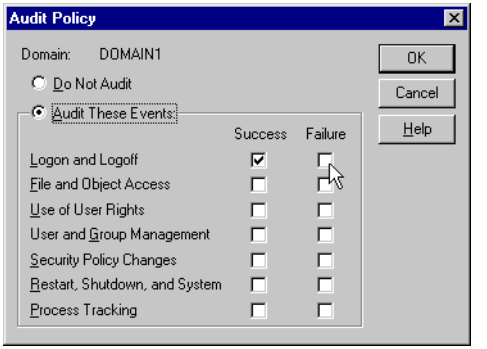
WebLogic jCOM currently only supports **Authentication**, not **Encryption**. If you would like encryption added, then please contact us.

Verifying that authenticated access is taking place

Under Windows NT, you may verify that authenticated access is taking place.

Start the `User Manager for Domains` tool using `Start|Programs|Administrative Tools|User Manager for Domains`, and then display the `Audit Policies` dialog using `Policies|Audit`.

From there you may enable the auditing of logins and logoffs:



Having enabled the auditing, start the `Event Viewer` using `Start|Programs|Administrative Tools|Event Viewer`, and view the security log using `Log|Security`.

WebLogic jCOM will send the name of the host under which it is running, with the string "(jCOM)", to be logged as the workstation name.

Identifying the identity of COM clients calling your Java code

When a COM client invokes methods in your Java object via WebLogic jCOM's DCOM engine, you can call:

- com.bea.jcom.AuthInfo.getCallerDomain() to find the NT domain of the caller, if it could be ascertained. This is a string.
- com.bea.jcom.AuthInfo.getCallerUser() to find the NT domain of the caller, if it could be ascertained. This is a string.
- com.bea.jcom.isCallerAuthenticated() to find out if WebLogic jCOM was able to verify the identity of the caller using the NT Challenge-Response protocol. In order to be able to do this, you must set the JCOM_NTAUTH_HOST property to the IP name of the NT machine that can authenticate the user.

Authenticating NT domain/user/passwords from pure Java software

In order to validate a domain/user/password from a Java program running on a UNIX box (or anywhere) use the static **com.bea.jcom.NTLMAuthenticate.validate(...)** method.

This is the Javadoc associated with the method:

```
public static void validate(String pdcTcpHost,
String domain,
String user,
String password) throws IOException
```

Attempt to authenticate an NT domain/user/password. Works from anywhere that supports Java and requires no native code (just the jcom.jar runtime). No password is transmitted over the network (WebLogic jCOM implements the NT Challenge-Response mechanism). If the domain/user/password are valid then this method simply returns, otherwise a security exception is thrown.

Parameters:
pdcTcpHost - the IP name of an NT machine against which WebLogic jCOM can perform the authentication
domain - the NT Domain name of the user
user - the NT user name of the user
password - the user's password

Throws: SecurityException
if the domain/user/password are not correct

Throws: IOException
if there were problems talking to the NT box against which the authentication was to take place

Note: This method does not have anything to do with our WebLogic jCOM pure Java-COM bridge, and you never need to call it when using WebLogic jCOM to access COM objects from Java, or the reverse.

Since we have implemented the NT Challenge-Response mechanism in pure Java as part of our DCOM engine, it was trivial to expose this method, which may be useful.

Listening for new connections from COM clients

This mechanism is not implemented when running in native mode.

Through WebLogic jCOM's ConnectionListener mechanism you can ask to be notified when new DCOM connections are opened and closed from COM clients, and you can reject incoming connections.

Create a class that implements the com.bea.jcom.ConnectionListener interface, and register an instance of that class by calling com.bea.jcom.Cleaner.addConnectionListener(...).

7 Supported COM data types, and their Java Equivalents

WebLogic jCOM supports the full range of COM Automation types. Our philosophy has been to avoid introducing new Java classes in order to access COM types. This means that COM Variants map onto Objects (java.lang.Long, etc.), and Variants containing arrays map onto Java arrays. You won't find a com.bea.jcom.Variant class that you need to use, for example.

WebLogic jCOM also supports accessing some Java collection types as Visual Basic Collections.

Below you will find

- Accessing Java Objects as Visual Basic Collections
- How COM IDL types map to Java, and VC++
- How COM Variants containing different types map to Java
- Type conversions used during late-bound access from COM to Java
- Accessing COM VariantEnums from Java

Accessing Java Objects as Visual Basic Collections

Visual Basic has a built-in COM class called *Collection*. WebLogic jCOM provides support for accessing instances of java.util.Vector and java.util.List as VB Collections.

Example Visual BASIC code

```
Private Sub Form_Load()
Dim c As Collection
Set c = GetObject("firstjvm:java.util.LinkedList")
c.Add "hello" ' List is now: "hello"
c.Add Now, , "hello" ' List is now: , "hello"
c.Add "Goodbye", , , "hello" ' List is now: now, "hello", "Goodbye"
c.Add "Before", , 3 ' List is now: now, "hello", "Before", "Goodbye"
c.Add "After", , , 4 ' List is now: now, "hello", "Before",
"Goodbye", "After"
For Each e In c
MsgBox e
Next
c.Remove 2 ' List is now: now, "Before", "Goodbye", "After"
c.Remove "Goodbye" ' List is now: now, "Before", "After"
MsgBox c.Item(1)
MsgBox c.Count
End Sub
```

VB	Maps to java.util.Vector	Notes
Collection.Add(*item*)	aVector.addElement(*item*)	
Collection.Add(*item*, , *before*)	aVector.insertElementAt(*before* - 1, *item*)	if *before* is a number
Collection.Add(*item*, , *before*)	aVector.insertElementAt(aVector.indexOf(*before*), *item*)	if *before* is not a number
Collection.Add(*item*, , , *after*)	aVector.insertElementAt(*after*, *item*)	if *after* is not a number

Collection.Add(*item*, , , *after*)	aVector.insertElementAt(aVector.indexOf(*after*) + 1, *item*)	if *after* is a number
Collection.Count	aVector.size()	
Collection.Item(*index*)	aVector.elementAt(*index* - 1)	*index* must be a number
Collection.Remove(*index*)	aVector.removeElementAt(*index* - 1)	if *index* is a number
Collection.Remove(*index*)	aVector.removeElement(*index*)	if *index* is not a number
For Each x in aCollection	x will take on the value of each object in the aVector.elements() enumeration	

VB	**Maps to java.util.List**	**Notes**
Collection.Add(*item*)	aList.target.add(*item*)	
Collection.Add(*item*, , *before*)	aList.add(*before* - 1, *item*)	if *before* is a number
Collection.Add(*item*, , *before*)	aList.add(aVector.indexOf(*before*), *item*)	if *before* is not a number
Collection.Add(*item*, , , *after*)	aList.add(*after*, *item*)	if *after* is not a number
Collection.Add(*item*, , , *after*)	aList.add(aVector.indexOf(*after*) + 1, *item*)	if *after* is a number
Collection.Count	aList.size()	
Collection.Item(*index*)	aList.get(*index* - 1)	*index* must be a number
Collection.Remove(*index*)	aList.remove(*index* - 1)	if *index* is a number

Collection.Remove(*index*)	aList.remove(*index*)	if *index* is not a number
For Each x in aCollection	x will take on the value of each object in the aList.listIterator() iterator	

How COM IDL types map to Java, VB, and VC++

In the following table we show each IDL type, describing its use from Java, from Visual BASIC and from Visual C++.

Table 7-1

IDL	Java	Visual BASIC	Visual C++
[in] VARIANT_BOOL	**boolean**	ByVal boolean	VARIANT_BOOL
[in, out] or [out] VARIANT_BOOL *	**boolean[]** *single element array*	boolean	VARIANT_BOOL*
[in] unsigned char	**byte**	ByVal byte	unsigned char
[in, out] or [out] unsigned char*	**byte[]** *single element array*	byte	unsigned char*
[in] double	**double**	ByVal Double	double
[in, out] or [out] double*	**double[]** *single element array*	double	double*
[in] float	**float**	ByVal Single	float
[in, out] or [out] float*	**float[]** *single element array*	Single	float*

Table 7-1

IDL	Java	Visual BASIC	Visual C++
[in] long	**int** *IDL long is 32 bits*	ByVal Long>	long>
[in, out] or [out] long*	**int[]** *single element array*	Long>	long*
[in] short	**short**	ByVal Integer>	short>
[in, out] or [out] short*	**short[]** *single element array*	Integer	short*
[in] BSTR	**java.lang.String**	ByVal String	BSTR
[in, out] or [out] BSTR*	**java.lang.String[]** *single element array*	String	BSTR*
[in] CY	**long** *CY is fixed point, 64 bit*	ByVal Currency	CURRENCY
[in, out] or [out] CY*	**long[]** *single element array*	Currency	CURRENCY*
[in] DATE	**java.util.Date**	ByVal Date	DATE
[in, out] or [out] DATE*	**java.util.Date[]**	Date	DATE*
[in] IDispatch*	**java.lang.Object**	ByVal Object	IDispatch*
[in, out] or [out] IDispatch**	**java.lang.Object[]** *single element array*	Object	IDispatch**
[in] ISomeInterface*	**ISomeInterface** *Generated Java Interface*	ByVal ISomeInterface	ISomeInterface*
[in, out] or [out] ISomeInterface**	**ISomeInterface[]** *single element array*	ISomeInterface	ISomeInterface**
[in] SomeClass*	**SomeClass** *Generated Java Class*	ByVal SomeClass	SomeClass*
[in, out] or [out] SomeClass**	**SomeClass[]** *single element array*	SomeClass	SomeClass**
[in] IUnknown*	**java.lang.Object**	Specific class/interface	IUnknown*

Table 7-1

IDL	Java	Visual BASIC	Visual C++
[in] Variant	**java.lang.Object**	ByVal Variant	VARIANT
[in, out] or [out] Variant*	**java.lang.Object[]** *single element array*	Variant	VARIANT*
[in] SAFEARRAY(unsigned char)	**byte[]**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(unsigned char)*	**byte[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(VARIANT_BOOL)	**boolean[]**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(VARIANT_BOOL)*	**boolean[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(short)	**short[]**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(short)*	**short[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(long)	**int[]**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(long)*	**int[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(float)	**float[]**	not supported by VB	SAFEARRAY*

Table 7-1

IDL	Java	Visual BASIC	Visual C++
[in, out] or [out] SAFEARRAY(float)*	**float[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(double)	**double[]**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(double)*	**double[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(CURRENCY)	**long[]**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(CURRENCY)*	**long[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(DATE)	**java.util.Date[]**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(DATE)*	**java.util.Date[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(BSTR)	**String[]**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(BSTR)*	**String[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(LPDISPATCH)	**Object[]**	not supported by VB	SAFEARRAY*

Table 7-1

IDL	Java	Visual BASIC	Visual C++
[in, out] or [out] SAFEARRAY(LP DISPATCH)*	**Object[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(VA RIANT)	**Object**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(VA RIANT)*	**Object[][]**	not supported by VB	SAFEARRAY**
[in] SAFEARRAY(LP UNKNOWN)	**Object**	not supported by VB	SAFEARRAY*
[in, out] or [out] SAFEARRAY(LP UNKNOWN)*	**Object[][]**	not supported by VB	SAFEARRAY**

How COM Variants containing different types map to Java

The following table shows how Variants are mapped to Java types.

Variant Containing a	Maps to Java type
VARIANT_BOOL	**java.lang.Boolean**
VARIANT_BOOL*	**java.lang.Boolean[]** *single element array*

Variant Containing a	**Maps to Java type**
unsigned char	**java.lang.Byte**
unsigned char*	**java.lang.Byte[]** *single elment array*
double	**java.lang.Double**
double*	**java.lang.Double[]** *single element array*
float	**java.lang.Float**
float*	**java.lang.Float[]** *single element array*
long	**java.lang.Integer** *IDL long is 32 bits*
long*	**java.lang.Integer[]** *single element array*
short	**java.lang.Short**
short*	**java.lang.Short[]** *single element array*
BSTR	**java.lang.String**
BSTR*	**java.lang.String[]***single element array*
CY	**java.lang.Long**
CY*	**java.lang.Long[]***single element array*
Decimal	**java.math.BigDecimal**
Decimal*	**java.math.BigDecimal[]***single element array*
DATE	**java.util.Date**
DATE*	**java.util.Date[]**
SCODE	**java.lang.Long**
SCODE*	**java.lang.Long[]** *single element array*
IDispatch*	**java.lang.Object**
IUnknown*	**java.lang.Object**
Single dimensional array of VARIANT_BOOLs	**boolean[]**

Variant Containing a	Maps to Java type
Single dimensional array of unsigned chars	**byte[]**
Single dimensional array of doubles	**double[]**
Single dimensional array of floats	**float[]**
Single dimensional array of longs	**int[]** *IDL long is 32 bits*</TD
Single dimensional array of shorts	**short[]**
Single dimensional array of BSTRs	**java.lang.String[]**
Single dimensional array of CYs	**long[]**
Single dimensional array of DATEs	**java.util.Date[]**
Single dimensional array of SCODEs	**long[]**
Single dimensional array of IDispatch*s	**java.lang.Object[]**
Single dimensional array of IUnknown*s	**java.lang.Object[]**
Single dimensional array of Variants	**java.lang.Object[]**
Two dimensional array of Variants	**java.lang.Object[][]**

You can also pass a Variant of type VT_EMPTY or VT_NULL by passing com.bea.jcom.EmptyVariant.TYPE and com.bea.jcom.NullVariant.TYPE.

Type conversions used during late-bound access from COM to Java

These tables show the Java expression used to convert a VB value 'vb' of the VB type on the left to the Java type at the top. Note that conversions marked with a ***** can also raise *Type Mismatch Error*.

Java boolean, byte, short and int

Java(across) VB (down)	boolean	byte	short	int
Boolean	natural	vb ? 1 : 0	vb ? 1 : 0	vb ? 1 : 0
Byte(0..255)	vb != 0	new Integer(vb).byteValue()	new Integer(vb).shortValue()	new Integer(vb).intValue()
Integer(16 bit)	vb != 0	new Short(vb).byteValue()	natural	new Short(vb).intValue()
Long(32 bit)	vb != 0	new Integer(vb).byteValue()	new Integer(vb).shortValue()	natural
Single	vb != 0.0F	new Float(vb).byteValue()	new Float(vb).shortValue()	new Float(vb).intValue()
Double	vb != 0.0	new Double(vb).byteValue()	new Double(vb).shortValue()	new Double(vb).intValue()
Currency(64 bit)	vb != 0	new Long(vb).byteValue()	new Long(vb).shortValue()	new Long(vb).intValue()
String	new Boolean(vb).booleanValue()	new Long(vb).byteValue()*	new Long(vb).shortValue()*	new Long(vb).intValue()*

Java(across) VB (down)	boolean	byte	short	int
Date	*Type Mismatch Error*	*Type Mismatch Error*	*Type Mismatch Error*	*Type Mismatch Error*
Object	*Type Mismatch Error*	*Type Mismatch Error*	*Type Mismatch Error*	*Type Mismatch Error*
Variants	As above depending on content			

Java long, char, float and double

VB/Java	long	char	float	double
Boolean	vb ? 1 : 0	vb ? (char)1 : (char)0	vb ? 1.0F : 0.0F	vb ? 1.0 : 0.0
Byte(0..255)	new Integer(vb).longValue()	(char)(new Integer(vb).intValue())	new Integer(vb).floatValue()	new Integer(vb).doubleValue()
Integer(16 bit)	new Short(vb).longValue()	(char)(new Short(vb).intValue())	new Short(vb).floatValue()	new Short(vb).doubleValue()
Long(32 bit)	new Integer(vb).longValue()	(char)(new Integer(vb).byteValue())	new Integer(vb).floatValue()	new Integer(vb).doubleValue()
Single	new Float(vb).longValue()	(char)(new Float(vb).intValue())	natural	new Float(vb).doubleValue()
Double	new Double(vb).longValue()	(char)(new Double(vb).intValue())	new Double(vb).floatValue()	natural
Currency(64 bit)	natural	(char)(new Long(vb).intValue())	new Long(vb).floatValue()	new Long(vb).doubleValue()
String	new Long(vb).longValue()	(char)(new Long(vb).intValue())	new Long(vb).floatValue()	new Long(vb).doubleValue()

VB/Java	long	char	float	double
Date	*Type Mismatch Error*	*Type Mismatch Error*	*Type Mismatch Error*	*Type Mismatch Error*
Object	*Error*	*Error*	*Error*	*Error*
Variants	As above depending on content			

Java String, Date and Object

VB/Java	String	Date	Object
Boolean	vb + ""	Error	new Boolean(vb)
Byte(0..255)	vb + ""	Error	new Byte(vb)
Integer(16 bit)	vb + ""	Error	new Short(vb)
Long(32 bit)	vb + ""	Error	new Integer(vb)
Single	vb + ""	Error	new Float(vb)
Double	vb + ""	Error	new Double(vb)
Currency	vb + ""	Error	new Long(vb)
String	natural	new Date(vb)	vb (as String_
Date	vb.toString()	Natural	vb (as Date)
Object	vb.toString	Error	vb
Variant	As above depending on content		

Arrays

If a Java method has a parameter which is an array, then if you pass a VB parameter by value to that method you will get an error. If you pass a value by reference, and if the type matches precisely (see below), the Java method will receive a single element array whose single element can be modified to change the corresponding VB parameter. If you pass an array, and if the array content type matches precisely, the Java method will receive an array whose elements can be modified to change the corresponding VB parameter.

Java type	VB pass by reference match
boolean[]	Boolean passed by reference
byte[]	Byte passed by reference
short[]	Short passed by reference
int[]	Long passed by reference
long[]	Currency passed by reference
float[]	Single passed by reference
double[]	Double passed by reference
String[]	String passed by reference
java.util.Date[]	Date passed by reference

Arrays:

Java type	Parameter of this VB type will match
boolean[]	Dim anArray(n) as Boolean
byte[]	Dim anArray(n) as Byte

Java type	Parameter of this VB type will match
short[]	Dim anArray(n) as Short
int[]	Dim anArray(n) as Long
long[]	Dim anArray(n) as Currency
float[]	Dim anArray(n) as Single
double[]	Dim anArray(n) as Double
String[]	Dim anArray(n) as String
java.util.Date[]	Dim anArray(n) as Date
Object[]	Dim anArray(n) as String
Object[]	Dim anArray(n) as Date
Object[]	Dim anArray(n) as Object
Object[]	Dim anArray(n) as Variant

Accessing COM VariantEnums from Java

Some COM methods return a COM VariantEnumeration type. WebLogic jCOM's java2com tool automatically converts the returned type into a standard Java java.lang.Enumeration. There isn't a perfect match though, since COM Enumerations have no equivalent to the 'hasMoreElements()' call, meaning that you have to keep on doing 'nextElement' until you get a 'NoSuchElementException'.

We do provide a workaround -- if you set the **JCOM_PREFETCH_ENUMS** Java property, WebLogic jCOM will pre-fetch the next element behind the scenes, and thus allow the 'hasMoreElements()' method to work properly.

8 Threading

From Java to COM

WebLogic jCOM has been designed to handle multiple simultaneous requests to COM objects from Java.

The WebLogic jCOM runtime DCOM engine will allow multiple simultaneous requests to the same COM object, as well as to different COM objects, and since WebLogic jCOM talks DCOM, and DCOM is layered over DCE RPC, WebLogic jCOM will split large method invocations into multiple RPC PDUs, and reassemble responses that have been split up.

Whether the remote COM object correctly handles multiple simultaneous requests is entirely dependant on its threading model and implementation.

From COM to Java

WebLogic jCOM maintains a pool of threads for handling requests from COM objects. By default, WebLogic jCOM allows a maximum of twenty threads for handling such requests, however you may change the maximum by setting the **JCOM_MAX_REQUEST_HANDLERS** property to the number you want, as in:

```
java -DJCOM_MAX_REQUEST_HANDLERS=50 YourMainProgram
```

WebLogic jCOM creates such threads as and when they are required. That is to say, it does not immediately create twenty threads on start up. It creates the first thread when the first remote request comes in. If a second request comes in, and the first thread is busy servicing a request, then WebLogic jCOM will create a second thread to handle the second request, and so on, up to the maximum of twenty (by default).

If you are interested in seeing WebLogic jCOM's thread allocation mechanism in action, you can run WebLogic jCOM with internal logging enabled (set the JCOM_LOG_LEVEL property to 3, the maximum). The following Visual BASIC code makes a callback to a Java object (p1 is a Java object):

```
Public Sub method1(ByVal p1 As Object)
p1.aMethod
End Sub
```

This is an excerpt from the log, showing what happens. The *IDispatch::GetIDsOfNames* is handled internally by WebLogic jCOM, using reflection.

```
13:05:20+: ObjectExporter0 received an unfragmented request with
call ID 1
13:05:20+: Maximum number of request handler threads is set to 20
13:05:20+: There are 0 request handler threads, of which 0 are
currently busy.
13:05:20+: Creating a new request handler thread.
13:05:20 : IDispatch::GetIDsOfNames request on ExcepDemo@1f230b for
AMETHOD. Returning memid 9
13:05:20+: ObjectExporter0 sending 44 bytes
13:05:20+: ObjectExporter0 read 192 bytes
13:05:20+: ObjectExporter0 received an unfragmented request with
call ID 2
13:05:20+: There are 1 request handler threads, of which 0 are
currently busy.
13:05:20 : IDispatch::Invoke request received for public void
ExcepDemo.aMethod() on ExcepDemo@1f230b
```

If you wish to prevent the same method in your Java object from being invoked more than once at the same time by COM objects, then simply use the standard Java *synchronized* keyword, which will cause methods invocations to block if another object is already invoking the method.

9 Using COM Objects from Java

In this section we describe how you can create an instance of a COM object, and use it from Java using WebLogic jCOM. We also explain how to deal with new references to COM objects that are returned from method calls to COM objects:

- Java classes generated from COM classes by com2java
- Java interfaces & classes generated from COM interfaces by com2java

You may wish to read through the documentation on the com2java tool prior to reading this section of the documentation.

Java classes generated from COM classes by com2java

For each COM Class that the com2java tool finds in a type library, it generates a Java class which can be used 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 be used to wrap a returned object reference which is known to reference an instance of the COM Class

The default constructor

The default constructor can be used to create an instance of a COM object on the local machine using the default authentication if it has been set, or using no authentication if no default has been defined.

The second constructor (String host)

The second constructor can be used to create an instance of a COM object on a specific host using the default authentication if it has been set, or using no authentication if no default has been defined.

The third constructor (AuthInfo authInfo)

The third constructor can be used to create an instance of a COM object on the local host using the specified authentication.

The fourth constructor (String host, AuthInfo authInfo)

The fourth constructor can be used to create an instance of a COM object on the specified host using the specified authentication.

The final constructor (Object objRef)

This final constructor does not actually create a new instance of the COM Class. Instead, it can be used to access a reference to the COM class that was returned from another COM Class (from a method call, or via a property or event).

If a method or property returns a reference to a COM Class, then the com2java tool will automatically generate a method signature which returns the appropriate Java Class, if the returned COM Class is defined in the same type library.

Java interfaces & classes 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 method internally makes use of 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);
```

10 Java Properties in jCOM.jar

The following table documents the properties which can be set for the jcom.jar:

jCOM Property	Description
ENABLE_TCP_NODELAY	TCP/IP Connections have TCP_NODELAY set when running in DCOM mode. See the standard Java API documentation for an explanation of what this actually does.
JCOM_DCOM_PORT	Specifies the TCP/IP port jCOM uses to receive incoming DCOM requests (actually uses two ports).
JCOM_COINIT_VALUE	Sets the COM mode to be used when WebLogic jCOM initialises COM for new threads using CoInitializeEx.
JCOM_INCOMING_CONNECTION_TIMEOUT	This causes any incoming connections which have not been used for specified number of milliseconds to disconnect.

jCOM Property	Description
JCOM_OUTGOING_CONNECTION_TIMEOUT	This causes any outgoing connections (connections initiated by the WebLogic jCOM runtime) which have not been used for specified number of milliseconds to disconnect.
com.bea.jcom.server	Normally when running in DCOM mode theWebLogic jCOM runtime includes all the IP addresses for the local machine in DCOM object references to Java objects that it passes over to COM. Setting this property limits the IP address to that specified (also uses RMI_LOCAL_HOST but com.bea.jcom.server overrides it).
JCOM_MAX_REQUEST_HANDLERS	See the *Multi-Threading* section of the WebLogic jCOM documentation.
JCOM_NATIVE_MODE	Tells theWebLogic jCOM runtime to run in Native mode (the value associated with the property is ignored).

jCOM Property	Description
JCOM_NOGIT	Tells the WebLogic jCOM runtime to not use the COM Global Interface Table construct when running in in native mode. Setting this property means that WebLogic jCOM stores object reference pointers directly, instead of GIT cookies, which in turn means that you may have problems passing object references between threads.
JCOM_NTAUTH_HOST	Set this property to the name of a NT server machine that WebLogic jCOM can talk to to authenticate incoming DCOM calls. See the *Security* section of the WebLogic jCOM reference guide.
JCOM_LOCAL_PORT_START	Tells the WebLogic jCOM runtime what *local* port to try to use when opening DCOM connections to COM servers -- if the specified port is already in use, it tries the next one up, until it reaches JCOM_LOCAL_PORT_END .
JCOM_LOCAL_PORT_END	Tells the WebLogic jCOM runtime the upper limit to use for *local* ports to use when opening DCOM connections to COM servers (see JCOM_LOCAL_PORT_START).

jCOM Property	Description
JCOM_PROXY_PACKAGE	Used when the proxies generated by java2com have been put into a specific package (using an undocumented mechanism).
JCOM_SKIP_CLOSE	WebLogic jCOM will let peer close sockets when running in DCOM mode.
JCOM_WS_NAME	Sets the name that an NT machine will see as the client's workstation name when the client is running in DCOM mode, using AuthInfo.setDefault(...), or AuthInfo.setThreadDefault(...)

11 Troubleshooter

The following sections outline some common problems with WebLogic jCOM and useful tips to workaround them:

- Improving WebLogic jCOM's Performance
- DOS Errors
- Java Errors
- Visual BASIC Errors
- Visual C++ Errors
- DCOM Security problems when running under certain environments
- The IID*.java wrappers generated by java2com do not compile
- The Error I am getting is not listed

Improving WebLogic jCOM's performance

Use the reduced logging runtime

The easiest way to improve performance is to use the *reduced logging* WebLogic jCOM runtime: instead of putting *jcom.jar* in your CLASSPATH, use *jcom_reduced_logging.jar*, which is in the same directory.

Using this runtime inhibits the ability to generate WebLogic jCOM log files to help resolve issues, but it can in some circumstances result in an extremely substantial performance gain.

Run in native mode

If you are running in WebLogic jCOM's DCOM mode (the default) and your JVM is running under MS Windows, you might want to think about switching to Native Mode, which makes the WebLogic jCOM runtime use native code instead of DCOM to interact between Java and COM. You will not have to change your code at all - this switch to native mode is made simply by defining a runtime property.

See the section on Native Mode in the WebLogic jCOM reference guide, for more information.

DOS Errors

If the error is

- **DOS: The *java* command is not recognised**

Then your PATH environment variable does not include the JDK *bin* directory.

If the error is

- **DOS (while compiling using the *java* command): java.lang.OutOfMemoryError**

Tell the compiler to use more memory. Compile using:
java -Jmx64m -J-ms64m YourProgram.java

If the error is

- **DOS: The name specified is not recognized as an internal or external command, operable program or batch file.**

when trying to use **regtlb** or **regjvm**

Then your PATH environment variable does not include the WebLogic jCOM's *bin* directory.

Java Errors

If the error is

- **Java: Can't find class com/bea/java2com/Main**
- **Java: Package com.bea.jcom not found in import.**
- **Java: java.lang.NoClassDefFoundError: com/bea/jintact/Helper**

Then your CLASSPATH environment variable does not include the WebLogic jCOM runtime (*jcom.jar*).

If the error is

- **Java: AutomationException: 0x80080005 - Server execution failed. Note that Windows 95 does not support automatic launch of a server, it must be running already**

If running under Windows NT then check the NT event log (**Start|Programs|Adminstrative Tools|Event Viewer->Log|System**), which will often give you the reason for the launch failure.

A typical cause of this error is that the PATH in the registry for the server is incorrect.

There are also a couple of NT bugs which can also cause this error: http://support.microsoft.com/support/kb/articles/q185/1/26.asp and http://support.microsoft.com/support/kb/articles/q158/5/08.asp

If the error is

- **Java: AutomationException: 0x80010001 - Call was rejected by callee. in 'Invoke'**

You may well get this when running Excel. We are assuming that this is an Excel restriction. From one of the Microsoft support answers: *This error usually occurs when the server application is too busy to respond to the client.*

The 'com2java' tool has an option in the Options dialog box which causes the WebLogic jCOM code to retry if this error occurs -- check that option and regenerate the proxies.

If the error is

- **Java: AutomationException: 0x80020003 - Member not found in 'IDispatch::invoke'**

If you are using 'ocxhost' and attempting to load an ActiveX Control, then it is likely that the prog ID you are specifying is incorrect, or the control has not been properly registered on your machine.

You can find the correct progid by using the 'checkconfig' command:

```
checkconfig /typelib  config.log
```

If you look at the generated config.log file you will see the progid to use.

If the error is

- **java.lang.ClassNotFoundException**

Doublecheck that your CLASSPATH is defined correctly. If you are using a Servlet environment or other environment which loads classes in different class loaders, you should ensure that your generated proxies and the jcom.jar file are loaded by the same class loader.

If the error is

- **java.lang.UnsatisfiedLinkError: getNegociateMessage**

Check that your jcom.jar file is being loaded by the system class loader. This means that you are trying to use native authentication but the JNI is not working correctly because only classes loaded by the system class loader can use JNI. To ensure that the you use the system class loader, place the jcom.jar in your CLASSPATH environment variable at system level. See also the previous troubleshooting entry.

If the error is

- **Java: AutomationException: some other error code -**

If there is no error message related to the error code, in the case an AutomationException, you can look up the error code in the Microsoft Developer Network documentation or if you have Visual C++ installed the file winerr.h or the utility error lookup may help.

Visual BASIC Errors

If the error is

- **VB: Run-time error '-2147221020 (800401e3)': Automation Error Invalid Syntax**

You will get this error if you are attempting to instanciate a Java object using:

```
Set o = GetObject("SomeJvmId:SomeJavaClass")
```

where *SomeJvmId* has not been registered on this machine using the *regjvm* command.

If the error is

- **VB: Run-time error '429': ActiveX component can't create object**

You will get this error if the JVM is not running, or not registered correctly, or if the Java class could not be instanciated. Start your JVM with logging enabled using:

```
java -DJCOM_DCOM_PORT=??? -DJCOM_LOG_LEVEL=3 YourMainClassName
```

When you run your VB client, look and see if the WebLogic jCOM runtime receives a connection from the VB client. If no connection is received, then either:

The jvmId used when you are calling *com.bea.jcom.Jvm.register("jvmId")* is wrong

The port you specified when starting the JVM is wrong, or is already in use. You can see which port the WebLogic jCOM runtime uses when you have logging enabled:

929376053810 : OXID Resolver started. Listening on port 1111

You made a mistake when registering the JVM, either you specified the wrong JvmID, wrong host, or wrong port when invoking *regjvm jvmId host[port]* on the VB client machine.

The VB environment is very old, and does not support the "MyJvm:MyClass" parameter format to GetObject. A workaround is to use WebLogic jCOM's 'regprogid' command to map a progid to the JVM/class:

regprogid My.ProgId "MyJvm:MyClass"

Then in your code use GetObject("My.ProgId")

If the error is

- **VB: Run-time error '-2147483644 (8000004)': Automation Error No Such interface supported**

You will get this error when using early-bound access to a COM object, and you attempt to invoke a method on that object. *You have either not compiled the Java wrapper files generated by the 'java2com' tool, or they can not be loaded by the WebLogic jCOM runtime (jcom.jar) because they are not in your CLASSPATH.*

If you are using JDK1.2, you should not put the WebLogic jCOM runtime in the "ext" directory, since if it is in that directory it will unable to load classes in your CLASSPATH (such as the wrapper classes). Instead put the runtime in your CLASSPATH.

You can see this if you start the JVM with WebLogic jCOM logging set to '3' (full).: *java -DJCOM_LOG_LEVEL=3 Main*

Visual C++ Errors

If the error is

- **VC++: 0x80040154 (Class not registered)**

Follow the same steps as for the VB Error 429.

If the error is

- **VC++: 0x80000004**

Follow the same steps as for the VB Error 8000004.

DCOM security problems when running under certain environments

Java applications running under certain environments (servlets/IDEs) may result in DCOM security errors despite the fact that they run successfully from the command line.

The behaviour you are seeing may be caused by one of the following:

1. The Java code is running as a different user. Native authentication picks up the user running the Java code and this user may not have sufficient DCOM access.

2. The jcom\bin directory is not in your PATH at system level or the Servlet/IDE environment has not been restarted to pick up any changes in the PATH.

3. The WebLogic jCOM runtime is not being loaded by the system class loader. Native authentication uses JNI and JNI calls must be made from classes loaded by the system class loader. Some environments allow you to specify a classpath which is loaded by a different class loader. Java files in such a classpath will not be able to use JNI. A safe way to ensure that the system class loader is used is to specify the inclusion of jcom.jar in the classpath environment variable or using the -cp/-classpath/-Djava.class.path options when running java.exe. An additional thing to take into account is that WebLogic jCOM proxies files, and the jcom.jar runtime should be loaded by the same class loader.

If you enable WebLogic jCOM logging as specified in the trouble shooter, issues 2 and 3 will show that the native authentication DLL failed to load or run correctly. WebLogic jCOM logging also shows you which class loader the jcom.jar is run under - bootstrap is the system class loader.

Another solution to the problem may be to use com.bea.jcom.AuthInfo.setDefault(...) which uses pure java authentication (no JNI) and hence will also run on non-Windows platforms.

Also read entries in the trouble shooter and in the Security section of the documentation for further information on security issues.

The IID*.java Wrappers Generated by java2com do not compile

You may get errors like this:

```
tlb\remote\IID6cce3097_00e5_1000_500a_bf09cf1e0000Wrapper.java:2:
IID6cce3097_00e5_1000_500a_bf09cf1e0000Wrapper should
be declared abstract; it does not define getHomeInterfaceClass() in
IID6cce3097_00e5_1000_500a_bf09cf1e0000Wrapper
public class IID6cce3097_00e5_1000_500a_bf09cf1e0000Wrapper extends
com.bea.jcom.Dispatch implements javax.ejb.EJB

MetaData    {
```

or

```
D:\pure\JCOM\TLB\Bean>javac
IID828d8ad4_00e5_1000_502a_bf09cf1e0000Wrapper.java
IID828d8ad4_00e5_1000_502a_bf09cf1e0000Wrapper.java:2:
IID828d8ad4_00e5_1000_502a_bf09cf1e0000Wrapper should be declared
abstract; it does not define toString(boolean) in
IID828d8ad4_00e5_1000_502a_bf09cf1e0000Wrapper
public class IID828d8ad4_00e5_1000_502a_bf09cf1e0000Wrapper extends
com.bea.jcom.Dispatch implements java.security
.Certificate    {
```

By default the java2com tool ignores certain classes and methods, in order to attempt to reduce the number of generated wrappers. It may be that it is ignoring classes such as java.lang.Class (in the first case above), or methods such as toString() (in the second case). You will need to look at the error you are getting in order to determine which class or method is being ignored when it is actually required.

Unregister the type library (if you have already registered it), delete the generated wrappers, re-run java2com, and click on the "Names..." button, and remove the mapping of the required class or method to "".

The error I am getting is not listed

Email support@bea.com and include the following information:

- The WebLogic jCOM version you are using
- The JDK version you are using
- The platforms you are using, including any service releases/option packs installed
- If running your Java software in an Applet/Servlet/Application server, full version details
- The config.log generated by doing *checkconfig config.log*
- The jcom.log generated by enabling logging as described below

Certain configuration problems may be recognized by running the checkconfig tool in the jcom\bin directory. Run "`checkconfig /?`" for further details on usage.

Depending on the nature of the error you are getting, it may be useful to include the extensive logging information which WebLogic jCOM can generate.

To cause the WebLogic jCOM Java runtime to generate logging information to the file *jcom.log* add the call com.bea.jcom.Log.logImmediately. You must call this only **once** in your Java code -- the easiest way of including this is to put the call in a static block in the first Java class you use (when calling from Java to COM or from Java to COM):

```
public class YourClass {

  static {

    com.bea.jcom.Log.logImmediately(3, "c:\\jcom.log");

  }

  ...

}
```

```
If using an Applet then you can cause logging to go to the Java
console like this:
```

```
static {
  com.bea.jcom.Log.logImmediately(3, System.err);
}
```

3 is the log level -- it means verbose. You can specify any file name as the second parameter.

To enable logging inside the WebLogic jCOM Moniker (jintmk.dll -- the 'glue'), use the registry editor to create a value under the following key `HKEY_LOCAL_MACHINE\SOFTWARE\Linar\JintMkr`. The value should be called `logFile` and should be set to a file name, such as `c:\temp\jcommk.log`.

Please send us both of the above logs if you run into problems.