Microsoft Visual J++ 1.0 (Beta 2) Release Notes

This beta version of Microsoft Visual J++ allows you to create, build and debug Java programs under Microsoft Windows 95 and Microsoft Windows NT 4.0. Visual J++ is still in development and there are some known bugs that will be fixed before the final release. These release notes include important information on installing and using this beta version of Visual J++.

1.0 Installation

1.1 System Requirements

This beta version of Visual J++ has the following *minimum* hardware and software requirements:

- A 90MHz Pentium-based computer with 24MB of RAM
- Either Windows 95 or Windows NT v.4.0 (Beta)
- 35MB of free disk space

Note: Windows NT versions 3.1, 3.5 and 3.51 are not supported.

1.2 Updating Previous Versions of Visual J++Developer Studio

This beta version does not include all of the files necessary to update previous versions of Developer Studio (including Visual C++, Fortran, etc.). Please note the following consequences:

- If you have any earlier version of Developer Studio on your hard disk, you must install this beta of Visual J++ in a different directory.
- If you have installed an earlier beta release of Visual J++, it is recommended that you uninstall it before installing this release.
- If you have installed the "trial" version of this beta (downloaded from the web), it is recommended that you uninstall it before installing from this CD-ROM.

1.3 Installing Internet Explorer 3.0

You must install Internet Explorer 3.0 (Beta 2) before installing Visual J++. That version of IE3.0 has been included on the CD-ROM for your convenience.

- During the installation of IE3.0, if you are asked to replace a newer file with an older one, always **keep the newer file.**
- If you install IE30 **after** you install Visual J++, it will copy an **older** version of CLASSES.ZIP (without prompting) to your machine. This older version does not contain the com.ms.com.Variant CLASS, used for COM programming. If you find yourself in this state, reinstall Java Support for Internet Explorer by running \VM\IE30Java.exe from your Beta2 CD.

1.4 Installing Visual J++ 1.0 (Beta 2)

Please keep the following items in mind during installation:

- During the installation of Visual J++, if you are asked to replace a newer file with an older one, always **keep the newer file.**
- **Online registration is not supported for this beta release.** Please do not use online registration.
- The "Introducing Visual J++" video tutorial is not available in this release.
- In the "Custom Installation Option" configuration dialog, selecting "Java class library source code" will appear to work, but the appropriate source files will not be generated. Those source files are not available in this release.

2.0 Configuration

2.1 CLASSPATH

The Visual J++ compiler (JVC) uses the following technique to find .class files:

- The compiler first searches any directories specified in the registry key, HKEY_LOCAL_MACHINE/Software/Microsoft/JavaVM/Classpath.

- If you are compiling your program from within the IDE, the compiler will then search any directories specified in the Tools.Options.Directories dialog box.
- If you are compiling your program from the command line, the compiler will then search any directories specified in the CLASSPATH environment variable.

Note: When you compile programs from within the Visual J++ IDE, the value of the CLASSPATH environment variable is not used. Please use the Tools.Options.Directories dialog box instead.

3.0 Debugging

With this release of Visual J++, Java programmers can debug Java applets in Internet Explorer as well as Java applications in JVIEW.EXE, our stand-alone interpreter. Below are the known problems with debugging in this release:

Windows 95 - When you stop debugging an applet, the debugger will take approximately 30 seconds to exit. This will appear to be a hang, because the debugger will be in a running state. You will know the debugger has exited when [run] disappears from the title bar.

Windows NT 4.0 - After running an applet or application in the debugger, you will need to shutdown and restart Visual J++ before starting a new debug session.

4.0 Support Newsgroups

4.1 Beta 2 Support Newsgroups

For this beta release, we have created two sets of support newsgroups - the private Beta support newsgroups and the public Peer support newsgroups.

The private Beta supports newsgroups are the same groups that have been available during the Beta 1 timeframe. Microsoft Product Support engineers actively monitor these newsgroups and attempt to answer questions in a timely fashion. Instructions for accessing those newsgroups was included with the Beta 1 kit. Unfortunately we do not have the capacity to add additional users to those newsgroups at this time.

4.2 Peer Support Newsgroups

We have also created several public newsgroups for all users of this beta version of Visual J++. **These newsgroups are available only on the "msnews.microsoft.com" news server so you must configure your Internet newsreader program to use that news server in order to access these newsgroups.** If you do not have an Internet newsreader program already, check out http://www.microsoft.com/ie/imn for instructions on downloading Microsoft's Internet Mail and News 1.0 (Beta 2).

Note that these newsgroups **are not** actively monitored by Microsoft Product Support engineers.

- microsoft.public.visualj.installation
  Issues with installation and configuration can be posted here.
- microsoft.public.visualj.compiler
  Compiler syntax or code generation issues can be posted here.
- microsoft.public.visualj.debugger
  Debugger issues can be posted here.
- microsoft.public.visualj.dev-environment
  Issues with the editor, class view, and wizards can be posted here.
- microsoft.public.visualj.com-support
  Issues with COM support can be posted here.
- microsoft.public.visualj.discussion
  General Visual J++ discussions. Product-wide issues.
- microsoft.public.visualj.misc-tools
  Issues with command line tools can be posted here.

5.0 Java/COM Support

5.1 Introduction

For this beta release, the good news is that Java/COM support works. The not-so-good-news is that it has not been fully tested at this time. There are several Java/COM scenarios that we know work and there are several that we know do not work. These are listed below. We are including what we have at this point so that programmers that want to use COM with Java can see it in action and begin learning how to use it. We are working furiously to provide complete Java/COM support as soon as possible. Please check the microsoft.public.visualj.com-support newsgroup frequently for the latest information (see section 4.0 for details) .

5.2 What We Know Works

JavaBeep! JavaBeep is a complete Java/COM sample that you can build and run to see Java/COM support in action. The sample automatically registers a simple "Beeper" automation server. The Java code then uses that server to beep and create messages. JavaBeep can be found inside the integrated help system under the Samples.Microsoft.JavaBeep section.

Here are the steps to build and run JavaBeep:

1. Click on the icon to copy the JavaBeep project files then press "Copy All..." then press "OK"
2. Press "Close" then select File.Open Workspace from the main menu.
3. Use the Open dialog to navigate to the directory you copied the samples into (typically \msdev\samples\microsoft\javabeep) then double-click the javabeep.mdp file to open the project.
4. Select Build.Execute from the main menu. When asked for a classfile name, enter "javabeep" and press OK.
5. In a couple of seconds, you should see Internet Explorer appear with an animation of a spinning globe. Click the mouse anywhere near the globe should cause your computer to beep and the message "Hello World from Beeper" to appear.
6. To see the relevant Java code, go back to Visual J++, expand the "javabeep classes" node, expand the "javabeep" node, and double-click on the "mouseDown()" node.

- We have been able to use Java to interact with OLE automation servers generated by Visual Basic 4.0.
- We have been able to call into a Java program from a VB4 program using COM.
- What else? Let us know!

5.3 What We Know Does Not Work

- There is a limitation on the explicit use of the VARIANT type. Currently, a Java program can "put" variants, however it cannot "get" them. In COM terms, we do not yet support VT_VARIANT | VT_BYREF.
- Varients of type VT_DISPATCH and VT_UNKNOWN are not supported yet.
- Currently, you cannot pass references of existing COM objects into Java programs. A consequence of this is that, for this beta release, a Java applet cannot interact with an ActiveX control on the same HTML page.
- There are problems invoking Microsoft Excel as an automation server.

5.4 Java/COM Documentation

Here is an early draft of our on-line documentation for Java/COM support.

Using Java and COM Together

Visual J++ is the first compiler that lets you take full advantage of Java Support in Internet Explorer. Internet Explorer provides the reference implementation of the Java virtual machine (or VM) for Win32 operating systems. This VM supports integration between Java and the Component Object Model (COM), the protocol underlying the ActiveX platform, Automation, and Object Linking and Embedding (OLE).
While the Java Support in Internet Explorer by itself provides benefits for any Java applet, you must use a development tool that supports this integration in order to take full advantage of both Java and COM.
Using Visual J++, you can give your Java program access to any of the software components that support COM, including thousands of ActiveX controls and Automation objects.
This document describes the following scenarios for using Java and COM together:

- Controlling a Java Applet through Scripting
- Using a COM object from Java
- Exposing a Java Class as a COM Class

For background information on COM, see the Component Object Model Specification. For more information on developing for the ActiveX platform, see the ActiveX SDK. You can download it from http://www.microsoft.com/intdev/sdk.

Controlling a Java Applet through Scripting

Web developers can embed ActiveX controls into their web pages and drive them using VBScript. The ActiveX runtime for Java offers developers another option by making Java applets scriptable too. When a Java applet is running in a browser such as Internet Explorer, all the public methods and variables of the applet automatically become available to VBScript, JavaScript, or any other language supporting the ActiveX Scripting protocol.

As an example, consider a Java applet like the following:

```
class Buzzer extends Applet
{
     public int pitch;
     public void buzz()
     {
          //play a sound at the specified pitch
     }
     // other methods
     // ...
}
```

To include such an applet in your HTML page, simply use the <APPLET> tag, specifying the ID attribute so that you can refer to the applet from the scripting language:

```
<APPLET CODE="Buzzer.class" ID=doorbell>
```

You can then define buttons that allow the user to control the applet. For example:

```
<INPUT TYPE=BUTTON VALUE="Higher" NAME="BtnHigher">
<INPUT TYPE=BUTTON VALUE="Lower"  NAME="BtnLower">
<INPUT TYPE=BUTTON VALUE="Play"   NAME="BtnPlay">
```

In the VBScript portion of your HTML page, you can define `OnClick` handlers for each button, connecting them to the applet:

```
<SCRIPT LANGUAGE = VBScript>
Sub BtnHigher_OnClick
     doorbell.pitch = 880
End Sub

Sub BtnLower_OnClick
     doorbell.pitch = 440
End Sub

Sub BtnPlay_OnClick
     doorbell.buzz()
End Sub
</SCRIPT>
```

Now, when a user clicks a button the web page, the appropriate VBScript handler is invoked, which in turn manipulates the Java applet.

In this way, Java applets can be driven just like ActiveX controls. The script can read and write the public variables of the applet, as well as call its public methods (including passing parameters and reading return values).

Note that only the **Applet**-derived class is directly accessible from the scripting language. If your Java applet includes other classes that you want to be available to the scripting language, you must define public methods in your **Applet**-derived class that delegate to those classes.

You can also embed both Java applets and ActiveX controls in the same web page and use VBScript handlers to connect them. For example, you can read values from an ActiveX control and pass them to a Java applet, or vice versa.
This automatic scripting capability is provided by the Java Support in Internet Explorer. As a result, your script code can manipulate Java applets developed with any tool, not just ones developed with Visual J++.
For more information about VBScript, see the VBScript page at http://www.microsoft.com/vbscript.

Using a COM Object from Java

With compiler support for Java/COM integration, you can refer to COM objects directly from your Java source code. This makes any COM service, including those offered by programmable controls or Automation objects, available from any Java program.
In order to use a COM class from Java, you must first import it; this means creating a Java class that acts as a wrapper for the COM class. Visual J++ includes the Java Type Library Wizard for this purpose.
Type libraries are a mechanism defined by COM to store type information; each type library contains complete type information about one or more COM entities, including classes, interfaces, dispinterfaces, etc. For more information about type libraries, see Type Libraries and the Object Description Language, or the OLE SDK.

> To import a COM class for use in Java

1 From the Tools menu, choose Java Type Library Wizard.
The check-box list displays all the type libraries registered on your machine. This list reflects the entries beneath the \HKEY_CLASSES_ROOT\TypeLib key of the system registry.
3 Choose the type library that describes the COM class(es) you want to use.
4 Choose OK.
In the Output window, the Type Library Wizard prints out the **import** statement(s) appropriate for each package it has created. You can copy and paste these **import** statements into your Java source code; this allows you to refer to classes in those packages by their short names.

To find out what classes are available from a particular programmable control or Automation server, consult the documentation provided by the vendor. You can also use the OLE Object View tool, available from the Tools menu, to browse through the classes described by a type library.
As an example, suppose beeper.DLL, which defines a Beeper class and an IBeeper interface, is registered as having type library information. Running Java Type Library Wizard and selecting beeper.DLL would create a package containing the Java class Beeper and the Java interface IBeeper.
You can use the **import** statement in your Java source code to easily refer to classes in the beeper package. For example:

```
import beeper.*;   // import beeper package
```

This is identical to Java's syntax for importing an entire package.
You could also import a single class within a type library:

```
import beeper.Beeper;
```

Again, this is identical to Java's syntax for importing a single class.
The Java classes that wrap COM classes can be used just like any other Java classes. For example:

```
IBeeper testBeep = new Beeper();
int myTone = testBeep.getSound();
testBeep.putSound(64);
testBeep.Beep();
```

The above code declares an instance of the COM class Beeper, gets and sets the value of a property of the object, and then calls a method on the object. For more information, see Type Mappings Between Java and COM and COM Programming in Java and C++ Compared.
Note that the **import** statement is not strictly necessary. Recall that in Java, any class can be referenced using its fully qualified name (for example, **java.awt.Canvas**), as long as a class fitting that name can be found when searching the class path. The **import** statement merely allows you to refer to the class using its short name (for example, **Canvas**).

The same is true for COM classes. The **import** statement is not required in order to use the class, it simply makes allows the class to be referenced using its short name. Any COM class can always be referenced using its fully qualified name (`beeper.Beeper`, in the above example).

What the Java Type Library Wizard Does

The Java Type Library Wizard displays everything that has a registry entry under HKEY_CLASSES_ROOT\TypeLib. This is not restricted to .TLB files; .OLB, .OCX, .DLL and .EXE files can all contain type library information. For simplicity, this topic refers to all of these as "type libraries."
For each type library, the Java Type Library Wizard creates a directory below \windows\java\lib (or \winnt\java\lib if you're running Windows NT 4.0) having the same name as the type library. The Java Type Library Wizard fills that directory with .CLASS files, one for each COM class and interface described in the type library. All the generated classes and interfaces are part of a Java package having the name as the type library file.
For example, to return to the beeper example, running the Java Type Library Wizard on beeper.DLL would create the files Beeper.class and IBeeper.class and place them in the directory \windows\java\lib\beeper. Because \windows\java\lib is on the class path, the compiler can find the beeper directory and its contents, and the statement "`import beeper.*;`" allows you to refer to the classes by their short names.
Each .CLASS file generated by the Java Type Library Wizard contains a special attribute identifying it as a wrapper for a COM class. When the Java Support in Internet Explorer sees this attribute on a class, it translates all Java method invocations on the class into COM function invocations on the COM class. (For a COM interface that defines properties, the Java wrapper defines two methods for each property, named **get**<*property*> and **put**<*property*>. See Type Mappings Between Java and COM.)
Once you have run the Java Type Library Wizard for a given type library, you don't need to re-run the wizard unless the type library has changed. Once the wizard has generated .CLASS files from a type library, the type library is no longer needed. Consequently, when distributing an applet that uses COM, you need to provide only the .OCX, .DLL or .EXE file for the COM objects, plus the .CLASS files that wrap them, not the .TLB file. (You must, of course, include the .CLASS files for the applet itself. You must also register the COM object appropriately.)
Note From the command line, you can use the JavaTLB tool to generate Java class wrappers out of type library information. Simply type "javaTLB *filename*", where *filename* is the name of the .TLB file (or .OCX, .DLL, or .EXE file containing the type library information.) This will create the appropriate directory and .CLASS files under \windows\java\lib.

When using COM from applets embedded on a web page, the .OCX or .DLL files that implement the COM object should be code signed, as well as any applets that make calls to them. For more information on code signing, see the ActiveX SDK. (For "intranet" applications, where all applets and ActiveX controls originate inside your company, code signing may not be necessary.)

Type Mappings Between Java and COM

The Java Support in Internet Explorer allows any construct that can be specified in a type library to be accessed from Java. COM constructs that cannot be described in a type library are inaccessible from Java. There are five basic types of entities that can be defined in Object Definition Language (ODL), and thus in a type library. These are mapped to Java as follows:

- COM classes (declared with the **coclass** keyword) are visible in Java as classes.
- Custom (that is, v-table based) interfaces (declared with the **interface** keyword) are visible in Java as interfaces.
- Dispatch interfaces (declared with the **dispinterface** keyword) are visible in Java as interfaces; properties in the dispatch interface are accessible through two methods named **get**<*property*> and **put**<*property*> in the Java interface.
- Structures (declared with the **typedef** keyword) are visible in Java as classes.
- Module statements (declared with the **module** keyword) are ignored.

ODL types map to Java types in the following manner:

ODL Type	**Java Type**

boolean	boolean
char	char
double	double
int	int
float	float
long	int
short	short
unsigned char	byte
wchar_t	short
BSTR	class **java.lang.String**
CURRENCY	long
DATE	double
SCODE/HRESULT	int (see also class **com.ms.com.ComException**)
VARIANT	class **com.ms.com.Variant**
IDispatch *	class **java.lang.Object**
IUnknown *	class **java.lang.Object**
SAFEARRAY(<typename>)	array of <typename>
<typename> *	one-element array of <typename>
void	void
LPSTR	class **java.lang.String**

For information on accessing the contents of a VARIANT structure, see the class **com.ms.com.Variant**.
For more information about type libraries, see Type Libraries and the Object Description Language, or the OLE SDK.
A parameter marked with the **retval** attribute in ODL is treated as the return value in the corresponding Java method. For example, consider the following method declaration in ODL:

```
HRESULT foo( [in] char x, [out, retval] int *y );
```

The corresponding Java method would look like this:

```
int foo( char x );
```

For information on the HRESULT returned by a COM function, see Handling COM Errors in Java.

COM Programming in Java and C++ Compared

For those who are familiar with COM programming in C++, this topic compares it with COM programming in Java. Java's notion of multiple interfaces on an object, along with Java's garbage collection, map readily to COM's paradigm for managing objects. For more information on COM, see the Component Object Model Specification.

Object Allocation

In C++, you allocate a new COM object using syntax like the following:

```
IBar pBar;
CoCreateInstance(CLSID_FooBar, NULL, CLSCTX_SERVER,
                 IID_IBar, (void**)&pBar );
```

In Java, the equivalent code would look like this:

```
IBar bar = new FooBar();
```

This is identical to Java's syntax for allocating ordinary Java objects. So, even though this line performs a call to the COM function **CoCreateInstance** instead of allocating space in the runtime heap, it is indistinguishable from pure Java.

Changing Interfaces

COM objects must implement the **IUnknown::QueryInterface** function to allow clients to switch between the object's supported interface. In Java, the details are handled by the Java Support for Internet Explorer, so at the source-code level it is simply done with a typecast.
For example, in C++ you might have code like this:
```
// pBar points to an object that supports IBar and IFoo
IFoo *pFoo;
pBar->QueryInterface( IID_IFoo, (void **)&pFoo );
```

In Java, the equivalent code would look like this:
```
// bar refers to an object that supports IBar and IFoo
IFoo foo;
foo = (IFoo)bar;
```

A **ClassCastException** is thrown if the object does not implement the requested interface. You can also use the **instanceof** operator to check beforehand whether the object implements the requested interface. Again, this is identical to Java's syntax for casting between ordinary Java object types.

Object Identity

In COM, object identity is defined as having the same **IUnknown** pointer. For example, in C++, to check whether two pointers refer to the same object, you need code like this:
```
// comparing pFoo and pBar
IUnknown pUnk1, pUnk2;
pFoo->QueryInterface( IID_IUnknown, (void **)&pUnk1 );
pBar->QueryInterface( IID_IUnknown, (void **)&pUnk2 );
if ( pUnk1 == pUnk2 )
    printf( "Foo and Bar point to same object" );
```

In Java, the equivalent code would look like this:
```
// comparing foo and bar
if (foo == bar )
    System.out.println( "Foo and Bar are same object" );
```

As in the previous examples, this is identical to Java's syntax for comparing two object references.

Reference Counting

Reference counting is handled automatically in Java. There is no need to call **IUnknown::AddRef** when creating a new reference to an object, nor do you need to call **IUnknown::Release** when you're finished using a reference to an object. The Java garbage collector automatically keeps track of how many references there are to an object.

Reuse

In C++, you can extend the functionality of an existing COM class through aggregation. In Java, you can do the same simply using the **extends** clause, just as you would extend an ordinary Java class. However, because Java uses a single inheritance model, you can extend only a single COM class, even though COM allows multiple aggregation. (In the rare situation where the original COM class does not support aggregation, extending the class will not work; an exception is thrown when you try allocating an instance of the derived class. See Handling COM Errors in Java for more information.)

Summary

As shown by these examples, COM programming is much simpler in Java than in C++. The Java Support in Internet Explorer performs all the tedious work for you, making COM classes as easy to use as native Java classes.

Handling COM Errors in Java

The Java Support for Internet Explorer defines a class called **com.ms.com.ComException**. This is used to communicate error information from COM back to Java whenever a COM method fails.
Whenever a COM method is exposed to Java, it is exposed as a Java method with an implicit **throws** clause:

```
char foo( int x ) throws com.ms.com.ComException;
```

COM methods typically return a value known as an HRESULT. The **ComException** class defines a **getHResult** method that returns the error code describing the specific error.

```
try
{
    // call COM methods
}
catch (com.ms.com.ComFailException e)
{
    System.out.println( "COM Exception:" );
    System.out.println( e.getHResult() );
}
```

Note that the **ComException** class is referenced using its fully-qualified name. If you added an "`import com.ms.com.*;`" statement to the beginning of the file, you could refer to the class by its short name.
For those familiar with Visual Basic programming, this error handling mechanism is similar to the **On Error Goto** statement and the **Err** variable used in error handlers.

Exposing a Java Class as a COM Class

Besides allowing Java programs to use COM objects, the Java Support in Internet Explorer also allows Java programs to expose their functionality as COM services. This lets you use Java for developing component software without requiring that your clients use Java; your clients can use any language that is compatible with COM, such as Visual Basic or C++.
The first step is to use Object Definition Language (ODL) to define how your Java class will be exposed. For example, you might define a COM class that implements two interfaces. Define the interfaces in an ODL file:

```
// mystuff.odl
[ uuid (42E5AFC3-DBAA-11cf-BAFD-00AA0057B223) ]
library LMyStuff
{
     importlib("stdole32.tlb");

     [ uuid (42E5AFC4-DBAA-11cf-BAFD-00AA0057B223) ]
     interface IFoo
     {
          HRESULT blah1();
          HRESULT blah2();
     }

     [ uuid (42E5AFC5-DBAA-11cf-BAFD-00AA0057B223) ]
     interface IBar
```

```
    {
        HRESULT blah3();
        HRESULT blah4();
    }
    [ uuid (42E5AFC6-DBAA-11cf-BAFD-00AA0057B223) ]
    coclass MyClass
    {
        interface IFoo;
        interface IBar;
    }
};
```

You can expose your Java interfaces through COM as interfaces, dispatch interfaces, or dual interfaces. Services that cannot be described in a type library cannot be exposed by your Java program.
Use MkTypLib to build mystuff.ODL into mystuff.TLB. Then use the Java Type Library Wizard to generate .CLASS files based on the type library.
Next, create a .JAVA source file containing the Java implementation of the class. Declare a Java class that implements all the interfaces supported by the coclass in the .ODL file. The Java class is not required to have the same name that was used in the **coclass** statement, but using the same name may make your code more readable. For example:

```
import com.ms.com.*;
import mystuff.*;

class MyClass implements IFoo, IBar
{
    void blah1() throws ComException {}
    void blah2() throws ComException {}
    void blah3() throws ComException {}
    void blah4() throws ComException {}
}
```

The compiler will generate an error if the class does not implement all the methods defined by the interfaces specified in the .ODL file.
Compile the implementation .JAVA file into a .CLASS file. COM clients can read the .TLB file to determine what services your class exposes.
Finally, run the JavaReg tool from the command line to register your Java class as a COM class. Specify the name of the Java class and the CLSID that you want to register it under. For example:

```
javareg /register /class:MyClass /clsid:{42E5AFC6-DBAA-11cf-BAFD-00AA0057B223}
```

The CLSID you specify on the command line must match the one specified for the coclass in the .ODL file. It is this CLSID which associates the COM class defined in the .ODL file with the Java class you wrote.
The JavaReg tool creates registry entries that distinguish a COM class written in Java from other COM classes. The most important entries are as follows:

```
HKEY_CLASSES_ROOT
    CLSID
        {42E5AFC6-DBAA-11cf-BAFD-00AA0057B223}
            InprocServer32 = msjava.dll
                JavaClass = mystuff.MyClass
```

When an object of that CLSID is requested, the Java Support in Internet Explorer is launched and the specified class is loaded. The runtime exports a **DllGetClassObject** function and creates a COM class factory for the Java class. This means that a COM client doesn't need to know that a given class is implemented in Java. To create an instance of the class, the client simply passes the CLSID to **CoCreateInstance**, just as it would with any other COM class.
Note that for all COM classes implemented in Java, the value of the **InprocServer32** key is identical: MSJAVA.DLL, which contains the Java Support for Internet Explorer. What differs for each class is the value named **JavaClass** beneath the **InprocServer32** key.

If you distribute an COM class written in Java, you must provide an installation program to add the appropriate entries to the registry. In addition, your installation program should install the .CLASS file on the class path of the client machine so that the Java Support for Internet Explorer can find it.
Note that while any COM class qualifies as an ActiveX control, a COM object must implement many more interfaces in order to be a full-fledged control that can appear in tool palettes and be inserted in control containers.
Java classes exposed as COM classes are automatically aggregatable.

Returning HRESULTs from Java

Throw an instance of **com.ms.com.ComFailException** to indicate failure. You can specify a particular HRESULT when constructing the **ComFailException** object. The HRESULT will be used as the return value for the COM method.
To indicate successful completion, you don't need to do anything; just return normally. To return S_FALSE (indicating a successful completion but a return value of Boolean FALSE), throw an instance of **com.ms.com.ComSuccessException**.

Using JavaReg

Using JavaReg to generate a CLSID

If you aren't defining new interfaces, but are simply writing a Java class that implements existing interfaces, you may not need to write an .ODL file. First run the Java Type Library Wizard over the type library that describes the COM interfaces you want to implement; this generates .CLASS files for those interfaces. Then write a Java class that implements those COM interfaces.
Finally, run JavaReg with the /register and /class options, but without the /clsid option.This instructs JavaReg to generate a new CLSID and register the class using that value. You must record the CLSID that JavaReg generates, since this is the value you must publish when distributing your class. As long that the interfaces your class implements are properly registered, COM clients can specify your class's CLSID and use its interfaces as usual.

Remoting with DCOM

The Java Support in Internet Explorer runs as an in-process server, and in-process servers cannot normally be remoted using NT 4.0's Distributed COM (DCOM). However, it is possible to launch a "surrogate" .EXE in its own process that then loads the in-process server. This surrogate can then be remoted using DCOM, in effect allowing the in-process server to be remoted.
You can use JavaReg's /surrogate option to support remote access to a COM class implemented in Java . When first registering the class, specify the /surrogate option on the command line. For example:
```
javareg /register /class:MyClass /clsid:{42E5AFC6-DBAA-11cf-
BAFD-00AA0057B223} /surrogate
```

This adds a **LocalServer32** key to the registry in addition to the usual **InprocServer32** key. The command line under the **LocalServer32** key specifies JavaReg with the /surrogate but without the /register option.
```
HKEY_CLASSES_ROOT
     CLSID
          {42E5AFC6-DBAA-11cf-BAFD-00AA0057B223}
               InprocServer32 = msjava.dll
               LocalServer32 = javareg /clsid:{42E5AFC6-DBAA-11cf-
BAFD-00AA0057B223} /surrogate
```

This causes JavaReg to act as the surrogate itself. When a remote client requests services from the COM class that you've implemented using Java, JavaReg is invoked. JavaReg then loads the Java Support in Internet Explorer with the specified Java class.

You can remove the LocalServer32 key by rerunning JavaReg with the /class option, specifying the same class name, but without the /clsid or /surrogate options.

Other JavaReg Options

To remove the registry entries for a particular class, run JavaReg with the /unregister option and the /class option, specifying the name of the class.
JavaReg will accept a /progid option if you want to specify a ProgID for your class.

Variant

```
public final class com.ms.com.Variant
{
    // Fields
    public static final short VariantEmpty;
    public static final short VariantNull;
    public static final short VariantShort;
    public static final short VariantInt;
    public static final short VariantFloat;
    public static final short VariantDouble;
    public static final short VariantCurrency;
    public static final short VariantDate;
    public static final short VariantString;
    public static final short VariantDispatch;
    public static final short VariantError;
    public static final short VariantBoolean;
    public static final short VariantVariant;
    public static final short VariantObject;
    public static final short VariantByte;
    public static final short VariantTypeMask;
    public static final short VariantArray;
    public static final short VariantByref;
    // Constructors
    public          Variant()
    // Methods
    public          void    getEmpty();
    public          void    getNull();
    public          short   getShort();
    public          int     getInt();
    public          float   getFloat();
    public          double  getDouble();
    public          long    getCurrency();
    public          double  getDate();
    public  native  String  getString();
    public  native  Object  getDispatch();
    public          int     getError();
    public          boolean getBoolean();
    public  native  Object  getObject();
    public          byte    getByte();
    public          void    putEmpty();
    public          void    putNull();
    public          void    putShort(short   val);
    public          void    putInt(int     val);
    public          void    putFloat(float   val);
    public          void    putDouble(double  val);
    public          void    putCurrency(long    val);
    public          void    putDate(double  val);
    public  native  void    putString(String  val);
    public  native  void    putDispatch(Object  val);
    public          void    putError(int     val);
    public          void    putBoolean(boolean val);
    public  native  void    putObject(Object  val);
    public          void    putByte(byte    val);
    public  native  void    putEmptyRef();
    public  native  void    putNullRef();
    public  native  void    putShortRef(short[] val);
    public  native  void  putIntRef(int[] val);
    public  native  void  putFloatRef(float[] val);
    public  native  void  putDoubleRef(double[] val);
    public  native  void  putCurrencyRef(long[] val);
    public  native  void  putDateRef(double[] val);
    public  native  void  putStringRef(String[] val);
    public  native  void  putDispatchRef(Object[] val);
    public  native  void  putErrorRef(int[] val);
    public  native  void  putBooleanRef(boolean[] val);
    public  native  void  putObjectRef(Object[] val);
    public  native  void  putByteRef(byte[] val);
    public  native  short   toShort()
      throws ClassCastException;
    public  native  int     toInt()
      throws ClassCastException;
    public  native  float   toFloat()
      throws ClassCastException;
    public  native  double  toDouble()
      throws ClassCastException;
    public  native  long    toCurrency()
      throws ClassCastException;
    public  native  double  toDate()
      throws ClassCastException;
    public  native  String  toString()
      throws ClassCastException;
    public  native  Object  toDispatch()
      throws ClassCastException;
    public  native  int     toError()
      throws ClassCastException;
    public  native  boolean toBoolean()
      throws ClassCastException;
    public  native  Object  toObject()
      throws ClassCastException;
    public  native  byte    toByte()
      throws ClassCastException;
    public  native  void    VariantClear();
    public  native  Variant clone();
    public  native  Variant cloneIndirect();
    public  native  void    changeType(short vartype);
    public  native  void    changeType(int lcid,
      short flags,  short vartype);
    public        short     getvt();
}
```

Variant is the wrapper for the VARIANT structure.

ComException

```
public abstract class com.ms.com.ComException
    extends RuntimeException
{
    // Constructors
    public ComException(int hr);
    public ComException();
    public ComException(String message);
    // Methods
    public int getHResult();
}
```

ComException is the Java class that wraps an HRESULT, the return type for most methods in the Component Object Model (COM). When calling COM methods from Java, you catch **ComException** objects to handle error conditions.
When implementing COM classes using Java, you throw **ComException** objects to signal error conditions. Since **ComException** is an abstract class, so you cannot construct a **ComException** object directly. Instead, you construct either a **ComFailException** object or a **ComSuccessException** object.

ComException.ComException

public ComException( );
public ComException(int hr **);**
public ComException(String message **);**

Parameters
hr The HRESULT to return
message The detail message.
Remarks
Constructs a new **ComException** object. This constructor is called only by the constructors of the classes derived from **ComException**.

ComException.getHResult

public int getHResult( );

Remarks
Retrieves the HRESULT returned by the COM method. Call this method when catching a **ComException** thrown by a COM method.

ComFailException

```
public abstract class com.ms.com.ComFailException
    extends com.ms.com.ComException
{
    // Constructors
    public ComFailException(int hr);
    public ComFailException();
    public ComFailException(String message);
}
```

ComFailException indicates a failure. The default value of the stored HRESULT is E_FAIL.

ComFailException.ComFailException

public ComFailException( );
public ComFailException(int hr **);**
public ComFailException(String message **);**

Parameters

hr The HRESULT to return
message The detail message.
Remarks
Constructs a new **ComFailException** object. Throw a **ComFailException** object in order to signal an error condition when implementing a COM class using Java.

ComSuccessException

```
public abstract class com.ms.com.ComSuccessException
    extends com.ms.com.ComException
{
    // Constructors
    public ComSuccessException(int hr);
    public ComSuccessException();
    public ComSuccessException(String message);
}
```

ComSuccessException indicates a success. The default value of the stored HRESULT is S_FALSE, which is used to indicate the successful completion of a method returning a Boolean FALSE value.
To return an S_OK value when implementing a COM method in Java, simply have the method return as usual.

ComSuccessException.ComSuccessException

public ComSuccessException( );
public ComSuccessException(int hr **);**
public ComSuccessException(String message **);**

Parameters
hr The HRESULT to return
message The detail message.
Remarks
Constructs a new **ComSuccessException** object. Throw a **ComSuccessException** object in order to signal a successful return when implementing a COM class using Java.