



Advertisement: Support JavaWorld, click here!



May 1999

HOME

FEATURED
TUTORIALS

COLUMNS

NEWS &
REVIEWS

FORUM

JW
RESOURCES

ABOUT
JW

Java Q&A column debuts!

**The JavaWorld experts answer your pressing Java questions --
every week**

Summary

JavaWorld's new weekly **Java Q&A** column features expert answers to your unbeatable Java questions. In this inaugural installment, our Java experts turn their attention to thread safety, race-condition bugs, running Java programs without the JDK runtime environment, and dealing with inner classes. **Plus:** As a bonus for the **Java Q&A** kickoff, we've included an in-depth look at RMI callbacks. Starting next month, we'll publish two new **Q&As** each and every week. (3,500 words)

Index

- [Thread-safe design](#)
- [Race-condition bugs](#)
- [Run the Java program without the JDK runtime? Yes you can!](#)
- [Inner classes](#)
- [RMI callbacks](#)

Welcome to the first installment of *JavaWorld's* newest resource for Java developers: the **Java Q&A** column. If you're losing sleep over Java, start making a list of the questions that plague you. Beginning in June, you'll be able to tune in weekly for answers from the Java experts at [Random Walk Computing](#) -- Wall Street's leading Java consulting shop. They'll be on hand to impart masterful answers to a variety of your challenging Java questions. And be sure to visit the **Java Q&A** index page for links to previous Qs & As.

Call for Java questions

Do you have a burning Java question?

One whose answer would benefit not just you, but other *JavaWorld* readers? We've got the experts to help. While we can't answer everyone's questions, our Java experts will pick the most interesting and relevant questions for inclusion in upcoming **Java Q&A** columns. To submit your questions to **Java Q&A**, e-mail jawaqa@javaworld.com

Thread-safe design

Q: I would like to know how to write a thread-safe client/server program in Java. How should I approach the threading? What sort of issues should I be aware of?

A: Thread-safe programming is only necessary if you have data that can be modified by more than one thread at a time. In a client/server situation, usually the server has multiple clients. If all those clients do is read data from the server, and if no other programs are modifying that data, then you have nothing to worry about. But if those clients can change shared data on the server, they must not be allowed to conflict with one another. Normally, if two clients try to change shared data, you have to hope that the first client is able to finish with the data before the second client begins to modify it. This situation is called *racing*.

Under these circumstances, the outcome depends on how the underlying scheduler happens to allocate time to the various clients' requests. In the case of Java, one doesn't know what the underlying scheduler's policies are (because they're platform-dependent, or natively implemented), but even with a known scheduling policy, the actual allocation will depend on the availability of various resources (typically I/O) and user inputs; therefore it is never readily predictable.

Thread-safe design replaces racing situations with *choreographed* access to shared data. In Java, thread-safe design gets a lot of support from the underlying language (through *synchronized* blocks) and the standard class library (through the wait/notify mechanism in `Object`).

The simplest way to make a thread-safe design is to use synchronized blocks so that only one client can access the server at any one time. All other clients must wait until the server finishes with that client. Obviously, such a solution doesn't scale well as more clients are added, since clients will have to wait a long time for their turn to access the server.

In fact, if a poorly implemented server entered an infinite loop during a call by one client, then all its other clients would wait forever. This is an example of a general problem called *starvation*, which has to do with situations where a client never finishes its task because one or more other clients have monopolized the resource it needs.

In general, a client shouldn't monopolize a resource that it isn't actively using. When a client has a resource but then blocks while waiting for another resource (typically for I/O), Java will allow a second client to access the server. This situation makes sense, and is often desirable, but must be taken into account in the choreography since the first client may not actually have been done with its task, and the second client may want to access the same resources. The way around this is for the server design to use `wait()` and `notify()` calls within the synchronized blocks, so that the various clients know when it's safe to proceed.

If resource A on the server is independent of resource B, it is reasonable for the server to allow one client to use A at the same time that it allows another client to use B (rather than having the second client wait for the first one to finish with A before it can use B). This situation is safe, as long as A and B are really independent and given that they remain independent after subsequent changes to the server. If A and B aren't truly independent, we can end up with what is called *deadlock*, a situation in which the first client has A and also needs B, while the second client has B and also needs A. This is a special case of starvation, in which the client shares responsibility for its own starvation. Note that Oaks and Wong state in *Java Threads* that "Deadlock ... is the hardest problem to solve in any threaded program."

In summation, thread-safe programming seeks to maximize efficiency by eliminating racing situations, while at the same time avoiding starvation situations.

Resources

- *Java Threads, Second Edition*, Scott Oaks, Henry Wong, et al. (O'Reilly, 1999)
<http://www1.fatbrain.com/asp/bookinfo/bookinfo.asp?theisbn=1565924185>

[Back to index](#)

Race-condition bugs

Q: Why does the following code generate a `NullPointerException`?

```
public class SimpleApplet extends java.applet.Applet {
    java.awt.Image art;
    public void init() {
        art = getImage(getDocumentBase(), getParameter("img"));
    }
    public void paint(java.awt.Graphics g) {
        g.drawImage(art, 0, 0, this);
    }
}
```

A: Because `art` is null.

There is a race-condition (threading) bug in some older applet environments (the JVM and the browser) such that an Applet's `paint()` method can be called before its `init()` method. The workaround (aside from updating your environment) is to add `if (art != null)` before the call to `drawImage()`. Once `init()` is called, the subsequent calls to `paint()` will work properly.

Of course, you'll need to make sure that `SimpleApplet` can find the image file; make sure `getDocumentBase()` returns a valid URL, and that the HTML page is set up correctly (so that `getParameter()` returns a valid image-file name).

[Back to index](#)

Run the Java program without the JDK runtime? Yes you can!

Q: Can I run Java programs without the JDK runtime environment?

A: Yes. Just as you can make standalone executables from C/C++ programs, you can also make standalone executables from Java source code. All that is required in either case is that you have a compiler to convert the source code to binary machine code. For example, the IBM VisualAge Java application development product has a High Performance Java Compiler built in, which compiles Java to native code.

Of course, the resulting binaries can only run on the platform for which they were compiled, so they lose the benefit of "platform independence" that has made Java so successful on the Internet and in distributed multiplatform environments.

Resources

- Download IBM's VisualAge product <http://www.software.ibm.com/ad/vajava/>
- The IBM alphaWorks Web site offers many useful Java-related products <http://www.alphaworks.ibm.com/>

[Back to index](#)

Inner classes

Q: Is it possible to declare a class inside another class and to have the inner class access the private variable of the main class?

A: Certainly! Java 1.1 introduced the concept of inner classes to the Java language. Here's an example of an inner class accessing a private data member of an enclosing class:

```
public class OuterClass {  
    private static String hiding = "You can't see me!";  
    public static class InnerClass {  
        public static void showMe() {  
            System.out.println(hiding);  
        }  
    }  
    public static void main (String args[]) {  
        InnerClass.showMe();  
    }  
}
```

When run, this application will print out "You can't see me!"

Resources

- *The Java Programming Language*, Ken Arnold and James Gosling (Addison Wesley, 1996) The official description of Java's inner classes; see "Appendix D" on the Sun home page for updates to JDK 1.1 <http://java.sun.com/docs/books/javaprogramming/firstedition/1.1Update.html>
- "A look at inner classes," Chuck McManis (*JavaWorld*, October 1997) **Java In-Depth** columnist Chuck McManis provides a more gentle introduction to the topic of inner classes <http://www.javaworld.com/javaworld/jw-10-1997/jw-10-indepth.html>
- The *Java Tutorial* chapter on inner classes <http://java.sun.com/docs/books/tutorial/java/more/nested.html>

[Back to index](#)

RMI callbacks

And here, at last, is our bonus question:

Q: I have a problem with RMI callbacks. Specifically, my server program successfully invokes a client method using callbacks. However, since the client method executes on the server side in this case, how am I supposed to manipulate my frontend AWT components from this method? Or is it not possible using RMI callbacks? I need my frontend AWT components, such as Lists, to be updated with values upon callback. Please explain.

A: We've received quite a few questions regarding RMI this month, from specific issues such as yours to things like, "What's the ding-dong deal with this RMI stuff?" So we thought we'd use your question as a jumping-off point to address RMI as a whole.

Here's the short answer to your question. It isn't correct to say that the client method executes on the server. When using RMI to do callbacks to an application client, the client code doesn't execute on the application server. Only the RMI proxy or stub class executes on the application's server machine. The proxy invocation then arranges for the implementation class for the callback to execute on the virtual machine of the application client. Therefore, since the RMI server logic and the frontend AWT components coexist on the same VM, there's no problem in having an RMI "callback" update information kept by AWT components.

Here's a much longer answer, with a sample program that illustrates what's going on:

It can be difficult when learning RMI to predict what's going on where. The trick is to remember that for any RMI interface there will be two classes that execute: the proxy or stub class runs in the VM of the caller, and the implementation class runs in the VM of the server.

The other thing to keep straight is that it can get confusing to label one VM the "client" and the other the "server" for a particular application. In RMI terminology, any VM that initiates an RMI invocation is the client, while any VM that receives and processes an RMI invocation is the server. I find it easier to think of Java VMs as peers, any of which may act as RMI clients or servers depending on the circumstances. Thus, in RMI there is really no such thing as a "callback" -- a callback is really just an RMI invocation in the other direction! In the context of what many people call client/server applications, it's helpful to designate the program that performs the user interface functions as the frontend or application client, and the program that provides services to that GUI as the backend or application server.

Using this new nomenclature, the question boils down to: Can RMI be used to allow a backend VM to send update information to a frontend VM and have the frontend VM use that information to update the information presented by the GUI? Put this way, the answer is: Yes!

Enough text. Let's look at this in code. Suppose we have a frontend whose mission in life is to display a list of stock symbols and stock prices. The job of the backend would then be to notice when a stock's price changes and update the frontend with this information. Upon receiving such an update, the frontend would replace the old price for that stock with the new price in the AWT-based user interface.

Since you asked about updating the contents of AWT components using application server-side RMI calls, we'll use an AWT `List` component in the GUI. It's not the prettiest way to do it, but it does keep the code short. Apologies in advance for the flicker; this isn't a course on AWT programming!

RMI in a nutshell

Remote Method Invocation (RMI) allows Java programs to call other Java programs that execute in other virtual machines, usually across a network.

The interface to an RMI server is defined by an interface that extends the `java.rmi.Remote`

interface. All methods defined in remote interfaces must be declared to throw the `java.rmi.RemoteException`.

RMI servers begin by replacing the default `SecurityManager` with an instance of `RMISecurityManager`. To advertise available services, an instance of an RMI server object is bound to a name in an RMI registry using the `java.rmi.Naming.rebind` method. The RMI registry is established by running the `rmiregistry` daemon.

An RMI server implementation must extend the `java.rmi.server.UnicastRemoteObject` class and must implement the remote interface for the server. Once this class is defined and compiled, the `rmic` program is used to automatically generate stub and skeleton classes.

RMI clients use the `java.rmi.Naming.Lookup` method to obtain a reference to RMI server objects.

The RMI interfaces

In any well-designed distributed system, the remote interfaces are the most important parts of the system architecture, so let's start with them. The application server provides an interface called `StockInfo`, which defines two method signatures: `register()` and `unregister()`. This is how the frontend client tells the `StockInfo` backend server that the frontend client exists.

A registered `StockInfo` client must implement the `StockUpdate` interface and must pass a reference to an object that implements the `StockUpdate` interface as the sole argument to the `register()` method. Once the `StockInfo` server has a reference to the `StockUpdate` object, the `StockInfo` server implementation may use the reference to talk to the frontend client at will. It is this object that acts as the "callback" object in this architecture. Here is the `StockInfo` interface:

```
package rmistock;import java.rmi.*;
public interface StockInfo extends java.rmi.Remote {
    void register(StockUpdate o) throws RemoteException;
    void unregister(StockUpdate o) throws RemoteException;
}
```

The `StockUpdate` interface looks like this:

```
package rmistock;import java.rmi.*;
public interface StockUpdate extends java.rmi.Remote {
    void update(String symbol, String price) throws RemoteException;
}
```

Once a client has registered with the `StockInfo` server, the server will periodically invoke the `update()` method of the `StockUpdate` object. This means that the application client is itself an RMI server and must be prepared to service RMI invocations at any time after it makes the `StockInfo.register` call.

The StockInfo server

Now that the RMI interfaces are defined, there have to be implementation classes for those interfaces. For the `StockInfo` interface, that class is called `StockInfoImpl`. The class has three responsibilities:

1. To implement the `StockInfo` interface.
2. To take care of the mainline chores of starting and registering an object as a server for the `StockInfo` interface that will name `rmistock.StockInfo`.
3. To implement a background thread that obtains stock symbols and prices, and notifies each registered client of any change in price for a stock symbol. Implementing the `StockInfo` interface is very simple: The `register()` method takes the `StockUpdate` instance passed to it and stores it uniquely in a private `Vector`. The `unregister()` method deletes the `StockUpdate` instance from the `Vector`. Notice that these two methods are *synchronized* to avoid problems that result from the contention of multiple invocations trying to simultaneously access the `Vector` of `StockUpdate` instances.

Here are the implementations of the `StockInfo` interface methods:

```
public synchronized void register(StockUpdate o) throws RemoteException {
    if (!(clients.contains(o))) {
        clients.addElement(o);
        System.out.println("Registered new client " + o);
    }
}
public synchronized void unregister(StockUpdate o) throws RemoteException {
    if (clients.removeElement(o)) {
        System.out.println("Unregistered client " + o);
    } else {
        System.out.println("unregister: client " + o + "wasn't registered.");
    }
}
```

The `StockInfoImpl` server mainline-code creates an RMI registry on TCP port 5001, instantiates a `StockInfoImpl` object, registers that object with the RMI registry, and then creates the background stock-price update thread and starts it running:

```
public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    try {
        LocateRegistry.createRegistry(5001);
        StockInfoImpl sii = new StockInfoImpl();
        Naming.rebind("//:5001/rmistock.StockInfo", sii);
        System.out.println("StockInfoImpl registered and ready");
        Thread updateThread = new Thread(sii, "StockInfoUpdate");
        updateThread.start();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The price update thread implements a simple and not particularly realistic simulation of an equities market. It knows about two stocks: IBM and SUN, and simply increments their prices in lockstep every second from \$5 to \$25 in quarter-point steps, dropping precipitously back to \$5 when the price hits \$25. Every time the price changes, the price update thread iterates through the saved

Vector of `StockUpdate` references. It is here that the callback is performed via the `update()` method:

```
StockUpdate client = (StockUpdate) e.nextElement();
try {
    client.update(symbol, "" + price);
} catch (RemoteException ex) {
    try {
        unregister(client);
    } catch (RemoteException rex) {
    }
}
```

The code that executes here in the backend's VM isn't the actual implementation of the `StockUpdate` interface's `update()` method, but rather a proxy method that serializes the `symbol` and `price` strings back across to the frontend VM. On the frontend side, those serialized strings are turned back into Java objects and then passed to the implementation method, which then executes on the frontend's VM. We'll see what that implementation does when we step through the frontend code below.

Look at the code in the `catch` clause -- this is triggered when a frontend VM terminates. The `update()` method throws a `java.rmi.RemoteException`. This `catch` clause uses the exception as an opportunity to unregister the client so that future iterations of the price update thread don't bother to try to contact that application client again. The `try/catch` around the `unregister` calls is just for form's sake: `unregister` is really supposed to be invoked remotely, but here we use it locally so there's no possibility of a `RemoteException` being raised.

The `StockUpdate` GUI client

The frontend component of this system is implemented in two classes: `StockWatcherGUI` and `StockList`.

`StockWatcherGUI` handles the mainline code for the GUI -- it sets up an AWT frame, and inserts a `StockList` object into the frame. It also sets up the frontend VM as an RMI server and implementation for the `StockUpdate` interface. Since the frontend always passes a reference to the `StockUpdate`-implementing object directly to the backend as an argument to the `register()` method, there is no need to register the `StockUpdate` instance in the RMI registry -- no one ever has to look them up. The mainline code, however, does look up the instance reference bound to the `rmistock.StockInfo` name in the RMI registry for later use. The *implementation* of the `StockUpdate` interface is the `update()` method of this `StockWatcherGUI` class.

```
public synchronized void update(String symbol, String price) throws
RemoteException {
    l.updateStock(symbol, price); // l is an AWT component
}
```

Notice again that it is synchronized -- even though in this system it is unlikely that we'll get more than one `update` call coming in at once, that's not guaranteed to be the interface definition, and a good RMI server must be ready to operate in a multithreaded environment. This method will execute in the frontend's VM, and it has access to the AWT components of the `StockWatcherGUI`. Once we're in this method, it's child's play to update the AWT component. In fact, all this method

does is forward the call to our `StockList` AWT component via the `StockList`'s `updateStock` method. The `StockList` component is a subclass of `java.awt.List`, so a `StockList` object is an AWT component. The `updateStock` method directly modifies the string elements displayed in the list box, formatting them with the incoming stock symbols and prices.

```
public synchronized void updateStock(String symbol, String price) {
    int i = 0;
    for (i = 0; i < stocks.length; i++) {
        if (stocks[i].equals(symbol)) break;
    }
    if (i >= stocks.length) return;
    replaceItem(formatStock(symbol, price), i);
}
```

That's pretty much all there is to it. The complete code, along with build and run scripts for Windows is at [jw-05-javaqa.zip](http://www.javaworld.com/javaworld/jw-05-1999/jw-05-javaqa.zip). You have to unpack the zip file and then run the scripts from the RMI subdirectory. Figure 1 shows a diagram of where each of the objects exist at runtime and how the invocations flow between each of these objects in this RMI-based system.

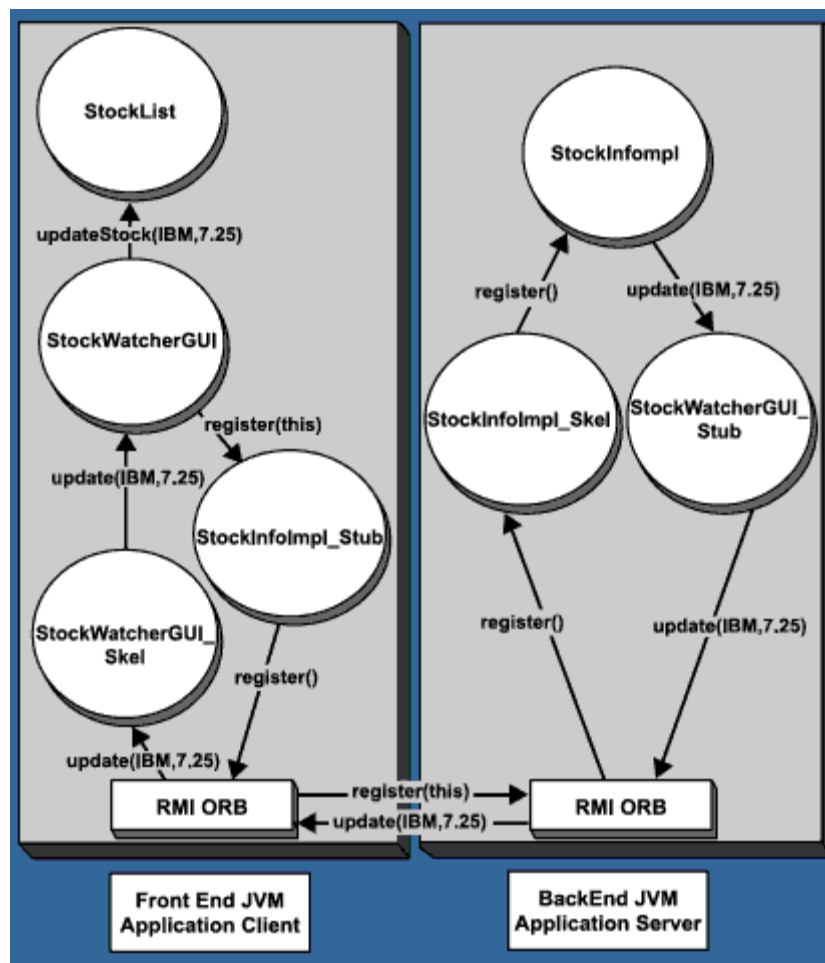


Figure 1. Objects and interactions in the `rmistock` example

Just to prove to yourself that callback code actually executes on the front end, load the class files on two separate machine. On the machine that will be the back end, delete the `StockWatcherGUI.class` and `StockWatcherGUI_Skel.class` files so that the server doesn't have access to them at all. All it needs is the `StockWatcherGUI_Stub.class` file -- this is the proxy class itself that was generated by `rmic`.

Resources

- Download the complete code, along with build and run scripts for Windows. You'll have to unpack the zip file and then run the scripts from the RMI subdirectory [jw-05-javaqa.zip](#)
- "Increase the functionality in your distributed client/server apps," by Michael Shoffner (*JavaWorld*, October 1997). **Step by Step** columnist Michael Shoffner provides a good introduction to RMI <http://www.javaworld.com/javaworld/jw-10-1997/jw-10-step.html>
- "Networking our whiteboard with Java 1.1," by Merlin Hughes (*JavaWorld*, December 1997). **Step by Step** columnist Merlin Hughes delivers another perspective on RMI, including a design pattern that helps it all seem more automatic <http://www.javaworld.com/javaworld/jw-12-1997/jw-12-step.html>
- Sun's RMI Web page offers basic and detailed information on RMI <http://java.sun.com/products/jdk/rmi/>

[Back to index](#) 

About the author

[Random Walk Computing](#) is the largest Java/CORBA consulting boutique in New York, focusing on solutions for the financial enterprise. Known for their leading-edge Java expertise, Random Walk consultants publish and speak about Java in some of the most respected forums in the world. .



Advertisement: Support JavaWorld, click here!

Do you like technology?

[HOME](#) | [FEATURED TUTORIALS](#) | [COLUMNS](#) | [NEWS & REVIEWS](#) | [FORUM](#) | [JW RESOURCES](#) | [ABOUT JW](#) | [FEEDBACK](#)

Copyright © 2005 JavaWorld.com, an IDG company