WebSphere Application Server Enterprise Services

ActiveX to Java support

Notices:

[page of standard boilerplate and edition information]

Contents

ActiveX to EJB bridge concept articles

This part contains concept topics about the ActiveX to EJB bridge provided by WebSphere Application Server 4.0 enterprise services. These topics are intended to provide background information that you should understand to be able to complete tasks to enable and use the ActiveX to EJB bridge.

An overview of the ActiveX to EJB bridge

WebSphere Application Server provides an ActiveX to EJB bridge that enables ActiveX programs to access WebSphere Enterprise JavaBeans through a set of ActiveX automation objects. The bridge accomplishes this by loading the JVM into any ActiveX automation container such as Visual Basic, VBScript, and Active Server Pages. The ActiveX to EJB bridge scenario is shown in the following figure:

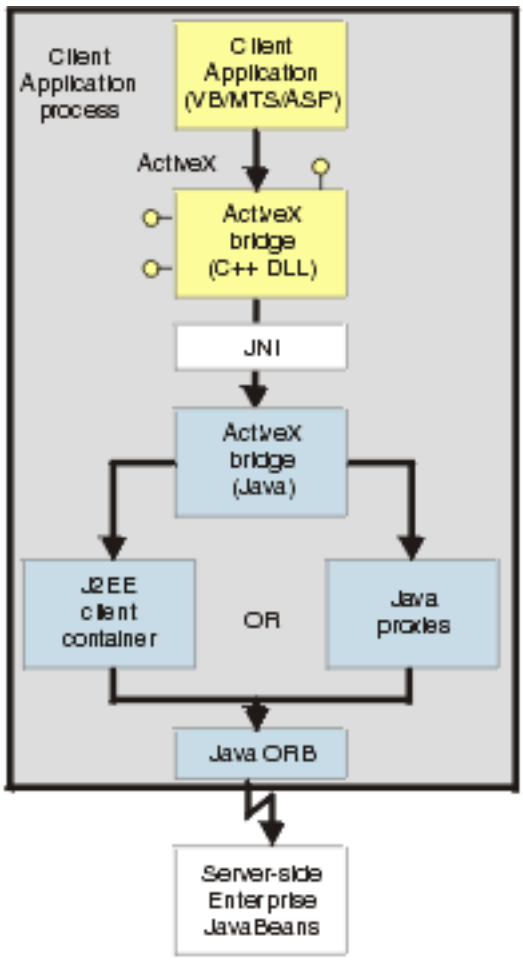

Figure 1 of 1. The ActiveX to EJB bridge scenario

There are two main environments that the ActiveX to EJB bridge runs in:

1. Client applications, such as Visual Basic and VBScript, are programs that a user would start from the command line, desktop icon, or start menu shortcut.
2. Client services, such as Active Server Pages, are programs that are started by some automated means like the Services control panel applet.

The ActiveX to EJB bridge uses the Java Native Interface (JNI) architecture to programmatically access the JVM. Therefore the JVM exists in the same process space as the ActiveX application (Visual Basic, VBScript, or ASP) and remains attached to the process until that process terminates. To create a JVM, an ActiveX client program calls the XJBInit() method of the XJB.JClassFactory object. For more information about creating a JVM for an ActiveX program, see “ActiveX to EJB bridge, initializing the JVM” (on page 3)

After an ActiveX client program has initialized the JVM, it calls several methods to create a

proxy object for the Java class. When accessing a Java class or object, the real Java object exists in the JVM; the automation container contains the proxy for that Java object. The ActiveX program can use the proxy object to access the Java class, object fields, and methods. For more information about using Java proxy objects, see “ActiveX to EJB bridge, using Java proxy objects” (on page 3) For more information about calling methods and access fields, see “ActiveX to EJB bridge, calling Java methods” (on page 4)and “ActiveX to EJB bridge, accessing Java fields” (on page 6)

The client program performs primitive data type conversion through the COM IDispatch interface (use of the IUnknown interface is not directly supported). Primitive data types are automatically converted between native Automation types and Java types. All other types are handled automatically by the Proxy Objects For more information about data type conversion, see “ActiveX to EJB bridge, converting data types” (on page 6)

Any exceptions thrown in Java are encapsulated and re-thrown as a COM error, from which the ActiveX program can determine the actual Java exceptions. For more information about handling exceptions, see “ActiveX to EJB bridge, handling errors” (on page 11)

The ActiveX to EJB bridge supports both free-threaded and apartment-threaded access and implements the Free Threaded Marshaler to work in a hybrid environment such as Active Server Pages. For more information about the support for threading, see “ActiveX to EJB bridge, using threading” (on page 12)

ActiveX to EJB bridge, initializing the JVM

For an ActiveX client program (Visual Basic, VBScript, or ASP) to access Java classes or objects, the first step that the program must do is to create a JVM within its process. To create a JVM, the ActiveX program calls the XJBInit() method of the XJB.JClassFactory object. When an XJB.JClassFactory object is created and the XJBInit() method called, the JVM is initialized and ready for use.

Note: To enable the XJB.JClassFactory to find the Java runtime DLLs when initializing, the JRE bin and bin\classic directories must be exist in the System path environment variable.

The XJBInit() method accepts only one parameter; an array of Strings. Each string in the array represents a command-line argument that for a Java program you would normally specify on the Java.exe command-line. This string interface is used to set the classpath, stack size, heap size, and debug settings. You can get a listing of these parameters by typing "java -?" from the command-line.

Note: If you set a parameter incorrectly, you receive a 0x6002 "Failed to initialize VM" error message.

Due to the current limitations of JNI, the JVM cannot be unloaded or reinitialized after it has been loaded. Therefore, after XJBInit() has been called once, subsequent calls have no effect other than to create a duplicate JClassFactory object for you to access. It is best to store your XJB.JClassFactory object globally and continue to reuse that object.

The following Visual Basic extract shows an example of initializing the JVM:

```
Dim oXJB as Object
set oXJB = CreateObject("XJB.JClassFactory")
Dim astrJavaInitProps(0) as String
astrJavaInitProps(0) = _
     "-Djava.class.path=.;c:\myjavaclasses;c:\myjars\myjar.jar"
oXJB.XJBInit(astrJavaInitProps)
```

ActiveX to EJB bridge, using Java proxy objects

After an ActiveX client program (Visual Basic, VBScript, or ASP) has initialized the XJB.JClassFactory (and thereby the JVM), it can access Java classes and initialize Java objects. To do this, the client program uses the XJB.JClassFactory FindClass() and NewInstance() methods.

In Java there are two ways to access Java classes: direct invocation through the Java compiler, and through Java's Reflection interface. Because the ActiveX to Java bridge needs no compilation and is a completely runtime interface to Java, it depends on the latter Reflection interface to access its classes, objects, methods, and fields. The XJB.JClassFactory FindClass() and NewInstance() methods behave very similarly to the Java Class.forName() and the Method.invoke() and Field.invoke() methods.

XJB.JClassFactory.FindClass() takes the fully-qualified class name as its only parameter and returns a Proxy Object (JClassProxy). You can use the returned Proxy object like a normal Java Class object, from it you can call static methods and access static fields. You can also create a Class Instance (or object) from it, as described below. For example, the following Visual Basic code extract returns a Proxy object for the Java class java.lang.Integer:

```
...
Dim clsMyString as Object
Set clsMyString = oXJB.FindClass("java.lang.Integer")
```

After the proxy has been created, you can access its static information directly. For example, the following code extract can be used to convert a decimal integer to its hexadecimal representation.

```
...
Dim strHexValue as String
strHexValue = clsMyString.toHexString(CLng(255))
```

The equivalent Java syntax is: `static String toHexString(int i)`. Because ints in Java are really 32-bits (which translate to Long in VB), the CLng() function is used to convert the value from the default int to a long. (For more detailed information about why this is important, see “ActiveX to EJB bridge, calling Java methods” (on page 4). Also, even though the toHexString() function returns a java.lang.String, the code extract does not return an Object Proxy. Instead, the returned java.lang.String is automatically converted to a native Visual Basic String.

To create an object from a class, you use the JClassFactory.NewInstance() method. This creates an Object Instance and takes whatever parameters your Class's constructor needs. When the object has been created, you then have access to all of it's public instance methods and fields. For example, the following Visual Basic code extract can be used to create an instance of a java.lang.Integer:

```
...
Dim oMyInteger as Object
set oMyInteger = oXJB.NewInstance(CLng(255))
Dim strMyInteger as String
strMyInteger = oMyInteger.toString
```

ActiveX to EJB bridge, calling Java methods

In the ActiveX to EJB bridge, methods are called using the native language's method invocation syntax. This topic provides some conceptual information about calling Java methods, based on the following important differences between Java invocation and ActiveX Automation invocation:

- Unlike Java, ActiveX does not support method (and constructor) polymorphism; that is, you cannot have two methods in the same class with the same name.
- Java is case-sensitive, but ActiveX Automation is not case-sensitive.

You should take care when invoking Java methods via ActiveX Automation. If you use the

wrong case on a method call or use the wrong parameter type, you get an Automation Error 438 "Object doesn't support this property or method" being thrown.

To be able to compensate for Java's polymorphic behavior, you need to give the exact parameter types to the method call. The parameter types are what determine the correct method to invoke. For a listing of correct types to use, see “ActiveX to EJB bridge, converting data types” (on page 6)

For example, the following Visual Basic code would fail if CLng() was not present or toHexString was incorrectly typed as ToHexString:

```
...
Dim strHexValue as String
strHexValue = clsMyString.toHexString(CLng(255))
```

Sometimes it is difficult to force some development environments to leave the case of your method calls unchanged. For example, in Visual Basic if you wanted to call a method `close()` (uncapitalized), Visual Basic would try to capitalize it `"Close()"`. In Visual Basic, the only way to effectively get around this behavior is to use the CallByName() method. For example:

```
o.Close(123)                                    'Incorrect...
CallByName(o, "close", vbMethod, 123)           'Correct...
```

or in VBScript, use the Eval function:

```
o.Close(123)                                     'Incorrect...
Eval("o.Close(123)")                             'Correct...
```

The return value of a function is always converted dynamically to the correct type. However, you must take care to use the **set** keyword in Visual Basic. If you expect a non-primitive data type to be returned, you must use **set**. (If you expect a primitive data type to be returned, you do not need to use **set**.) For example:

```
Set oMyObject = o.getObject
iMyInt = o.getInt
```

In some cases, you may not know the type of object you will get back from a method call, because wrapper classes are converted automatically to primitives (for example, java.lang.Integer returns an ActiveX Automation Long). In such cases, you may need to use your language's built-in exception handling techniques to try to coerce the returned type (for example, On Error and Err.Number in Visual Basic).

Methods with Character Arguments

Because ActiveX automation does not natively support character types supported by Java, the ActiveX to EJB bridge uses Strings (byte or VT_I1 do not work, because Characters have multiple bytes in Java). If you try to call a method that takes a char or java.lang.Character type you must use the JMethodArgs argument container to pass character values to methods or constructors. For more information about how this is used, see “Methods with "Object" Type as Argument and Abstract Arguments” (on page 5)

Methods with "Object" Type as Argument and Abstract Arguments

Because of the polymorphic nature of Java, the ActiveX to Java bridge uses direct argument type mapping to find a method. This works well in most cases, but sometimes methods are declared with a Parent or Abstract Class as an argument type (for example, java.lang.Object). You need the ability to send an object of arbitrary type to a method. To accomplish this, you must use the XJB.JMethodArgs object to coerce your parameters to match the parameters on your method. You can get a JMethodArgs instance by using the JClassFactory.GetArgsContainer() method.

The JMethodArgs object is a container for method parameters or arguments. It enables you to add parameters to it one-by-one and then you can send the JMethodArgs object to your method call. The JClassProxy and JObjectProxy objects recognize the JMethodArgs object

and attempt to find the correct method and let Java coerce your parameters appropriately.

For example, to add an element to a Hashtable object the method syntax is `Object put(Object key, Object value)`. In Visual Basic, the method usage would look like this:

```
Dim oMyHashtable as Object
Set oMyHashtable = _
    oXJB.NewInstance(oXJB.FindClass("java.utility.Hashtable"))
' This line will not work. The ActiveX to EJB bridge cannot find a method
' called "put" that has a short and String as a parameter:
oMyHashtable.put 100, "Dogs"
oMyHashtable.put 200, "Cats"
' You must use a XJB.JMethodArgs object instead:
Dim oMyHashtableArgs as Object
Set oMyHashtableArgs = oXJB.GetArgsContainer
oMyHashtableArgs.AddObject("java.lang.Object", 100)
oMyHashtableArgs.AddObject("java.lang.Object", "Dogs")
oMyHashtable.put oMyHashTableArgs
' Reuse the same JMethodArgs object by clearing it.
oMyHashtableArgs.Clear
oMyHashtableArgs.AddObject("java.lang.Object", 200)
oMyHashtableArgs.AddObject("java.lang.Object", "Cats")
oMyHashtable.put oMyHashTableArgs
```

ActiveX to EJB bridge, accessing Java fields

Using the ActiveX to EJB bridge, access to Java fields has the same case sensitivity issue that it has when invoking methods. Field names must use the same case as the Java field syntax.

Visual Basic has the same problem with unsolicitied case changing on fields as it does with methods. (For more information about this problem, see “ActiveX to EJB bridge, calling Java methods” (on page 4). You may need to use the CallByName() function to set a field in the same way that you would call a method in some cases. For Fields, you use VBLet for primitive types and VBSet for Objects. For example:

```
o.MyField = 123                          'Incorrect...
CallByName(o, "MyField", vbLet, 123)     'Correct...
```

or in VBScript:

```
o.MyField = 123                           'Incorrect...
Eval("o.myField = 123")                   'Correct...
```

ActiveX to EJB bridge, converting Java primitive data types

All primitive Java data types are automatically converted to native ActiveX Automation types. However, not all Automation data types are converted to Java types (for example, VT_DATE). Variant data types are used for data conversion. Variant data types are a requirement of any Automation interface, and are used automatically by Visual Basic and VBScript. The tables below provide details about how primitive data types are converted between Automation types and Java types.

Table: 1. ActiveX to Java primitive data type conversion

Visual Basic Type	Variant Type	Java Type	Notes
Byte	VT_I1	byte	Byte in Visual Basic is unsigned, but is signed in Java. See

Visual Basic Type	Variant Type	Java Type	Notes
			“Byte Helper Function” (on page 10)
Boolean	VT_BOOL	boolean	***
Integer	VT_I2	short	***
Long	VT_I4	int	***
Currency	VT_CY	long	See “Currency Helper Function” (on page 10)
Single	VT_R4	float	***
Double	VT_R8	double	***
String	VT_BSTR	java.lang.String	***
String	VT_BSTR	char	For information about how chars are handled, see “ActiveX to EJB bridge, calling Java

Visual Basic Type	Variant Type	Java Type	Notes
			methods ” (on page 4) and “ActiveX to EJB bridge, accessing Java fields ” (on page 6)
Date	VT_DATE	n/a	***

Table: 2. Java to ActiveX primitive data type conversion

Java Type	Variant Type	Visual Basic Type	Notes
• byte • java.lang.Byte	VT_I1	Byte	Byte in Visual Basic is unsigned, but is signed in Java. See “Byte Helper Function” (on page 10)
• boolean • java.lang.Boolean	VT_BOOL	Boolean	***
• short • java.lang.Short	VT_I2	Integer	***

Java Type	Variant Type	Visual Basic Type	Notes
• int • java.lang.Integer	VT_I4	Long	***
• long • java.lang.Long	VT_CY	Currency	***
• float • java.lang.Float	VT_R4	Single	***
• double • java.lang.Double	VT_R8	Double	***
• java.lang.String	VT_BSTR	String	***
• char • java.lang.Character	VT_BSTR	String	***
n/a	VT_DATE	Date	Not available. java.util.Date objects are represented as normal Object Proxy objects.

ActiveX to EJB bridge, helper methods for data type conversion

Generally, data type conversion between ActiveX (Visual Basic and VBScript) and Java occurs automatically, as described in “ActiveX to EJB bridge, converting data types” (on page 6). However, the following helper functions are provided for cases where automatic conversion is not possible:

- “Byte helper function” (on page 10)
- “Currency helper function” (on page 10)

Byte helper function

Because the Java Byte data type is signed (-127 through 128) and the Visual Basic Byte data type is unsigned (0 through 255), you need to convert unsigned Bytes to a Visual Basic Integers, which look like the Java signed byte. To do this, you can use the following helper function:

```
Private Function GetIntFromJavaByte(Byte jByte) as Integer
            GetIntFromJavaByte = (CInt(jByte) + 128) Mod 256 - 128
End Function
```

Currency helper function

Visual Basic 6.0 cannot properly handle 64-bit integers like Java can (as the Long data type). Therefore, Visual Basic uses the Currency type, which is intrinsically a 64-bit data type. The only side-effect of using the Currency type (the Variant type VT_CY) is that a decimal point is inserted into the type. To extract and manipulate the 64-bit Long value in Visual Basic, you need to use code like the following example. For more details on this technique for converting Currency data types, see Q189862, "HOWTO: Do 64-bit Arithmetic in VBA", on the “ Microsoft Knowledge Base”.

```
' Currency Helper Types
Private Type MungeCurr
    Value As Currency
End Type
Private Type Munge2Long
    LoValue As Long
    HiValue As Long
End Type
' Currency Helper Functions
Private Function CurrToText(ByVal Value As Currency) As String
    Dim Temp As String, L As Long
    Temp = Format$(Value, "#.0000")
    L = Len(Temp)
    Temp = Left$(Temp, L - 5) & Right$(Temp, 4)
    Do While Len(Temp) > 1 And Left$(Temp, 1) = "0"
        Temp = Mid$(Temp, 2)
    Loop
    Do While Len(Temp) > 2 And Left$(Temp, 2) = "-0"
        Temp = "-" & Mid$(Temp, 3)
    Loop
    CurrToText = Temp
End Function
Private Function TextToCurr(ByVal Value As String) As Currency
    Dim L As Long, Negative As Boolean
    Value = Trim$(Value)
    If Left$(Value, 1) = "-" Then
        Negative = True
        Value = Mid$(Value, 2)
    End If
    L = Len(Value)
    If L < 4 Then
        TextToCurr = CCur(IIf(Negative, "-0.", "0.") & _
                          Right$("0000" & Value, 4))
    Else
        TextToCurr = CCur(IIf(Negative, "-", "") & _
                          Left$(Value, L - 4) & "." & Right$(Value, 4))
    End If
End Function
' Java Long as Currency Usage Example
Dim LC As MungeCurr
Dim L2 As Munge2Long
' Assign a Currency Value (really a Java Long)
' to the MungeCurr type variable
LC.Value = cyTestIn
' Coerce the value to the Munge2Long type variable
LSet L2 = LC
' Perform some operation on the value, now that we
' have it available in two 32-bit chunks
L2.LoValue = L2.LoValue + 1
' Coerce the Munge value back into a currency value
LSet LC = L2
cyTestIn = LC.Value
```

ActiveX to EJB bridge, using Arrays

Arrays are very similar between Java and Automation Containers like Visual Basic and VBScript. Here are some important points to consider when passing arrays back and forth between Java and Automation:

- Java arrays cannot mix types. All arrays in Java contain a single type, so when passing arrays of Variants to a Java Array, you must make sure that all of the elements in the Variant array are of the same base type. For example, in Visual Basic:

```
...
Dim VariantArray(1) as Variant
VariantArray(0) = CLng(123)
VariantArray(1) = CDbl(123.4)
oMyJavaObject.foo(VariantArray) '     Illegal!
VariantArray(0) = CLng(123)
VariantArray(1) = CLng(1234)
oMyJavaObject.foo(VariantArray) '     This works
```

- Arrays of Primitive Types are converted using the rules defined in Primitive Data Type Conversion.
- Arrays of Java Objects are handled through the use of arrays of JObjectProxy objects.
- Arrays of JObjectProxy objects must be fully-initialized and of the correct associated Java type. When initializing an array in Visual Basic (for example, Dim oJavaObjects(1) as Object), you must set each object to a JObjectProxy before you send the array to Java. The bridge is unable to determine the type of null or empty Object values.
- When receiving an array from Java, the lower-bound will always be zero. Java only supports zero-based arrays.
- Nested or multi-dimensional arrays are treated as zero-based multi-dimensional arrays in Visual Basic and VBScript.
- Uninitialized arrays or Array Types are unsupported. When calling a Java method that takes an array of objects as a parameter, you must fully initialize the array of JObjectProxy objects.

ActiveX to EJB bridge, handling errors

All exceptions thrown in Java are encapsulated and re-thrown as a COM error through the ISupportErrorInfo interface and the EXCEPINFO structure of IDispatch::Invoke(); the Err object in Visual Basic and VBScript. Because there are no error numbers associated with Java exceptions, whenever a Java exception is thrown, the entire stack trace is stored in the error description text and the error number assigned is 0x6003.

In Visual Basic or VBScript, you need to use the Err.Number and Err.Description fields to determine the actual Java error. Non-Java errors are thrown as you would expect via the IDispatch interface; for example, if a method cannot be found, then error 438 "Object doesn't support this property or method" is thrown.

Error number	Description
0x6001	JNI error
0x6002	Initialization error
0x6003	Java exception. Error description is the Java Stack Trace.

Error number	Description
0x6FFF	General Internal Failure

ActiveX to EJB bridge, using threading

The ActiveX to EJB bridge supports both free-threaded and apartment-threaded access and implements the Free Threaded Marshaler to work in a hybrid environment such as Active Server Pages. Each thread that is created in the ActiveX process will be mirrored in the Java environment when the thread communicates through the ActiveX to EJB bridge. In addition, once all references to Java objects (there are no JObjectProxy or JClassProxy objects) in a ActiveX thread, the ActiveX to EJB bridge will detach the thread from the JVM. Therefore, you must be careful that any Java code that you access from a multi-threaded Windows application is thread-safe. Visual Basic and VBScript applications are both essentially single-threaded. Therefore, Visual Basic and VBScript applications don't really need to worry about threading issues in the Java programs they access. Active Server Pages and multi-threaded C and C++ programs do need to worry.

Consider the following scenario:

1. A multi-threaded Windows Automation Container (our ActiveX Process) starts. It exists on Thread A.
2. The ActiveX Process initializes the ActiveX to EJB bridge, which starts the JVM. The JVM attaches to the same thread and internally calls it Thread1.
3. The ActiveX Process starts two threads: B and C.
4. Thread B in the ActiveX Process uses the ActiveX to EJB bridge to access an object that was created in Thread A. The JVM attaches to thread B and calls it Thread 2.
5. Thread C in the ActiveX Process never talks to the JVM, so the VM never needs to attach to it. This is a case where the JVM doesn't have a one-to-one relationship between ActiveX threads and Java threads.
6. Thread B later releases all of the JObjectProxy and JClassProxy objects that it used. Java's Thread 2 is detached.
7. Thread B again uses the ActiveX to EJB bridge to access an object that was created in Thread A. The JVM again attaches to thread and calls it Thread 3.

ActiveX Process	JVM Access by ActiveX Process
Thread A - Created in 1	Thread 1 - Attached in 2
Thread B - Created in 4	Thread 2 - Attached in 4, detached in 6 Thread 3 - Attached in 7
Thread C - Created in 4	***

Threads and Active Server Pages

Active Server Pages in Microsoft's Internet Information Server is a multi-threaded environment. When you create the XJB.JClassFactory object, you can store it in the Application collection as an Application-global object. This allows all threads within your ASP environment to access the same ActiveX to EJB bridge object. Active Server Pages by default creates 10 Apartment Threads per ASP process per CPU. This means that when your ActiveX to EJB bridge object is initialized it can be called by any of the 10 threads, not just the thread who created it.

If you need to simulate single-apartment behavior, you can create a Single-Apartment Threaded ActiveX DLL in Visual Basic and encapsulate the ActiveX to EJB bridge object there. This guarantees that all access to the JVM uses the same thread. You need to use the <OBJECT> tag to assign the XJB.JClassFactory to an Application object and must be aware of the consequences of introducing single-threaded behavior to a Web application.

The Microsoft KnowlegeBase has several articles about ASP and threads, including:

- Q243543 INFO: Do Not Store STA Objects in Session or Application
- Q243544 INFO: Component Threading Model Summary Under Active Server Pages
- Q243548 INFO: Design Guidelines for VB Components Under ASP

ActiveX to EJB bridge, good programming guidelines

In general, the best way to access Java components is to use the Java language. Because of this, you are recommended to do as much programming as possible in Java and to use a small simple interface between your COM Automation container (for example, Visual Basic) and Java. This avoids any overhead and performance problems that can occur when moving across the interface between ActiveX and Java.

- “Visual Basic guidelines” (on page 13)
- “Active Server Pages guidelines” (on page 14)
- “J2EE guidelines” (on page 15)

Visual Basic guidelines

The following guidelines are intended to help optimize use of the ActiveX to EJB bridge with Visual Basic:

- Launch Visual Basic through launchClientXJB.bat

  If you want to run your Visual Basic application through the Visual Basic debugger, you must be running the Visual Basic IDE (Integrated Development Environment) within the ActiveX to EJB bridge environment. After your Visual Basic project is created, you can simply launch it from a command-line; for example, `launchClientXJB MyAppliation.vbp`. Alternatively, you can launch Visual Basic alone in the ActiveX to EJB environment by changing the Visual Basic shortcut on the Windows Start menu so that launchClientXJB.bat precedes the call to VB6.EXE.

- Exit the Visual Basic IDE before debugging programs.

  Because the JVM attaches to the running process, you must exit out of the Visual Basic editor before debugging your program. If you run then exit your program within the Visual Basic IDE, the JVM continues to run and you reattach the same JVM when XJBInit() is called by the debugger. This causes problems if you are trying to update XJBInit() arguments (for example, classpath), because the changes are not be applied until you restart Visual Basic.

- Store XJB.JClassFactory Globally

  Because the JVM cannot be unloaded or reinitialized, you should cache the resulting

XJB.JClassFactory object as a global variable. The overhead of treating it as a global variable or passing a single reference around is much less than recreating a new XJB.JClassFactory object and calling XJBInit() more than once.

CScript and Windows Scripting Host (WSH)

The following guidelines are intended to help optimize use of the ActiveX to EJB bridge with CScript and Windows Scripting Host (WSH):

- Launch in ActiveX to EJB Environment

  To run VBScript files in .VBS files, you launch them in the ActiveX to EJB bridge environment. There are two common ways to launch your script:

- launchClientXJB MyScript.vbs
- launchClientXJB cscript MyScript.vbs

Active Server Pages guidelines

The following guidelines are intended to help optimize use of the ActiveX to EJB bridge with Active Server Pages:

- Use the ActiveX to EJB Helper Functions from Active Server Pages

  Because Active Server Pages typically uses VBScript, we have included some helper functions that you can use in any VBScript environment with minor changes. For more information about these helper functions, see “Helper functions for data type conversion” (on page 10). To be able to run outside of the ASP environment, the only change you need to make is to remove or change all references to the Server, Request, Response, Application and Session objects; for example, change `Server.CreateObject` to `CreateObject`.

- Set JRE Path Globally in System

  The XJB.JClassFactory must be able to find the Java runtime DLLs when initializing. In Internet Information Server you cannot specify a path for its processes independently; you must set the process paths in the system PATH variable. This means that you can only have a single JVM version available on a machine using ASP. Also, remember that after you change the System PATH variable you must reboot the Internet Information Server machine so that Internet Information Server can see the change.

- Set the system TEMP environment variable

  If the system TEMP environment variable is not set, Internet Information Server stores all temporary files in the WINNT directory. This is usually not a desired behavior.

- Use High Isolation or an Isolated Process

  When using the ActiveX to Java bridge with Active Server Pages, you are recommended to create your Web application in its own process. This is because you can only load one JVM in a single process and if you want to have more than one application running with different JVM environment options (for example, different classpaths), then you need to have separate processes.

- Use the Application Unload Button

  When debugging your application, use the **`Unload`** button when viewing your ASP Application properties in the Internet Information Server administration console to unload the process from memory (and thereby unload the JVM).

- Run One Process per Application

  In your ASP environment, only use one ASP application per J2EE Application or JVM

environment. If you need separate classpaths or JVM settings you need to have separate ASP Applications (Virtual Directories with High Isolation or Isolated Process).

- Store XJB.JClassFactory in Application Scope

  Because of the one-to-one relationship required between a JVM and a process, and because the JVM can never be detached or shutdown from a process independently, you should cache the XJB.JClassFactory object at Application scope and call XJBInit() once only.

  Because the ActiveX to EJB bridge employs a free-threaded marshaler, you should be able to take advantage of the multi-threaded nature of Internet Information Server and ASP. If you choose to reinitialize the XJB.JClassFactory object at Page scope (local variables), then the XJBInit() method can only initialize your local XJB.JClassFactory variable. Therefore, it is more efficient to do the XJBInit() once instead of many times.

- Use VBScript Conversion Functions

  Because VBScript only supports variant data types, you must use the CStr(), CByte(), CBool(), CCur(), CInt(), Clng(), CSng() and CDbl() functions to tell the activeX to EJB bridge what data type you really mean to use; for example `oMyObject.Foo(CDbl(1.234))`.

J2EE guidelines

The following guidelines are intended to help optimize use of the ActiveX to EJB bridge with J2EE:

- Store client container object globally

  Because you can only have one JVM per process, and a single J2EE client container (com.ibm.websphere.client.applicationclient.launchClient) per JVM, you should initialize your J2EE client container once only and to reuse it. For ASP, store the J2EE client container in an Application level variable and initialize it once only (either on the Application_OnStart() event in global.asa or by checking to see if it IsEmpty()).

  A side effect to storing the client container object globally is that you are unable to change the client container parameters without destroying the object and creating a new one. These parameters include the EAR file, BootstrapHost, classpath, and so on. If you are running a Visual Basic application and wish to change the client container parameters, you must end the application and restart it. If you are running an Active Server Pages application, you must first unload the application from Internet Information Server (see ""Use the Application Unload Button" under Active Server Pages guidelines" (on page 14)). Then load the Active Server Pages application with the different client container parameters. The first time the Active Server Pages application is loaded, the parameters will be set. Since the client container is stored on the Internet Information Server, the parameters are shared by all browser clients using the Active Server Pages application. This is normal Active Server Pages behavior, but it may be confusing when you try to run to different WebSphere Application Servers using the same Active Server Pages application. You cannot do that.

- Reuse custom temp directory for EAR file extraction

  By default, the client container is launched and extracts the application's EAR file to your temp directory and then sets up the thread's ClassLoader to use the extracted EAR file directory and JAR files included in the client Jar's manifest. This process is time consuming and because of some limitations with JVM Shutdown via JNI and file locking, these files are never cleaned up.

  Specifically, each time the client container's launch() method is called, it extracts the EAR

file to a random directory name in your temporary directory on your hard drive. The current Java thread's class Loader is then changed to point to this extracted directory which in turn locks the files within. In a normal J2EE Java client, these files will be automatically cleaned up after the application exits. This occurs when the client container's shutdown hook is called (which never happens in the ActiveX to EJB bridge) which leaves the temporary directory there.

To avoid these problems, you can specify a directory to extract the EAR file into by setting the com.ibm.websphere.client.applicationclient.archivedir Java system property before calling the client container's launch() method. If the directory does not exist or is empty, the EAR file is extracted like normal. If the EAR file was previously extracted, the directly is reused. This feature is particularly important for server processes (for example, ASP), which can stop and restart, potentially calling launchClient() several times.

If you need to update your EAR file, you must delete the temporary directory first. Then the next time the client container object is created, it extracts the new EAR file to the temporary directory. If you do not delete the temporary directory or change the system property value to point to a different temporary directory, the client container reuses the currently extracted EAR file, and your changed EAR file is not used.

Note: When specifying the com.ibm.websphere.client.applicationclient.archivedir property, make sure that the directory you specify *is unique* for each EAR file you use. For example, do not point MyEar1.ear and MyEar2.ear to the same directory.

If you choose not to use this system property, you need to regularly go to your Windows temp directory and delete the WSTMP* subdirectories. Over a relatively short period of time, these subdirectories can waste a very significant amount of space on the hard drive.

ActiveX to EJB bridge task articles

This part contains task topics about the ActiveX to EJB bridge provided by WebSphere Application Server 4.0 enterprise services. These topics are intended to describe how you should complete tasks to enable and use the ActiveX to EJB bridge.

- “ActiveX to EJB bridge concept articles” (on page 1)
- “ActiveX to EJB bridge example articles” (on page 19)
- “ActiveX to EJB bridge reference articles” (on page 28)

Developing an ActiveX program to use the ActiveX to EJB bridge

This topic provides an outline of developing an ActiveX program, such as Visual Basic, VBScript, and Active Server Pages, to use the WebSphere ActiveX to EJB bridge to access Enterprise JavaBeans. For more information about the concepts behind the steps involved, see the related topics.

This topic assumes that you are familiar with ActiveX programming. You should also consider the information given in “ActiveX to EJB bridge, good programming guidelines” (on page 13).

To use the ActiveX to EJB bridge to access a Java class, develop your ActiveX program to complete the following steps:

1. Create an instance of the XJB.JClassFactory object.
2. Create a JVM within the ActiveX program's process, by calling the XJBInit() method of the XJB.JClassFactory object.
   For more information about creating a JVM for an ActiveX program, see “ActiveX to EJB bridge, initializing the JVM” (on page 3).

   After the ActiveX program has created an XJB.JClassFactory object and called the XJBInit() method, the JVM is initialized and ready for use.
3. Create a proxy object for the Java class, by using the XJB.JClassFactory FindClass() and NewInstance() methods.
   For more information about creating and using Java proxy objects, see “ActiveX to EJB bridge, using Java proxy objects” (on page 3).

   The ActiveX program can use the proxy object to access the Java class, object fields, and methods.
4. Call methods on the Java class, using the Java method invocation syntax, and access Java fields as required.
   For more information about calling methods and access fields, see “ActiveX to EJB bridge, calling Java methods” (on page 4) and “ActiveX to EJB bridge, accessing Java fields” (on page 6).
5. Make use of the helper functions provided for conversion between Java Byte and Visual Basic Byte data types, and between Visual Basic Currency types and Java 64-bit data types, cases where automatic conversion is not possible.
6. Implement methods to handle any errors returned from the Java class. In Visual Basic or VBScript, use the Err.Number and Err.Description fields to determine the actual Java error.

Starting an ActiveX application to use the ActiveX to EJB bridge

To run an ActiveX client application that is to use the ActiveX to EJB bridge, you must perform some initial configuration to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its XJB.JAR file and the Java runtime. This sets up the environment within which the ActiveX client application can run.

To perform the required configuration, complete one or more of the following subtasks:

- “Starting an ActiveX application, configuring service programs” (on page 18)
- “Starting an ActiveX application, configuring non-service programs” (on page 19)

Starting an ActiveX application, configuring service programs

To run an ActiveX service program such as an Active Server Page that is to use the ActiveX to EJB bridge, you must perform some initial configuration to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its XJB.JAR file and the Java runtime. This sets up the environment within which the ActiveX service program can run.

The XJB.JClassFactory must be able to find the Java runtime DLLs when initializing. In a service program such as Internet Information Server you cannot specify a path for its processes independently; you must set the process paths in the system PATH variable. This means that you can only have a single JVM version available on a machine using ASP. Also, after you change the System PATH variable you must reboot the Internet Information Server machine so that Internet Information Server can see the change.

To add the JRE directories to your System path, complete one of the following subtasks:

1. On Windows 2000, complete the following substeps:
    1. Open the Control Panel, then double-click the System icon.
    2. Click the Advanced tab on the System Properties window.
    3. Click **Environment Variables**.
    4. In the System Variables window, edit the **Path** variable.
    5. Add the following to the beginning of the path displayed in the Variable Value input box:

        ```
        x:\WebSphere\AppClient\Java\jre\bin;C:\WebSphere\AppClient\Java\jre\bin\classic;
        ```

        Where `x:\WebSphere\AppClient` is the directory in which you installed the WebSphere Java client
    6. To apply the changes **OK** in the Edit System Variable window.
    7. Click **OK** in the Environment Variables window.
    8. Click **OK** in the System Properties window.
    9. Restart Windows 2000.
2. On Windows NT, complete the following substeps:
    1. Open the Control Panel, then double-click the System icon.
    2. Click the Environment tab on the System Properties window.
    3. In the System Variables window, edit the **Path** variable.
    4. Add the following to the beginning of the path displayed in the Value input box:

```
x:\WebSphere\AppClient\Java\jre\bin;C:\WebSphere\AppClient\Java\jre\bin\classic;
```

Where `x:\WebSphere\AppClient` is the directory in which you installed the WebSphere Java client

5. To apply the changes **Set**.
6. Click **OK**.
7. Restart Windows NT.

Starting an ActiveX application, configuring non-service programs

To run an ActiveX program initiated from an icon or command-line (a non-service program) that is to use the ActiveX to EJB bridge, you must perform some initial configuration to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its XJB.JAR file and the Java runtime. This uses a batch file to set up the environment within which the ActiveX program can run.

To perform the required configuration, complete the following steps:

1. **Optional:** Edit the setupCmdLineXJB.bat file to specify appropriate values for the environment variables required by the ActiveX to EJB bridge. For more information about these environment variables, see “ActiveX to EJB bridge, environment and configuration” (on page 29)
   For more information about creating a JVM for an ActiveX program, see “ActiveX to EJB bridge, initializing the JVM” (on page 3).

   After the ActiveX program has created an XJB.JClassFactory object and called the XJBInit() method, the JVM is initialized and ready for use.

2. Start the ActiveX client application by using one of the following methods:
   - Use the launchClientXJB.bat file to start the application; for example:

     ```
     launchClientXJB MyApplication.exe  parm1 parm2
     ```

     or

     ```
     launchClientXJB MyApplication.vbp
     ```

   - Use the setupCmdLineXJB.bat file to create an environment in which the application can be run, then start the application from within that environment.

ActiveX to EJB bridge example articles

This part contains example topics about the ActiveX to EJB bridge provided by WebSphere Application Server 4.0 enterprise services. These topics provide an overview of, and links to, the samples provided with WebSphere Application Server enterprise services.

ActiveX to EJB support, usage samples

WebSphere Application Server provides a set of samples to illustrate how ActiveX clients (Visual Basic, VBScript, and Active Server Pages) can use the ActiveX to EJB bridge to access WebSphere JavaBeans. For information about how to set up and run the samples provided with WebSphere Application Server, see the related samples information.

The following topics provide information about the generic programming technqiues used in the samples, and demonstrate some possible uses of the ActiveX to EJB bridge. The information is based on the code provided in the ActiveX to EJB Client samples and is fairly typical for accessing Enterprise JavaBeans from a Visual Basic or Active Server Pages environment. The code extracts can be useful for creating your own code and to create a small library to aid your own program development.

Examples: Visual Basic helper functions

The following Visual Basic code extracts can be used to help create various types of ActiveX to EJB bridge objects and Enterprise JaveBeans. This code is taken from the Visual Basic samples included with the ActiveX to EJB Client provided with WebSphere Application Server. For information about the samples, see the related samples information.

For equivalent Active Server Page example code, see "Examples: ASP helper functions" (on page 22)

Initializing the ActiveX to EJB bridge and JVM

The following code extract initializes the ActiveX to EJB bridge and the JVM:

```
 '-------------------------------------------------------------
 ' Initialize the ActiveX to EJB bridge and the JVM
 '-------------------------------------------------------------
 Private Function XJBInit(strWAS_HOME as String, strJAVA_HOME as String,
strNAMING_FACTORY as string)
     Dim oXJB
     ' Create an XJB.JClassFactory object.  This object
     ' is responsible for finding classes and instantiating
     ' objects.
     Set oXJB = CreateObject("XJB.JClassFactory")
     ' If you intend to debug using a JPDA debugger, you will need tools.jar
     ' strClassPath = strClassPath & ";" & strJAVA_HOME & "\lib\tools.jar"
     ' The ActiveX to EJB bridge properties are in the form of an array of strings.
     ' Each property that you would normally pass on a command-line
     ' is passed as an individual string in the array.
     ' Here, we are adding the classpath (put together above), extension directories
     ' which are required for J2EE access and the java.naming.factory.initial property
     ' which is always the same.
     Dim astrJavaProps(2)
     astrJavaProps(0) = "-Djava.ext.dirs=" & strWAS_HOME & "\classes;" & _
                        strWAS_HOME & "\lib\ext;" & strWAS_HOME & "\lib;" & _
                        strJAVA_HOME & "\jre\lib\ext;" & strWAS_HOME & "\properties;"
     astrJavaProps(1) = "-Dws.ext.dirs=" & strWAS_HOME & "\classes;" & _
                        strWAS_HOME & "\lib\ext;" & strWAS_HOME & "\lib;" & _
                        strJAVA_HOME & "\jre\lib\ext;" & strWAS_HOME & "\properties;"
     astrJavaProps(2) = "-Djava.naming.factory.initial=" & strNAMING_FACTORY
     ' If you want to add more to the class path, you can do it here
     ' astrJavaProps(3) = "-Djava.class.path=" & strClassPath
     ' If you wish to debug using a JPDA debugger (you can't use JDB because there is
     ' no console to get the agent password), use parameters similar to the following
     '    astrJavaProps(3) = "-Djava.compiler=NONE"
     '    astrJavaProps(4) = "-Xnoagent"
     '    astrJavaProps(5) = "-Xdebug"
     '    astrJavaProps(6) =
"-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000"
     ' Initialize the JVM using our properties that we just setup.
```

```
    ' The JVM will run in Visual Basic's process space (the IDE if running
    ' in the Visual Basic interactive debugger)
    ' The JVM remains in this process space until the process ends.
    ' Running XJBInit a second time will have no effect.  It continues
    ' to use the classpath and properties of the JVM that were initially used.
    oXJB.XJBInit astrJavaProps
    Set XJBInit = oXJB
End Function
```

Creating a J2EE client container

The following code extract creates a J2EE client container:

```
'-------------------------------------------------------------
' XJBGetJ2EEClientContainer - Get the J2EE Client Container Object
' Input:
'   oXJB             - Initialized the ActiveX to EJB bridge object
'   strEARFile       - Location where EJB stub jars and J2EE App Client Jar is stored
'   strBootstrapHost - Name of the server where the EJB app is running
'   strBootstrapPort - Port where the EJB server is running
'   strAppClientJar  - Name of the jar file in the EAR file that contains the Java
'                      Application Client.  We need this only when we have multiple
'                      client jars in a single ear.  Even though we don't run the
'                      Java client, the J2EE Client Container will use the manifest
'                      inside to reference the EJB JAR file(s) that we will need
'                      to reference.
'   strEARTempDir    - Directory to extract the EAR file to.
'-------------------------------------------------------------
Function XJBGetJ2EEClientContainer(oXJB As Object, strEARFile As String, _
                                   strBootstrapHost As String, strBootstrapPort As
String, _
                                   strAppClientJar As String, strEARTempDir as String)
    On Error GoTo ExHandler
    Dim oProps As Object
    Dim oCC As Object
    Dim astrAppParams(0) As String
    ' Initialize the J2EE Client Container
    ' Before initializing the Client Container, set the
    ' System Property that tells it what directory to
    ' extract the EAR file to.  If we don't do this, then
    ' a new directory is created each time we launch the
    ' Client Container and is not cleaned-up.
    if Trim(strEARTempDir) <> "" Then
        XJBPutProperty oXJB, oXJB.FindClass("java.lang.System").getProperties, _
                       "com.ibm.websphere.client.applicationclient.archivedir", _
                       strEARTempDir
    End If
    ' - Here we are creating a launchClient object which is supplied
    '   in appclient.jar supplied with WebSphere (Client or Server install).
    ' - Notice that we are eliminating the need to store the Class Proxy object
    '   returned by FindClass.  We are using as a temporary variable and immediately
    '   creating a new instance (Object Proxy) and throwing the Class Proxy away.
    Set oCC =
oXJB.NewInstance(oXJB.FindClass("com.ibm.websphere.client.applicationclient.launchClient"))
    ' Create a java.util.Properties object to use in our launchClient() J2EE API
    Set oProps = oXJB.NewInstance(oXJB.FindClass("java.util.Properties"))
    ' Add properties to the Properties object.
    ' - initonly=true is required to avoid running the Java Client that is stored
    '   in the EAR file.
    ' - BootstrapHost = "xxxx" : The name of the WebSphere server where your
    '                            Enterprise JavaBeans are stored.
    ' - BootstrapPort = nnnn   : The port of the listener on the WebSphere server.
    ' - jar = "xxxx"           : The name of the jar within the ear file that
    '                            contains the Java Client (if you have more than one
    '                            Java Client Jar in the same EAR file.
    '                           (the path is relative to the root of the EAR file)
    XJBPutProperty oXJB, oProps, "initonly", "true"
    If Trim(strBootstrapHost) <> "" Then
        XJBPutProperty oXJB, oProps, "BootstrapHost", strBootstrapHost
    End If
    If IsNumeric(strBootstrapPort) Then
        XJBPutProperty oXJB, oProps, "BootstrapPort", strBootstrapPort
    End If
    If Trim(strAppClientJar) <> "" Then
        XJBPutProperty oXJB, oProps, "jar", strAppClientJar
    End If
    ' If we want to turn tracing on, we can do something like this:
    'XJBPutProperty oXJB, oProps, "trace",
"com.ibm.ws.client.applicationclient.ApplicationClientMetaData=debug=enabled"
    'XJBPutProperty oXJB, oProps, "tracefile", "c:\xjb.log"
    DoEvents
    ' Launch the client container.
    ' In the java world, it would start the Java Application within the
    ' container.  Here, however, we ARE the application, so we avoid
    ' starting it by setting the property initonly=true (above).
    ' The Client Container overrides the Java Namespace so that we can load
    ' the appropriate EJB class files from within our supplied ear file.
    oCC.launch strEARFile, oProps, astrAppParams
    Set XJBGetJ2EEClientContainer = oCC
    Set oCC = Nothing
    Set oProps = Nothing
    Exit Function
ExHandler:
    MsgBox Err.Description
End Function
```

Getting an Enterprise JavaBean home object

The following code extract gets an Enterprise JavaBean's home object:

```
'-------------------------------------------------------------
' XJBGetEJBHome - Get the Home EJB Object
' Input:
```

```
'   oXJB             - Initialized ActiveX to EJB bridge object
'   oInitialContext  - Java InitialContext object
'   strJNDIName      - JNDI Name of the EJB you are going to
'                      get the Home of.
'   strHomeClassName - Class Name of the Home object
'-------------------------------------------------------------
Private Function XJBGetEJBHome(oXJB As Object, oInitialContext As Object, _
                               strJNDIName As String, strHomeClassName As String)
    On Error GoTo ExHandler
    ' Create an XJB.JMethodArgs object.
    ' The prototype for javax.rmi.PortableRemoteObject.narrow() is:
    '   javax.rmi.PortableRemoteObject.narrow(Object, Class)
    ' In this case we want to use our Java Object and the Java Class that this
    ' Java Object needs to be "narrowed" to.
    ' Because the ActiveX to EJB bridge cannot coerce data types from String to Object
       ' (or any other type of implicit coersion), we must manually tell the
       ' ActiveX to EJB bridge how to do this (map a String to an Object).
    ' The JMethodsArgs object is a container for Method Arguments.  Here we
    ' are adding two Objects, which are both Strings (Key and Value).
    Dim oArgs As Object
    Set oArgs = oXJB.GetArgsContainer
    ' Lookup an object's home
    ' - This is typical J2EE Java code.  We use the JNDI Name that was
    '   passed-in to lookup our EJB
    Dim oJavaObj As Object
    Set oJavaObj = oInitialContext.lookup(strJNDIName)
    ' Narrow it
    ' - This is more EJB/J2EE code.  Here we use or JMethodArgs object
    '   to coerce our EJB object to a specific class.   In this case
    '   our EJB Home class.
    ' - Notice that we are calling the static narrow() method on a Class Proxy
    '   object that was created from FindClass().
    Dim clsPortRemObj As Object
    Dim oHome As Object
    oArgs.AddObject "java.lang.Object", oJavaObj
    oArgs.AddObject "java.lang.Class", oXJB.FindClass(strHomeClassName)
    Set clsPortRemObj = oXJB.FindClass("javax.rmi.PortableRemoteObject")
    Set oHome = clsPortRemObj.narrow(oArgs)
    ' Return the actual bean
    Set XJBGetEJBHome = oHome
    Set oArgs = Nothing
    Set oJavaObj = Nothing
    Set clsPortRemObj = Nothing
    Set oHome = Nothing
    Exit Function
ExHandler:
    MsgBox Err.Description
End Function
```

Adding properties to java.util.Properties

The following code extract adds a property to the java.util.Properties object that is used in the launchClient() J2EE API:

```
'-------------------------------------------------------------
' XJBPutProperty - Add a string key/value to a Java
'                  Properties object
' Input:
'   oXJB              - Initialized ActiveX to EJB bridge object
'   oProperties       - A Instantiated java.util.Properties object
'   strKey            - String representing the Key of the Key=Value
'   strValue          - String representing the Value of the Key=Value
'-------------------------------------------------------------
Private Sub XJBPutProperty(oXJB As Object, oProperties As Object, _
                           strKey As String, strValue As String)
    ' Create an XJB.JMethodArgs object.
    ' The prototype for java.util.Properties.put() is:
    '   java.util.Properties.put(Object, Object)
    ' In this case we want to setup two Strings as properties.  A Key=Value pair.
    ' Because the ActiveX to EJB bridge cannot coerce data types from String to Object
       ' (or any other type of implicit coersion), we must manually tell the
       ' ActiveX to EJB bridge how to do this (map a String to an Object).
    ' The JMethodsArgs object is a container for Method Arguments.  Here we
    ' are adding two Objects, which are both Strings (Key and Value).
    Dim oArgs As Object
    Set oArgs = oXJB.GetArgsContainer
    ' Add the key and value to the JMethodArgs object and coerce them to Object types.
    oArgs.AddObject "java.lang.Object", strKey
    oArgs.AddObject "java.lang.Object", strValue
    ' Put the key/value into our Properties object using the put() method
    oProperties.put oArgs
    Set oArgs = Nothing
End Sub
```

Examples: ASP helper functions

The following Active Server Pages (ASP) code extracts can be used to help create various types of ActiveX to EJB bridge objects and Enterprise JaveBeans. This code is taken from the ASP samples included with the ActiveX to EJB Client provided with WebSphere Application Server. For information about the samples, see the related samples information.

- “Initializing the ActiveX to EJB bridge and JVM” (on page 23)
- “Creating a J2EE client container” (on page 23)
- “Getting an Enterprise JavaBean home object” (on page 24)

- “Adding properties to java.util.Properties” (on page 24)

For equivalent Visual Basic example code, see “Examples: Visual Basic helper functions” (on page 20).

Initializing the ActiveX to EJB bridge and JVM

The following code extract initializes the ActiveX to EJB bridge and the JVM:

```
'-------------------------------------------------------------
' Initialize the ActiveX to EJB bridge and the JVM
'-------------------------------------------------------------
Private Function XJBInit(strWAS_HOME, strJAVA_HOME, strNAMING_FACTORY)
    Dim oXJB
    ' Create an XJB.JClassFactory object.  This object
    ' is responsible for finding classes and instantiating
    ' objects.
    Set oXJB = Server.CreateObject("XJB.JClassFactory")
    ' If you intend to debug using a JPDA debugger, you will need tools.jar
    ' strClassPath = strClassPath & ";" & strJAVA_HOME & "\lib\tools.jar"
    ' The ActiveX to EJB bridge properties are in the form of an array of strings.
    ' Each property that you would normally pass on a command-line
    ' is passed as an individual string in the array.
    ' Here, we are adding the classpath (put together above), extension directories
    ' which are required for J2EE access and the java.naming.factory.initial property
    ' which is always the same.
    Dim astrJavaProps(2)
    astrJavaProps(0) = "-Djava.ext.dirs=" & strWAS_HOME & "\classes;" & _
                       strWAS_HOME & "\lib\ext;" & strWAS_HOME & "\lib;" & _
                       strJAVA_HOME & "\jre\lib\ext;" & strWAS_HOME & "\properties;"
    astrJavaProps(1) = "-Dws.ext.dirs=" & strWAS_HOME & "\classes;" & _
                       strWAS_HOME & "\lib\ext;" & strWAS_HOME  & "\lib;" & _
                       strJAVA_HOME & "\jre\lib\ext;" & strWAS_HOME & "\properties;"
    astrJavaProps(2) = "-Djava.naming.factory.initial=" & strNAMING_FACTORY
    ' If you want to add more to the class path, you can do it here
    ' astrJavaProps(3) = "-Djava.class.path=" & strClassPath
    ' If you wish to debug using a JPDA debugger (you can't use JDB because there is
    ' no console to get the agent password), use parameters similar to the following
    '    astrJavaProps(3) = "-Djava.compiler=NONE"
    '    astrJavaProps(4) = "-Xnoagent"
    '    astrJavaProps(5) = "-Xdebug"
    '    astrJavaProps(6) =
"-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000"
    ' Initialize the JVM using our properties that we just setup.
    ' The JVM runs in Visual Basic's process space (the IDE if running
    ' in the Visual Basic interactive debugger)
    ' The JVM remains in this process space until the process ends.
    ' Running XJBInit a second time has no effect.  It continues
    ' to use the classpath and properties of the JVM that were initially used.
    oXJB.XJBInit astrJavaProps
    Set XJBInit = oXJB
End Function
```

Creating a J2EE client container

The following code extract creates a J2EE client container:

```
'-------------------------------------------------------------------
' XJBGetJ2EEClientContainer - Get the J2EE Client Container Object
' Input:
'   oXJB             - Initialized ActiveX to EJB bridge object
'   strEARFile       - Location where EJB stub jars and J2EE App Client Jar is stored
'   strBootstrapHost - Name of the server where the EJB app is running
'   strBootstrapPort - Port where the EJB server is running
'   strAppClientJar  - Name of the jar file in the EAR file that contains the Java
'                      Application Client.  We need this only when we have multiple
'                      client jars in a single ear.  Even though we don't run the
'                      Java client, the J2EE Client Container will use the manifest
'                      inside to reference the EJB JAR file(s) that we will need
'                      to reference.
'   strEARTempDir    - Directory to extract the EAR file to.
'-------------------------------------------------------------
Function XJBGetJ2EEClientContainer(oXJB, strEARFile, strBootstrapHost, _
                                   strBootstrapPort, strAppClientJar, _
                                   strEARTempDir)
    Dim oProps
    Dim oCC
    Dim astrAppParams(0)
    ' Initialize the J2EE Client Container
    ' Before initializing the Client Container, set the
    ' System Property that tells it what directory to
    ' extract the EAR file to.  If we don't do this, then
    ' a new directory is created each time we launch the
    ' Client Container and is not cleaned-up.
    if Trim(strEARTempDir) <> "" Then
        XJBPutProperty oXJB, oXJB.FindClass("java.lang.System").getProperties, _
                       "com.ibm.websphere.client.applicationclient.archivedir", _
                       strEARTempDir
    End If
    ' - Here we are creating a launchClient object which is supplied
    '   in appclient.jar supplied with WebSphere (Client or Server install).
    ' - Notice that we are eliminating the need to store the Class Proxy object
    '   returned by FindClass.  We are using as a temporary variable and immediately
    '   creating a new instance (Object Proxy) and throwing the Class Proxy away.
    Set oCC =
oXJB.NewInstance(oXJB.FindClass("com.ibm.websphere.client.applicationclient.launchClient"))
    ' Create a java.util.Properties object to use in our launchClient() J2EE API
    Set oProps = oXJB.NewInstance(oXJB.FindClass("java.util.Properties"))
    ' Add properties to the Properties object.
    ' - initonly=true is required to avoid running the Java Client that is stored
```

```
    '   in the EAR file.
    ' - BootstrapHost = "xxxx" : The name of the WebSphere server where your
    '                            Enterprise JsvaBeans are stored.
    ' - BootstrapPort = nnnn   : The port of the listener on the WebSphere server.
    ' - jar = "xxxx"           : The name of the jar within the ear file that
    '                            contains the Java Client (if you have more than one
    '                            Java Client Jar in the same EAR file.
    '                           (the path is relative to the root of the EAR file)
    XJBPutProperty oXJB, oProps, "initonly", "true"
    If Trim(strBootstrapHost) <> "" Then
        XJBPutProperty oXJB, oProps, "BootstrapHost", strBootstrapHost
    End If
    If IsNumeric(strBootstrapPort) Then
        XJBPutProperty oXJB, oProps, "BootstrapPort", strBootstrapPort
    End If
    If Trim(strAppClientJar) <> "" Then
        XJBPutProperty oXJB, oProps, "jar", strAppClientJar
    End If
    ' If we want to turn tracing on, we can do something like this:
    'XJBPutProperty oXJB, oProps, "trace",
"com.ibm.ws.client.applicationclient.ApplicationClientMetaData=debug=enabled"
    'XJBPutProperty oXJB, oProps, "tracefile", "c:\xjb.log"
    ' Add an empty string to the Application Parameters
           astrAppParams(0) = ""
    ' Launch the client container.
    ' In the java world, it would start the Java Application within the
    ' container.  Here, however, we ARE the application, so we avoid
    ' starting it by setting the property initonly=true (above).
    ' The Client Container overrides the Java Namespace so that we can load
    ' the appropriate EJB class files from within our supplied ear file.
    oCC.launch strEARFile, oProps, astrAppParams
    Set XJBGetJ2EEClientContainer = oCC
    Set oCC = Nothing
    Set oProps = Nothing
End Function
```

Getting an Enterprise JavaBean home object

The following code extract gets an Enterprise JavaBean's home object:

```
'-------------------------------------------------------------
' XJBGetEJBHome - Get the Home EJB Object
' Input:
'   oXJB             - Initialized ActiveX to EJB bridge Object
'   oInitialContext  - Java InitialContext object
'   strJNDIName      - JNDI Name of the EJB you are going to
'                      get the Home of.
'   strHomeClassName - Class Name of the Home object
'-------------------------------------------------------------
Private Function XJBGetEJBHome(oXJB, oInitialContext, strJNDIName, strHomeClassName)
    ' Create an XJB.JMethodArgs object.
    ' The prototype for javax.rmi.PortableRemoteObject.narrow() is:
    '   javax.rmi.PortableRemoteObject.narrow(Object, Class)
    ' In this case we want to use our Java Object and the Java Class that this
    ' Java Object needs to be "narrowed" to.
    ' Because the ActiveX to EJB bridge is unable to coerce data types from String to
    ' Object (or any other type of implicit coersion), we must manually tell the
    ' ActiveX to EJB bridge how to do this (map a String to an Object).
    ' The JMethodsArgs object is a container for Method Arguments.  Here we
    ' are adding two Objects, which are both Strings (Key and Value).
    Dim oArgs
    Set oArgs = oXJB.GetArgsContainer
    ' Lookup an object's home
    ' - This is typical J2EE Java code.  We use the JNDI Name that was
    '   passed-in to lookup our EJB
    Dim oJavaObj
    Set oJavaObj = oInitialContext.lookup(strJNDIName)
    ' Narrow it
    ' - This is more EJB/J2EE code.  Here we use or JMethodArgs object
    '   to coerce our EJB object to a specific class.   In this case
    '   our EJB Home class.
    ' - Notice that we are calling the static narrow() method on a Class Proxy
    '   object that was created from FindClass().
    Dim clsPortRemObj
    Dim oHome
    oArgs.AddObject "java.lang.Object", oJavaObj
    oArgs.AddObject "java.lang.Class", oXJB.FindClass(strHomeClassName)
    Set clsPortRemObj = oXJB.FindClass("javax.rmi.PortableRemoteObject")
    Set oHome = clsPortRemObj.narrow(oArgs)
    ' Return the actual bean
    Set XJBGetEJBHome = oHome
    Set oArgs = Nothing
    Set oJavaObj = Nothing
    Set clsPortRemObj = Nothing
    Set oHome = Nothing
End Function
```

Adding properties to java.util.Properties

The following code extract adds a property to the java.util.Properties object that is used in the launchClient() J2EE API:

```
'-------------------------------------------------------------
' XJBPutProperty - Add a string key/value to a Java
'                  Properties object
' Input:
'   oXJB             - Initialized ActiveX to EJB bridge object
'   oProperties      - A Instantiated java.util.Properties object
'   strKey           - String representing the Key of the Key=Value
'   strValue         - String representing the Value of the Key=Value
'-------------------------------------------------------------
Private Sub XJBPutProperty(oXJB, oProperties, strKey, strValue)
    ' Create an XJB.JMethodArgs object.
```

```
     ' The prototype for java.util.Properties.put() is:
     '   java.util.Properties.put(Object, Object)
     ' In this case we want to setup two Strings as properties.  A Key=Value pair.
     ' Because the ActiveX to EJB bridge cannot coerce data types from String to
     ' Object (or any other type of implicit coersion), we must manually tell
          ' the ActiveX to EJB bridge how to do this (map a String to an Object).
     ' The JMethodsArgs object is a container for Method Arguments.  Here we
     ' are adding two Objects, which are both Strings (Key and Value).
     Dim oArgs
     Set oArgs = oXJB.GetArgsContainer
     ' Add the key and value to the JMethodArgs object and coerce them to Object types.
     oArgs.AddObject "java.lang.Object", strKey
     oArgs.AddObject "java.lang.Object", strValue
     ' Put the key/value into our Properties object using the put() method
     oProperties.put oArgs
     Set oArgs = Nothing
End Sub
```

Examples: Accessing Enterprise JavaBeans without a J2EE client container

The following code extracts can be used to access Enterprise JavaBeans through the ActiveX to EJB bridge *without using* a J2EE client container. This code is taken from the samples included with the ActiveX to EJB Client provided with WebSphere Application Server. For information about the samples, see the related samples information.

- “Visual Basic” (on page 25)
- “Active Server Pages” (on page 26)

Visual Basic

In this example, the Visual Basic code extract performs the following actions:

1. Initializes the ActiveX to EJB bridge and JVM
2. Creates an InitialContext using the specified provider URL and context factory
3. Accesses an Enterprise JavaBean home of the HelloEJB sample provided with WebSphere Application Server
4. Calls a method on the Enterprise JavaBean

These actions are performed using the example Visual Basic helper functions defined in “Examples: Visual Basic helper functions” (on page 20).

Note: Because the code extract does not use a J2EE client container, it sends the JNDI name to XJBGetEJBHome() instead of using a java:comp resource name. The code also needs the WebSphereSamples.HelloEJB.HelloHome class stubs in its classpath (which is not illustrated here). The stubs can be extracted from the Enterprise JavaBean JAR file. For information about how to change the classpath, see the XJBInit Visual Basic helper function.

```
 ' Some Globals:
 Dim oXJB as Object
 Sub Main()
     ' Initialize the ActiveX to EJB bridge using our Helper function above.
     ' Store it in a global variable.  In this example, it is initialized once only
          ' and uses values from our environment that was initialized by
launchClientXJB.bat
          ' or setupCmdLineXJB.bat
         If oXJB Is Nothing Then
         Set oXJB = XJBInit(Environ("WAS_HOME"), Environ("JAVA_HOME"),
Environ("NAMING_FACTORY"))
     End If
     ' Create the initial context
     ' - This creates a J2EE initial context.
     ' - We pass a Hashtable via the use of a JMethodArgs object to NewInstance
               Dim oArgs as Object
     Set oArgs = oXJB.GetArgsContainer
     ' Build our properties using the normal naming string constants.  Here they are
     ' static Fields of the javax.naming.Context object
     Dim Context_PROVIDER_URL As String
     Dim Context_INITIAL_CONTEXT_FACTORY As String
     Context_PROVIDER_URL = oXJB.FindClass("javax.naming.Context").PROVIDER_URL
     Context_INITIAL_CONTEXT_FACTORY =
oXJB.FindClass("javax.naming.Context").INITIAL_CONTEXT_FACTORY
     ' We connect to MyServer.com and pull the Initial Context Factory out
     ' of an environment variable which was set in launchClientXJB.bat.
     Dim oHT As Object
     Set oHT = oXJB.NewInstance(oXJB.FindClass("java.util.Hashtable"))
     XJBPutProperty oXJB, oHT, Context_PROVIDER_URL, "iiop://MyServer.com"
     XJBPutProperty oXJB, oHT, Context_INITIAL_CONTEXT_FACTORY,
Environ("NAMING_FACTORY")
     Dim initialContext As Object
     Set initialContext =
oXJB.NewInstance(oXJB.FindClass("javax.naming.InitialContext"), oHT)
```

```
     ' Get EJB Home
     ' We specify the following:
     ' - InitialContext: Our J2EE InitialContext
     ' - JNDIName:       The JNDI Name of our EJB using
     ' - HomeClassName:  Name of the home class.  This is pulled out of the
     '                   appropriate EJB jar file within our EAR.
     Dim hello As Object
     Dim helloHome As Object
     Set helloHome = XJBGetEJBHome(oXJB, initialContext, "WSsamples/HelloEJBHome", _
                                   "WebSphereSamples.HelloEJB.HelloHome")
     ' Create the EJB Home
     ' Note:  We use CallByName() because Visual Basic likes to
     'Capitalize the "create" method as Create, which
     ' doesn't really exist
           set hello = CallByName(helloHome, "create", vbMethod)
     ' Execute a method on our EJB.
     MsgBox hello.getMessage()
 End Sub
```

Active Server Pages

In this example, the Active Server Pages code extract performs the following actions:

1. Initializes the ActiveX to EJB bridge and JVM
2. Creates an InitialContext using the specified provider URL and context factory
3. Accesses an Enterprise JavaBean home of the HelloEJB sample provided with WebSphere Application Server
4. Calls a method on> the Enterprise JavaBean

These actions are performed using the example ASP helper functions defined in “Examples: ASP helper functions” (on page 22).

Note: Because the code extract does not use a J2EE client container, it sends the JNDI name to XJBGetEJBHome() instead of using a java:comp resource name. The code also needs the WebSphereSamples.HelloEJB.HelloHome class stubs in its classpath (which is not illustrated here). The stubs can be extracted from the Enterprise JavaBean JAR file. For information about how to change the classpath, see the XJBInit ASP helper function.

```
 <-- #include virtual ="/WSASPIncludes/setupASPXJB.inc" -->
 <%
 Sub Main()
     ' Initialize the ActiveX to EJB bridge using our Helper function above.
     ' Store it in a global variable.  In this example, it is initialized once only
        ' and uses values from our environment that was initialized in our
        ' setupASPXJB.inc include file
           Application.lock
     If IsEmpty(Application("oXJB")) Then
         Set Application("oXJB") = XJBInit(com_ibm_websphere_washome,
com_ibm_websphere_javahome,com_ibm_websphere_namingfactory)
     End If
     Application.unlock
     ' Create the initial context
     ' - This creates a J2EE initial context.
     ' - We pass a Hashtable via the use of a JMethodArgs object to NewInstance
           Dim oArgs
     Set oArgs = Application("oXJB").GetArgsContainer
     ' Build our properties using the normal naming string constants.  Here they are
     ' static Fields of the javax.naming.Context object
     Dim Context_PROVIDER_URL
     Dim Context_INITIAL_CONTEXT_FACTORY
     Context_PROVIDER_URL =
Application("oXJB").FindClass("javax.naming.Context").PROVIDER_URL
     Context_INITIAL_CONTEXT_FACTORY =
Application("oXJB").FindClass("javax.naming.Context").INITIAL_CONTEXT_FACTORY
     ' We connect to MyServer.com and pull the Initial Context Factory out
     ' of an environment variable which was set in launchClientXJB.bat.
     Dim oHT
     Set oHT =
Application("oXJB").NewInstance(Application("oXJB").FindClass("java.util.Hashtable"))
     XJBPutProperty Application("oXJB"), oHT, Context_PROVIDER_URL,
"iiop://MyServer.com"
     XJBPutProperty Application("oXJB"), oHT, Context_INITIAL_CONTEXT_FACTORY,
com_ibm_websphere_namingfactory
     Dim initialContext
     Set initialContext =
Application("oXJB").NewInstance(Application("oXJB").FindClass("javax.naming.InitialContext"),
oHT)
     ' Get EJB Home
     ' We specify the following:
     ' - InitialContext: Our J2EE InitialContext
     ' - JNDIName:       The JNDI Name of our EJB using
     ' - HomeClassName:  Name of the home class.  This is pulled out of the
     '                   appropriate EJB jar file within our EAR.
     Dim hello
     Dim helloHome
     Set helloHome = XJBGetEJBHome(Application("oXJB"), initialContext,
"WSsamples/HelloEJBHome", _
                                   "WebSphereSamples.HelloEJB.HelloHome")
     ' Create the EJB Home
           set hello = helloHome.create
     ' Execute a method on our EJB.
```

```
    Response.Write(hello.getMessage())
End Sub
%>
```

Examples: Accessing Enterprise JavaBeans through a J2EE client container

The following code extracts can be used to access Enterprise JavaBeans through the ActiveX to EJB bridge using a J2EE client container. This code is taken from the samples included with the ActiveX to EJB Client provided with WebSphere Application Server. For information about the samples, see the related samples information.

- “Visual Basic” (on page 27)
- “Active Server Pages” (on page 27)

For equivalent example code to access Enterprise JavaBeans without using a J2EE client container, see “Examples: Accessing Enterprise JavaBeans without a J2EE client container” (on page 25).

Visual Basic

In this example, the Visual Basic code extract performs the following actions:

1. Initializes the ActiveX to EJB bridge and JVM
2. Creates a J2EE client container
3. Creates an InitialContext using the namespace generated by the client container
4. Accesses an Enterprise JavaBean home of the HelloEJB sample provided with WebSphere Application Server
5. Calls a method on the Enterprise JavaBean

These actions are performed using the example Visual Basic helper functions defined in “Examples: Visual Basic helper functions” (on page 20)

```
 ' Some Globals:
 Dim oXJB as Object
 Dim oCC as Object
 Sub Main()
     ' Initialize the ActiveX to EJB bridge using our Helper function above.
     ' Store it in a global variable.  In this example, it is initialized once only.
     If oXJB Is Nothing Then
         Set oXJB = XJBInit(Environ("WAS_HOME"), Environ("JAVA_HOME"),
Environ("NAMING_FACTORY"))
     End If
     ' Initialize the J2EE Client Container using our Helper function above.
     ' Store it in a global variable.  In this example, it is initialized once only
     ' and we use the XMI/XML files from our client Jar stored in Samples.ear.
         ' It later connects to a server called MyServer.com and use port 900 (the
default).
         ' If we had more than one client JAR file in our EAR, then we could specify the
name
         ' of the Client Jar as the fourth parameter.
             If oCC Is Nothing Then
         Set oCC = XJBGetJ2EEClientContainer(oXJB, "c:\myearfiles\Samples.ear",
"MyServer.com", "900", "")
     End If
     ' Create the initial context
     ' - This creates a J2EE initial context that allows us to use
     '   the java:comp namespace.
     Dim initialContext As Object
     Set initialContext =
oXJB.NewInstance(oXJB.FindClass("javax.naming.InitialContext"))
     ' Get EJB Home
     ' We specify the following:
     ' - InitialContext: Our J2EE InitialContext
     ' - JNDIName:       The JNDI Name of our EJB using the java:comp namespace
     ' - HomeClassName:  Name of the home class.  This is pulled out of the
     '                   appropriate EJB jar file within our EAR.
     Dim hello As Object
     Dim helloHome As Object
     Set helloHome = XJBGetEJBHome(oXJB, initialContext, "java:comp/env/ejb/Hello", _
                                   "WebSphereSamples.HelloEJB.HelloHome")
     ' Create the EJB Home
     ' Note:  We use CallByName() because Visual Basic likes to
     'Capitalize the "create" method as Create, which
     ' doesn't really exist
             set hello = CallByName(helloHome, "create", vbMethod)
     ' Execute a method on our EJB.
     MsgBox hello.getMessage()
 End Sub
```

Active Server Pages

In this example, the Active Server Pages code extract performs the following actions:

1. Initializes the ActiveX to EJB bridge and JVM
2. Creates a J2EE client container
3. Creates an InitialContext using the namespace generated by the client container
4. Accesses an Enterprise JavaBean home of the HelloEJB sample provided with WebSphere Application Server
5. Calls a method on> the Enterprise JavaBean

These actions are performed using the example ASP helper functions defined in “Examples: ASP helper functions” (on page 22)

```
<-- #include virtual ="/WSASPIncludes/setupASPXJB.inc" -->
<%
Sub Main()
            ' Use an Application lock to ensure that both the ActiveX to EJB bridge and
Client Container objects are
            ' created in the same thread.
            Application.Lock
    ' Initialize the ActiveX to EJB bridge using our Helper function above.
    ' Store it in a global variable.  In this example, it is initialized once only.
        ' The com_ibm_websphere... variables come from
    ' our setupASPXJB.inc file above.
    If IsEmpty(Application("oXJB")) Then
        Set Application("oXJB") = XJBInit(com_ibm_websphere_washome,
com_ibm_websphere_javahome,com_ibm_websphere_namingfactory)
    End If
    ' Initialize the J2EE Client Container using our Helper function above.
    ' Store it in a global variable.  In this example, it is initialized once only
        ' and we use the XMI/XML files from our client Jar stored in Samples.ear.
        ' It later connects to a server called MyServer.com and uses port 900 (the
default).
        ' If we had more than one client JAR file in our EAR, then we could specify the
name
        ' of the Client Jar as the fourth parameter.
            If IsEmpty(Application("oCC")) Then
        Set Application("oCC")= XJBGetJ2EEClientContainer(Application("oXJB"),
"c:\myearfiles\Samples.ear", "MyServer.com", "900", "")
    End If
            Application.unlock
    ' Create the initial context
    ' - This creates a J2EE initial context that allows us to use the java:comp
namespace.
    Dim initialContext
    Set initialContext =
Application("oXJB").NewInstance(Application("oXJB").FindClass("javax.naming.InitialContext"))
    ' Get EJB Home
    ' We specify the following:
    ' - InitialContext: Our J2EE InitialContext
    ' - JNDIName:       The JNDI Name of our EJB using the java:comp namespace
    ' - HomeClassName:  Name of the home class.  This is pulled out of the
    '                   appropriate EJB jar file within our EAR.
    Dim hello
    Dim helloHome
    Set helloHome = XJBGetEJBHome(Application("oXJB"), initialContext,
"java:comp/env/ejb/Hello", _
                                  "WebSphereSamples.HelloEJB.HelloHome")
    ' Create the EJB Home
    set hello = helloHome.create
    ' Execute a method on our EJB.
    Response.Write(hello.getMessage())
End Sub
%>
```

ActiveX to EJB bridge reference articles

This part contains reference topics about the ActiveX to EJB bridge provided by WebSphere Application Server 4.0 enterprise services. These topics are intended to provide reference information that provides extra detailed information relevant to the ActiveX to EJB bridge that support the concept and task articles.

- “ActiveX to EJB bridge concept articles” (on page 1)
- “ActiveX to EJB bridge example articles” (on page 19)
- “ActiveX to EJB bridge task articles” (on page 16)

ActiveX to EJB bridge, environment and configuration

This topic provides reference information about the aids that client applications and client services can use to access the ActiveX to EJB bridge. This enables the ActiveX to EJB bridge to find its XJB.JAR file and the Java runtime.

Client applications

The following batch files are provided for client applications to use the ActiveX to EJB bridge:

setupCmdLineXJB.bat

Sets the client environment variables

launchClientXJB.bat

Calls setupCmdLineXJB.bat and launches the application you specify as its arguments; for example:

```
launchClientXJB.bat myapp.exe  parm1 parm2
```

or

```
launchClientXJB MyApplication.vbp
```

These batch files set the following environment variables:

JAVA_HOME

Path to the Java runtime directory installed with the WebSphere Advanced Server Client.

WAS_HOME

Path to the WebSphere Advanced Server Client directory.

NAMING_FACTORY

Used to setup the Java java.naming.factory.initial system property

COMPUTERNAME

(Optional) Name of the computer where the WebSphere Advanced Server Client is installed. If you intend to talk to a single specific computer, you are recommended to change this value to become the server name that you intend to access.

Client services

The following aids are provided for client services to set some environment variables globally in the system environment or in your program:

System Settings

To enable the ActiveX to EJB bridge to access the Java runtime DLLs, the following directories must exist in the system PATH environment variable:

```
was_client_home \java\jre\bin;was_client_home \java\jre\bin\classic
```

Where *was_client_home* is the name of the directory where you installed the WebSphere Application Server Client (for example, `C:\WebSphere\AppClient`).

Note: This technique enables only *one* Java runtime to be activated on a machine, therefore all client services on that machine must use the same Java runtime. Client applications do not have this limitation, because they each have their own private, non-system scope.

ASP include file

An include file is provided for ASP users to use to automatically set the

following page-level (local) environment variables:

com_ibm_websphere_javahome

Path to the Java runtime directory installed with the WebSphere Advanced Server Client.

com_ibm_websphere_washome

Path to the WebSphere Advanced Server Client directory.

com_ibm_websphere_namingfactory

Sets the Java java.naming.factory.initial system property.

com_ibm_websphere_computername

(Optional) Name of the computer where the WebSphere Advanced Server Client is installed. If you intend to talk to a single specific computer, you are recommended to change this value to become the server name that you intend to access.

The include file is located in the *was_client_home*\Enterprise\WSASPIncludes directory. You can include the file into your ASP application with the following syntax in your ASP page:

```
<--  #include virtual ="/WSASPIncludes/setupASPXJB.inc" -->
```

This assumes that you have created a virtual directory in Internet Information Server called **WSASPIncludes** that points to the *was_client_home*\Enterprise\WSASPIncludes directory.

ActiveX to EJB bridge, class reference

This topic provides reference information about the object classes of the ActiveX to EJB bridge.

XJB.JClassFactory

JClassFactory is the object used to access the majority of the JVM's features. It handles JVM initialization, accessing classes and creating class instances (objects). The majority of tasks for accessing your Java classes and objects are handled with the JClassProxy and JObjectProxy objects.

XJBInit(String astrJavaParameterArray())

Initializes the JVM environment using an array of Strings that represent the command-line parameters you would normally send to java.exe.

JClassProxy FindClass(String strClassName)

Uses the current thread's class loader to load the specified fully-qualified class name and returns a JClassProxy object representing the Java Class object.

JObjectProxy NewInstance()

Creates a Class Instance for the specified JClassProxy object using the parameters supplied to call the Class Constructor. For more information about using JMethodArgs, see “ActiveX to EJB bridge, calling Java methods” (on page 4).

```
JObjectProxy NewInstance(JClassFactory obj, Variant vArg1, Variant vArg2, Variant
vArg3, ...)
 JObjectProxy NewInstance(JClassFactory obj, JMethodArgs args)
```

JMethodArgs GetArgsContainer()

Returns a JMethodArgs object (Class instance).

XJB.JClassProxy

A JClassProxy object is created from the JClassFactory.FindClass() method and can also be created from any Java method call that would normally return a Java Class object. You can use this object as if you had direct access to the Java Class object. All of the class' static methods and fields are accessible as are the java.lang.Class methods. In case of a clash between static method names of the reflected user class and those of the java.lang.Class (for example, getName()), the reflected static methods would be executed first.

For example, here we have a static method called getName(). The java.lang.Class object also has a method called getName():

- In Java:

```
class foo{
        foo(){};
        public static String getName(){return "abcdef";}
        public static String getName2(){return "ghijkl";}
        public String toString2(){return "xyz";}
}
```

- In Visual Basic:

```
...
Dim clsFoo as Object
set clsFoo = oXJB.FindClass("foo")
clsFoo.getName()          ' Returns "abcdef" from the static foo class
clsFoo.getName2() ' Returns "ghijkl" from the static foo class
clsFoo.toString() ' Returns "class foo" from the java.lang.Class object.
oFoo = oXJB.NewInstance(clsFoo)
oFoo.toString() ' Returns some text from the java.lang.Object's
                  ' toString() method which foo inherits from.
oFoo.toString2() ' Returns "xyz" from the foo class instance
```

XJB.JObjectProxy

A JObjectProxy object is created from the JClassFactory.NewInstance() method, and can

be created from any Java method call that would normally return a Class Instance object. You can use this object as if you had direct access to the Java object and can access all the static methods and fields of the object. All of the object's instance methods and fields are accessible (including those accessible through inheritance).

XJB.JMethodArgs

The JMethodArgs object is created from the JClassFactory.GetArgsContainer() method. Use this object as a container for method and constructor arguments. You must use this object when overriding the object type when calling a method (for example, when sending a java.lang.String JProxyObject to a constructor that normally takes a java.lang.Object type).

There are two groups of methods to add arguments to the collection: **Add** and **Set**. You can use **Add** to add arguments in the order that they are declared. Alternatively, you can use **Set** to set an argument based on its position in the argument list (where the first argument is in position 1).

For example, if you had a Java Object Foo that took a constructor of Foo(int, String, Object), you could use a JMethodArgs object as shown in the following code extract:

```
...
Dim oArgs as Object
set oArgs = oXJB.GetArgsContainer()
oArgs.AddInt(CLng(12345))
oArgs.AddString("Apples")
oArgs.AddObject("java.lang.Object", oSomeJObjectProxy)
Dim clsFoo as Object
Dim oFoo as Object
set clsFoo = oXJB.FindClass("com.mypackage.foo")
set oFoo = oXJB.NewInstance(clsFoo, oArgs)
' To reuse the oArgs object, just clear it and use the add method
' again, or alternatively, use the Set method to reset the parameters
' Here, we'll use Set
oArgs.SetInt(1, CLng(22222))
oArgs.SetString(2, "Bananas")
oArgs.SetObject(3, "java.lang.Object", oSomeOtherJObjectProxy)
Dim oFoo2 as Object
set oFoo2 = oXJB.NewInstance(clsFoo, oArgs)
```

AddObject (String strObjectTypeName, Object oArg)

Adds an arbitrary object to the argument container in the next available position, casting the object to the class name specified in the first parameter. Arrays are specified using the traditional [ ] syntax; for example:

```
AddObject("java.lang.Object[][]", oMy2DArrayOfFooObjects)
```

or

```
AddObject("int[]", oMyArrayOfInts)
```

AddByte (Byte byteArg)

Adds a primitive byte value to the argument container in the next available position.

AddBoolean (Boolean bArg)

Adds a primitive boolean value to the argument container in the next available position.

AddShort (Integer iArg)

Adds a primitive short value to the argument container in the next available position.

AddInt (Long lArg)

Adds a primitive int value to the argument container in the next available position.

AddLong (Currency cyArg)

Adds a primitive long value to the argument container in the next available position.

AddFloat (Single fArg)

Adds a primitive float value to the argument container in the next available position.

AddDouble (Double dArg)

Adds a primitive double value to the argument container in the next available position.

AddChar (String strArg)

Adds a primitive char value to the argument container in the next available position.

AddString (String strArg)

Adds a primitive char value to the argument container in the next available position.

SetObject (Integer iArgPosition, String strObjectTypeName, Object oArg)

Adds an arbitrary object to the argument container in the specified position casting it to the class name or primitive type name specified in the second parameter. Arrays are specified using the traditional [ ] syntax; for example:

```
SetObject(1, "java.lang.Object[][]", oMy2DArrayOfFooObjects)
```

or

```
SetObject(2, "int[]", MyArrayOfInts)
```

SetByte (Integer iArgPosition, Byte byteArg)

Sets a primitive byte value to the argument container in the position specified.

SetBoolean (Integer iArgPosition, Boolean bArg)

Sets a primitive byte value to the argument container in the position specified.

SetShort (Integer iArgPosition, Integer iArg)

Sets a primitive short value to the argument container in the position specified.

SetInt (Integer iArgPosition, Long lArg)

Sets a primitive int value to the argument container in the position specified.

SetLong (Integer iArgPosition, Currency cyArg)

Sets a primitive long value to the argument container in the position specified.

SetFloat (Integer iArgPosition, Single fArg)

Sets a primitive float value to the argument container in the position specified.

SetDouble (Integer iArgPosition, Double dArg)

Sets a primitive double value to the argument container in the position specified.

SetChar (Integer iArgPosition, String strArg)

Sets a primitive char value to the argument container in the position specified.

SetString (Integer iArgPosition, String strArg)

Sets a java.lang.String value to the argument container in the position specified.

Object Item(Integer iArgPosition)

Returns the value of an argument at a specific argument position.

Clear()

Removes all arguments from the container. And resets the next available position to 1.

Long Count()

Returns the number of arguments in the container.