

# Using RMI and JDBC From Visual Basic

The Microsoft JVM provides an opportunity to use almost any Java system API from COM automation controllers

Dan Adler

**Y**OU HAVE JUST written your first commercial client/server application in Java. Because you needed to pass complex Java objects like hash tables by value, you chose to use RMI. Some of your colleagues kept warning you: "Don't use RMI. It only works in a pure Java environment," but you didn't have the time to consider alternatives like CORBA or DCOM. You needed a fast and portable solution.

So, now your client and server are fully functional, running on both UNIX and Windows platforms, as your boss required. You have tested the server using a variety of client configurations including stand-alone,

Applets, and Servlets. You even tested the client on Internet Explorer 4, and it worked as expected, once you included the RMI class libraries.

You are feeling pretty good about this project, until one morning your boss wanders in and tells you that this huge new customer has made it a requirement that your client code be made available to their Microsoft Excel users as an add-in. At first, panic sets in. Not only have you never written Excel add-ins, but those colleagues of yours are starting to remind you how they told you not to paint yourself into a corner by using RMI. You start thinking about con-

## Using RMI and JDBC From Visual Basic

*"You can leverage your RMI code directly in VB, with almost no code changes."*

verting your code to CORBA, because the CORBA vendors support an OLE Automation bridge, which means you could use your code from Visual Basic (VB).

In your misery, you download the latest Microsoft SDK for Java and the accompanying documentation. As you feverishly scan the online documentation of the Microsoft VM for Java you find a promising paragraph: "The MS virtual machine (VM) automatically implements IUnknown and IDispatch for most Java classes at runtime." You think

about this for a while. Suddenly, nirvana sets in. You realize that neither Sun nor Microsoft have been telling you the full story on Java-COM interoperability. You *can* leverage your RMI code directly in VB, with almost no code changes. You code frantically for a few hours, and before the day is done, you walk into your boss's office with the working prototype. Your RMI client code works in Excel's VBA without modifications and with only one new 100% Pure Java helper class.

### KEEPING YOUR OPTIONS OPEN

If you are like most Java programmers, you are either firmly in the Sun camp or in the Microsoft camp. If you are in the 100% Pure Java camp, you have probably never downloaded the Microsoft SDK for Java because Sun has convinced you that it is "impure" in some way and will lead you to a Windows-only implementation. If you are in the Visual J++ camp, you have probably bought into the Microsoft marketing messages that you need to build Windows-specific features into your Java applications from the ground up, and you probably look down at those 100% Pure Java guys who are writing once and debugging everywhere. In both cases, you are left in the dark about some key interoperability issues. If you are in the "Pure" camp, then you can't effectively support existing desktop OLE Automation clients, whereas if you are using Microsoft's DCOM and data access APIs like ADO, then you cannot communicate with pure Java servers running on UNIX.

However, it turns out that there is a vast middle ground and you may be surprised to find out, the middle ground is provided by Mi-

crosoft. Without any marketing hype and without much publicity to developers, Microsoft has made their Java VM extend the reach of Java to support OLE Automation at runtime.

What is even more surprising to most Java programmers, is that to take advantage of these features, you don't even need to use Microsoft's Java compiler. You can sit there in the comfort of your own JDK or your Symantec Café development environment, code and debug your class files, and then deploy them as COM Automation objects using a simple registration process as part of your installation process. Furthermore, all you have to register are your top-level Java classes (e.g., those that will be explicitly created from VB using a **CreateObject** or **New**). Objects of unregistered classes that are returned from method calls are handled automatically.

Of course, there is no free lunch. You have to pay a price for this added functionality. The price comes in the form of some features that you cannot use in the Java code that you intend to call from COM. Most of these limitations are annoying and will require some workarounds. The good news is that almost all workarounds can be written in 100% Pure Java, so you don't need to write any MS Java-specific code unless you choose to. It is also very likely that Microsoft will fix these anomalies as time goes on, so in the future you may be able to use your Java code with no modifications.

### HOW THE MICROSOFT JAVA-AUTOMATION BRIDGE WORKS

The Microsoft VM implements the IDispatch interface at runtime for Java classes. What this means is that the VM is an Automation server, and when it gets an IDispatch method call on an object that is a Java object, it uses reflection to find the object's Java methods and map the call and arguments back and forth. So, the caller (which is VB or another Automation controller) sends in Automation data types (which are typically wrapped in structures called **Variants**) and the VM maps them to Java types. Then the method is called and the return value is mapped back to a Variant.

The process of mapping values from Variants to Java types is defined in the SDK documentation in terms of ODL types. ODL is Microsoft's Object Definition Language for building COM interfaces.

Table 1, from the SDK documentation, lists the ODL types that are supported and the Java type that each ODL type maps to.

**Table 1. ODL types and the Java types they map to.**

ODL TYPE	JAVA TYPE
Boolean	boolean
char	char
Double	double
int	int
int64	long
float	float
long	int
Short	short
Unsigned char	byte
BSTR	class java.lang.String
CURRENCY/CY	Long
DATE	double
SCODE/HRESULT	int (see also class ComException)
VARIANT	class com.ms.com.Variant
IUnknown *	interface com.ms.com.IUnknown
IDispatch *	class java.lang.Object
SAFEARRAY(typename)	class com.ms.com.SafeArray
Typename *	single-element array of typename on [out], error on [in]
void	void

The key points to note are that all the basic ODL types map directly to the corresponding Java types, including single and multidimensional arrays of such basic data types, which map cleanly as well. **BSTR**'s represent VB strings, and those map back and forth to **java.lang.String**. One key point to note is that **java.util.Date**'s do not map to Variant date types, so if you need to support dates in your interface, you will have to provide some Microsoft-specific Java code, which does the mapping using the SDK's **com.ms.com.Variant** class.

Also, note that all **Java Objects** (descendants of **java.lang.Object**) show up as another **IDispatch** interface pointer. This is the magic that allows you to return pure Java types such as **java.util.Hashtable**'s, which has no mapping, and invoke its methods directly using Automation.

The one question that remains is how does this whole process get bootstrapped, i.e., how do VB and other Automation controllers obtain an initial reference to a Java

object, because they cannot call "new" directly. The next section explains this.

## USING JAVAREG

The tool that makes this all possible is called **javareg** and you will find it in the SDK's Bin directory (you can download the SDK from Microsoft's Web site). The online documentation and the help are fairly meager, and you will need to understand some basic COM terminology to use it.

The key idea is simple: you register a compiled Java class using a Program ID (ProgID) and a globally unique identifier (GUID) called a Class ID (CLSID). ProgIDs map to CLSIDs through a COM API call to registry. The CLSID of the Java class appears in the system registry and is mapped to an in-process Automation server called **msjava.dll** (the MS Java VM) with a parameter that gives the class name and optional class path. This means that applications that never even heard of Java (but support OLE Automation) can load in Java classes and execute methods through the Automation mechanism.

In the specific case of VB, it turns out that you don't even have to use **javareg**, because the **GetObject** call supports a *Java Moniker*. Monikers use a URL-like syntax to locate objects, so you could use a moniker of the form **java:java.util.Date** to construct a date object. The main advantage of this method is that you don't need to put anything in the user's registry. However, most other Automation controllers don't currently support moniker-based object creation, so it's safer to use **javareg**.

**javareg** also has an option to build a COM type-library from your Java classes. This allows you to use a form of early binding in VB wherein your Java classes are recognized by the Object Browser inside the IDE as well as the IntelliSense method-completion features in the VB editor. At the time of this writing, it seems that you cannot build type-libraries for Java interfaces using **javareg**, only instances of Java classes.

## PITFALLS

There are currently some limitations on what can be seamlessly mapped from Java code to Automation for existing applications. Microsoft will hopefully fix some of these limitations in upcoming releases of the VM.

The main prerequisite is that your code should run on the Microsoft VM. If you write and compile your code using a Sun-based compiler, you will need to test it using Microsoft's *jview* or *IE4*. This is typically

## Using RMI and JDBC From Visual Basic

an easy step because, contrary to popular belief, almost everything I have tried with JDK 1.1 works well on the Microsoft VM.

Once your code works in the Microsoft VM, you still need to adhere to some limitations. The most difficult of these is that you can't use static methods. Currently, those are simply not mapped. So, you will need to write object method wrappers around calls to static methods. This can be especially annoying for factories where you use static methods by design. However, you can work around this limitation in pure Java, so it's not a disaster.

Another pitfall is overloaded Java methods with equal numbers of arguments. The VM simply picks the first one without matching the argument types. This is prob-

ably done as an optimization to avoid multiple calls to `IDispatch::GetIDsOfNames()`.

Finally, if your class is going to be instantiated from VB (using `CreateObject` or `New`) then it needs to have a constructor that takes no arguments, because you can't pass arguments to those methods.

All of these issues may force you to restructure your Java code or provide some wrappers. So far, all of these wrappers can be written and compiled in Sun JDK-based compilers. However, if you need to do some custom type conversions (e.g., `java.util.Date` to Variant Date's) then you may have to write some code that uses Microsoft-specific classes and which needs to be compiled using the SDK's JVC compiler and works only on Windows platforms. In most cases, you can do this as an add-on piece to your pure Java API without resorting to building the whole thing with Microsoft's compiler.

### Listing 1.

#### Using a simple RMI remote interface.

```
// LookupServer.java

import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class LookupServer extends Unicast
RemoteObject
    implements Lookup
{
    public LookupServer() throws RemoteException {}

    public String findInfo(String info)
    {
        return info;
    }

    public int addInt(int i, int j)
    {
        return i + j;
    }

    public double addDbI(double i, double j)
    {
        return i + j;
    }

    public static void main(String args[])
    {
        try {
            RMISecurityManager security =
                new RMISecurityManager();
            System.setSecurityManager(security);
            LookupServer server = new LookupServer();
            Naming.rebind("LookupServer", server);
            System.out.println("LookupServer ready...");
        }
        catch (Throwable e) {
            System.err.println("exception: " + e);

            System.exit(1);
        }
    }
}
```

### SUN'S JAVA PLUG-IN

Sun has recently released its Java Plug-in, which also includes its own ActiveX Bridge. This technology used to be a part of the Beans Developer Kit (with full source code), and is now packaged as the part of the Plug-in that enables it to function inside of ActiveX controllers (like IE4).

Although Sun's Java Plug-in also has much of the functionality needed to bridge Java APIs like RMI and JDBC into the ActiveX world, its packaging is currently more oriented toward the hosting of graphical Java beans. Unfortunately, it requires a bean to be fully packaged in a jar file before it can be bridged, whereas the Microsoft VM simply maps any Java class directly. On the other hand, Sun's bridge has some more general capabilities like the `getNew` method, which allows you to get a handle to a reflection-based Java constructor, so that you can construct Java objects at runtime.

It's very likely that future releases of this product will have more of the functionality that Java developers need to expose their APIs to ActiveX desktop applications. This should expose the full range of capabilities of the Java language and APIs to COM, and would hopefully include a dynamic way to install class translators (e.g., from `java.util.Date` to a Variant Date representation).

It is also conceivable that a third-party or shareware implementation of the Java-Automation bridge will emerge. Anyone with a good understanding of the Automa-

## Using RMI and JDBC From Visual Basic

tion IDispatch interface, the Java native interface, and the reflection package can create their own Java-Automation Bridge with some effort.

### BACK TO RMI...

So, how does this work with RMI? The answer is amazingly simple. When Java clients want to use remote objects, they use the **Naming.lookup(URL)** method to locate the object and cast it to a predetermined interface. Once this step is completed, the object behaves as though it's a local Java object. In fact, the VM in which it is running does not know that it is remote, only the skeleton and stub code know that. Now, we already know that the Microsoft VM automatically implements OLE Automation access to local Java objects, so there is no obstacle, once you have a local stub, to call remote methods on it.

Let's use a simple RMI remote interface called **Lookup** to illustrate the point. It has three methods that take and return different basic data types.

```
import java.rmi.*;

public interface Lookup extends Remote {
    public String findInfo(String info) throws
    RemoteException;
    public int addInt(int i, int j) throws Remote-
    Exception;
    public double addDbL(double i, double j)
    throws RemoteException;
}
```

We can implement a simple server for this interface as shown in Listing 1.

Now, we have to compile the **LookupServer** class, run **rmic** on it, and then start the **rmiregistry**. Once that is done, we can start the server.

A simple Java client could be used to test this server: Typically, such a client would call **Naming.lookup** to find the remote object reference through the registry; but because the Java-COM marshaling does not support static methods, we have to create a wrapper class. We call this class **RmiClient**:

```
import java.rmi.*;
import java.net.*;
import java.rmi.server.*;

public class RmiClient {
    public Lookup getLookup(String host, String
    server) throws
    RemoteException,NotBoundException,Malformed
    URLException
    {
        String name = "rmi://" + host + "/" + server;
        return (Lookup)Naming.lookup(name);
    }
}
```

The reason we don't catch all of the exceptions, is that if an exception occurs, VB will actually put up a fairly useful message to the user, and the user has the opportunity to handle the exception in VB using the **OnError** mechanism. So, catching the exception actually makes it harder to debug in VB when something goes wrong. Note that the **getLookup()** method returns the result of **Naming.lookup** after it casts it to the remote interface. It is this casting operation that actually loads in the remote stub code. I could

#### Listing 2.

**Class calls RmiClient to get remote reference and then calls a remote method.**

```
// LookupClient.java
import java.rmi.*;
import java.rmi.server.*;

public class LookupClient {
    public static void main(String args[])
    {
        RmiClient rc = new RmiClient();
        try {
            Lookup look_obj = rc.getLookup("localhost", "LookupServer");
            String results = look_obj.findInfo(args[0]);
            if (results == null)
                System.err.println("*** not found ***");
            else
                System.out.println(results);
        } catch (Exception e) {}
    }
}
```

#### Listing 3.

**Now the remote interface can be bound to the VB object.**

```
Private Sub Command1_Click()
    Dim x As Object
    Dim y As Object
    Dim z As String

    Set x = CreateObject("RmiClient")
    Set y = x.getLookup("localhost", "LookupServer")
    If y Is Nothing Then
        MsgBox "can't bind to lookup"
    Else
        z = y.findInfo("HelloWorld")
        MsgBox z
        zz = y.addInt(123, 456)
        MsgBox zz
        zzz = y.addDbL(123.455, 456.789)
        MsgBox zzz
    End If
End Sub
```

## Using RMI and JDBC From Visual Basic

not find a way to write this function generically, so that I wouldn't have to write it over for each new registry lookup of a remote Java class. When I returned it without casting, then VB couldn't bind it to the stub, and I could not find a way to use Java reflection to cast at runtime based on a class name passed as a string.

Now, the test client looks like Listing 2.

This class (Listing 2) just calls RmiClient to get the remote reference and then calls a remote method.

### Listing 4.

#### JDBCWrapper class.

```
import java.sql.*;

public class JdbcWrap
{
    public void loadDriver(String driverName)
    {
        try
        {
            Class.forName(driverName).newInstance();
        }
        catch (Exception e)
        {
            System.out.println("can't find jdbc driver:" + driverName);
        }
    }

    public Connection connect(String url) throws java.sql.SQLException
    {
        Connection connection = null;
        connection = DriverManager.getConnection(url, "", "");
        return connection;
    }

    public static void main(String[] args)
    {
        JdbcWrap jdbcwrap = new JdbcWrap();
        jdbcwrap.loadDriver("com.inven.rcapi.sql.RangerDriver");
        try
        {
            Connection connect = jdbcwrap.connect("jdbc:ranger://localhost:8008");
            if (connect == null) return;
            Statement stmt = connect.createStatement();
            ResultSet results = stmt.executeQuery(args[0]);
            ResultSetMetaData metadata = results.getMetaData();
            int cols = metadata.getColumnCount();
            for(int i=1;i<=cols;i++)
            {
                if (i>1) System.out.print(",");
                System.out.print("[ "+metadata.getColumnLabel(i)+" ]");
            }
            System.out.println();
            while (results.next())
            {
                for(int i=1;i<=cols;i++)
                {
                    System.out.print(results.getString(i) + " ");
                }
                System.out.println("");
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
}
```

The next step is to register RmiClient as an automation server. This is done as follows:

```
javareg /register /class:RmiClient /progid:
RmiClient
```

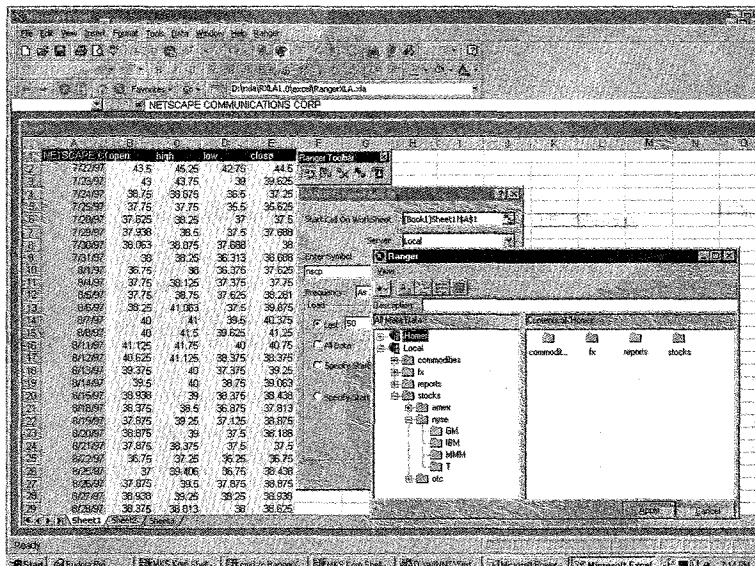
This is all that is needed to make this class known to VB (for production installations, it is recommended that you assign your own CLSID, specify a class path, and create a type-library as well). Now we can start a VB project that will test this. I just created a form with a single button, and in the callback of the button, place the code shown in Listing 3.

Listing 3 instantiates an RmiClient object, gets the Lookup interface by using its getLookup method, and this now binds the remote interface to the VB object. Now, we can simply call the remote methods on this object and pass it VB data types, and all the marshaling is done automatically by the Microsoft VM. When you call MsgBox (which takes a VB string), the VM calls your object's toString() method automatically.

### Listing 5.

#### Visual Basic form.

```
Private Sub Command1_Click()
    Dim jdbcwrap, connect, stmt, results, metadata
    Dim cols, rows, i, col_names, RowData
    Set jdbcwrap = CreateObject("JdbcWrap")
    jdbcwrap.loadDriver ("com.inven.rcapi.sql.Ranger-
    Driver")
    Set connect = jdbcwrap.connect("jdbc:ranger://-
    localhost:8008")
    If Not connect Is Nothing Then
        Load in JDBC result set
        Set stmt = connect.createStatement
        Set results = stmt.executeQuery(Text1.Text)
        Set metadata = results.getMetaData
        cols = metadata.getColumnCount
        rows = results.getRowCount
        Initialize the grid
        MSFlexGrid1.Clear
        MSFlexGrid1.rows = 0
        MSFlexGrid1.cols = cols
        Set column names
        col_names = "date"
        For i = 1 To cols
            col_names = col_names & vbTab & md.get-
            ColumnLabel(i)
        Next
        MSFlexGrid1.AddItem col_names
        Fill in the data
        While results.Next
            RowData = "" & rs.getString(1)
            For i = 2 To cols
                RowData = RowData & vbTab & rs.get-
                String(i)
            Next
            MSFlexGrid1.AddItem (RowData)
        Wend Else
            MsgBox ("can't connect")
        End If
    End Sub
```

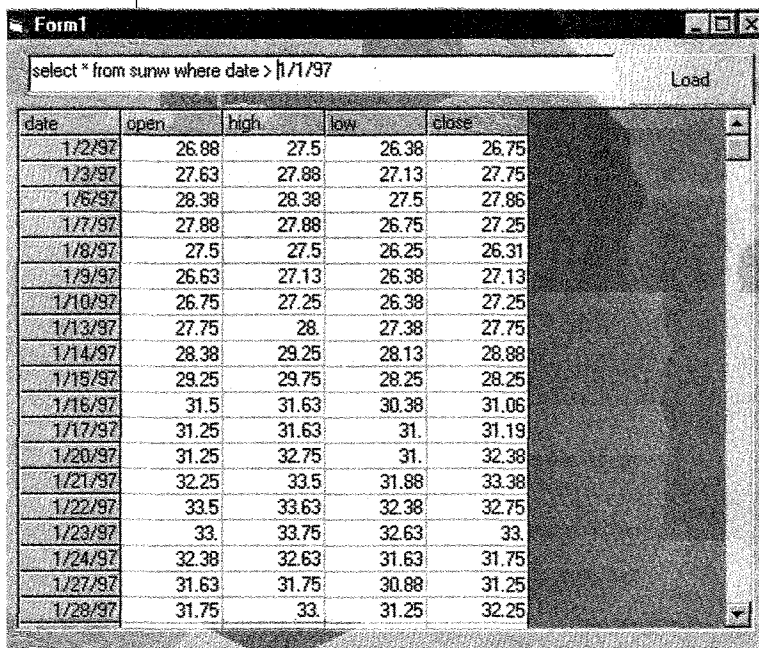


**Figure 1.**  
Inventure's  
RANGER  
Excel add-in.

The most interesting aspect of this is that all the Java code we wrote is 100% Pure Java compiled with any Java compiler. We then need to use **javareg** only on the entry-point classes, i.e., those that we will instantiate from VB directly. All other classes (like the RMI client stubs in this case) are automatically loaded at runtime by the MS JVM and exposed as IDispatch interfaces to the caller.

You can use code like this to write functions in Excel VBA that will pass in ranges of values to a Java RMI call and get a result back. Ranges of values are passed in as VB arrays, which get automatically mapped to and from Java arrays. Figure 1 shows Inventure's RANGER Excel add-in, which is used by several global banks to deliver market

**Figure 2.**  
VB form.



data and analytics to end-users. The add-in is written in VBA on top of a Java API.

## USING JDBC FROM VB

The use of JDBC from VB or any other Automation controller is just as simple as RMI. First, we have to solve the same problem as with RMI, which is to wrap the static methods as class methods. I have created a sample JDBC wrapper class called **JdbcWrap** that does exactly this. It also has a static **main** method, which allows me to test the functionality from Java first, as well as run it under *jview* to verify that the JDBC classes are in my classpath (see Listing 4).

This class (Listing 4) defines two methods **loadDriver(String driverName)** and **connect(String url)**, which are simple wrappers to the static methods. Now, all we need to do is register this class:

```
javareg /register /class:JdbcWrap
/progid:JdbcWrap
```

and we are ready to go. Figure 2 shows the VB form, which has a text box, a button, and a grid. The code for the form is as shown in Listing 5.

Note the fact that we only use our registered **JdbcWrap** class to bootstrap the process. Once we have a **Connection** object, the rest of the API is simply using the JDBC classes directly. Currently, Automation objects created with **javareg** cannot be early-bound unless you generate a type library. This means you have to manipulate them as VB Object's. Objects that are not registered at all are late-bound by definition.

## CONCLUSION

The Microsoft JVM provides an opportunity for developers to use almost any Java system API from COM Automation controllers, and to expose their own APIs in the same way. This capability is extremely powerful, as it allows you to deploy Java-based components to desktop applications that support Microsoft's Automation model.

In particular, the ability to leverage RMI applications directly from any COM Automation controller should help to dissolve the myth that RMI applications are limited by the fact that they are Java-to-Java only. Because COM is language-independent but platform-limited, and RMI is language-limited and platform-independent, the two technologies make a powerful combination. ■

Dan Adler is CTO of Inventure America ([www.inven.com](http://www.inven.com)), New York, NY. He can be contacted at [adan@inven.com](mailto:adan@inven.com).