Java™ *RMI Tutorial*

Prebeta Draft
Revision 1.1

November 1, 1996

Table of Contents

Getting Started 1

This chapter shows you the steps to follow to create a distributed version of the classic Hello World program using Java™ Remote Method Invocation (RMI).

The distributed Hello World example uses an applet to make a remote method call to the server from which it was downloaded to retrieve the message “Hello World!”. When the applet runs, the message is displayed on the client.

To accomplish this, you will:

1. Write The HTML and Java Source Files
2. Compile and Deploy Class Files and HTML Files
3. Start the Remote Object Registry, Server, and Applet

1.1 Write The HTML and Java Source Files

There are four source files for the Hello World server and applet:

1. The Java remote interface.
2. The Java remote object (server) which implements the remote interface.
3. The Java applet that remotely invokes the server’s method.
4. The HTML code for the web page that references the applet.

Because the Java language requires a mapping between the fully qualified package name of a class file and the directory path to that class, before you begin writing Java code you need to decide on package and directory names. (This mapping allows the Java compiler to know the directory in which to find the class files mentioned in a Java program.) For the Hello World program developed in this chapter, the package name is `examples.hello` and the root directory is `$HOME/java/mysrc/examples/hello`.

For example, to create the directory for your source files on Solaris, execute this command:

```
mkdir $HOME/java/mysrc/examples/hello
```

1.1.1 Define a Remote Interface

Remote method invocations can fail in very different ways from local method invocations, due to network related communication problems and server problems. To indicate that it is a remote object, an object implements a remote interface, which has the following characteristics:

- The remote interface must be public. Otherwise, a client will get an error when attempting to load a remote object that implements the remote interface.
- The remote interface extends the interface `java.rmi.Remote`.
- Each method must declare `java.rmi.RemoteException` in its throws clause, in addition to any application-specific exceptions.
- A remote object passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface, not the implementation class.

Here is the interface definition for Hello World. The interface contains just one method, `sayHello`, which returns a string to the caller:

```
package examples.hello;
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```

1.1.2 Write an Implementation Class

To write a remote object, you write a class that implements one or more remote interfaces. The implementation class needs to:

1. Specify the remote interface(s) being implemented.
2. Define the constructor for the remote object.
3. Provide implementations for the methods that can be invoked remotely.
4. Create and install a security manager.
5. Create one or more instances of a remote object.
6. Register at least one of the remote objects with the RMI remote object registry, for bootstrapping purposes.

For example, here is the source for the `HelloImpl.java` file, which contains the code for the Hello World server. The code is followed by an explanation of each of the preceding six steps.

```
package examples.hello;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello
{
    private String name;

    public HelloImpl(String s) throws RemoteException {
        super();
        name = s;
    }

    public String sayHello() throws RemoteException {
        return  "Hello World!";
    }
```

```
        public static void main(String args[])
        {
            // Create and install a security manager
            System.setSecurityManager(new RMISecurityManager());

            try {
                HelloImpl obj = new HelloImpl("HelloServer");
                Naming.rebind("//myhost/HelloServer", obj);
                System.out.println("HelloServer bound in registry");
            } catch (Exception e) {
                System.out.println("HelloImpl err: " + e.getMessage());
                e.printStackTrace();
            }
        }
    }
```

▼ Implement a remote interface

The implementation class for the Hello World example is `HelloImpl`. An implementation class specifies the remote interface(s) it is implementing. It can also optionally indicate the remote server that it is extending, in this example `java.rmi.server.UnicastRemoteObject`. Here is the `HelloImpl` class declaration:

```
public class HelloImpl
    implements Hello
    extends java.rmi.server.UnicastRemoteObject
```

Extending `UnicastRemoteObject` indicates that the `HelloImpl` class is used to create a single (non-replicated) remote object that uses RMI's default sockets-based transport for communication. If you choose to extend a remote object from a non-remote class, you need to explicitly export the remote object by calling the method `UnicastRemoteObject.exportObject`.

Define the constructor for the remote object

The constructor for a remote class is no different than the constructor for a non-remote class: it initializes the variables of each newly created instance of the class.

Here is the constructor for the `HelloImpl` class, which initializes the private string variable `name` with the name of the remote object:

```
private String name;
public HelloImpl(String s) throws java.rmi.RemoteException {
    super();
    name = s;
}
```

Note the following:

- The `super` method call invokes the no-arg constructor of `java.rmi.server.UnicastRemoteObject`, which "exports" the remote object by listening for incoming calls to the remote object on an anonymous port.
- The constructor must throw `java.rmi.RemoteException`, because RMI's attempt to export a remote object during construction may fail if communication resources are not available.

Although the call to the `super` no-arg constructor occurs by default if omitted, it is included in this example to make clear the fact that Java constructs the superclass before the class.

Provide an implementation for each remote method

The implementation class for a remote object contains the code that implements each of the remote methods specified in the remote interface.

For example, here is the implementation for the `sayHello` method, which returns the string `Hello World!` to the caller.

```
public String sayHello() throws RemoteException {
    return  "Hello World!";
}
```

Arguments to or return values from remote methods can be of any Java type, including objects, as long as those objects implement the interface `java.io.Serializable`. Most of the core Java classes in `java.lang` and `java.util` implement the `Serializable` interface.

- Local objects are passed by copy, and only the non-static and non-transient fields are copied by default.
- Remote objects are passed by reference. A reference to a remote object is actually a reference to a stub, which is a client-side proxy for the remote object. Stubs are described fully in Section 1.2.2, “Generate Stubs and Skeletons”.

Note – A class can define methods not specified in the remote interface, but those methods can only be invoked within the virtual machine running the service and cannot be invoked remotely.

▼ Create and install a security manager

The `main` method of the service first needs to create and install a security manager, either the `RMISecurityManager` or one that you have defined yourself. For example:

```
System.setSecurityManager(new RMISecurityManager());
```

A security manager needs to be running so that it can guarantee that the classes loaded do not perform “sensitive” operations. If no security manager is specified, no class loading for RMI classes, local or otherwise, is allowed.

▼ Create one or more instances of a remote object

The `main` method of the service needs to create one or more instances of the remote object which provides the service. For example:

```
HelloImpl obj = new HelloImpl("HelloServer");
```

The constructor exports the remote object, which means that once created, the remote object is ready to begin listening for incoming calls.

▼ Register a remote object

For a caller (client, peer, or applet) to be able to invoke a method on a remote object, that caller must first obtain a reference to the remote object. Most of the time the reference will be obtained as a parameter to or a return value from another remote method call.

For bootstrapping, the RMI system also provides a URL-based registry that allows you to bind a URL of the form `//host/objectname` to the remote object, where `objectname` is a simple string name. Once a remote object is registered on the server, callers can look up the object by name, obtain a remote object reference, and then remotely invoke methods on the object.

For example, the following code binds the URL of the remote object named `HelloServer` to a reference for the remote object:

```
Naming.rebind("//myhost/HelloServer", obj);
```

Note the following about the arguments to the call:

- The host defaults to the current host if omitted from the URL, and no protocol needs to be specified in the URL.
- The RMI runtime substitutes a reference to the remote object's stub for the actual remote object reference specified by the `obj` argument. Remote implementation objects like instances of `HelloImpl` never leave the VM where they are created, so when a client performs a lookup in a server's remote object registry, a reference to the stub is returned.
- Optionally a port number can be supplied in the URL, for example //myhost:1234/HelloServer. The port defaults to 1099. It is necessary to specify the port number only if a server creates a registry on a port other than the default 1099.

Note – For security reasons, an application can bind or unbind only in the registry running on the same host. This prevents a client from removing or overwriting any of the entries in a server's remote registry. A lookup, however, can be done from any host.

1.1.3 Write an Applet that Uses the Remote Service

The applet part of the distributed Hello World example remotely invokes the HelloServer's `sayHello` method in order to get the string "Hello World!", which is displayed when the applet runs. Here is the code for the applet:

```
package examples.hello;

import java.awt.*;
import java.rmi.*;

public class HelloApplet extends java.applet.Applet {
    String message = "";
    public void init() {
        try {
            Hello obj = (Hello)Naming.lookup("//" +
                getCodeBase().getHost() + "/HelloServer");
            message = obj.sayHello();
        } catch (Exception e) {
            System.out.println("HelloApplet exception: " +
                e.getMessage());
            e.printStackTrace();
        }
    }
    public void paint(Graphics g) {
        g.drawString(message, 25, 50);
    }
}
```

1. The applet first gets a reference to the "HelloServer" from the server's registry, constructing the URL by using the `getCodeBase` method in conjunction with the `getHost` method.

2. The applet remotely invokes the `sayHello` method of the HelloServer remote object and stores the return value from the call (the string "Hello World!") in a variable named `message`.

3. The applet invokes the `paint` method to draw the applet on the display, causing the string "Hello World!" to be displayed.

Note – The constructed URL must include the host. Otherwise, the applet's lookup will default to the client, and the `AppletSecurityManager` will throw an exception since the applet cannot access the local system, but is instead limited to communicating only with applet host.

1.1.4 Write the Web Page that Contains the Applet

Here is the HTML code for the web page that references the Hello World applet:

```
<HTML>
<title>Hello World</title>
<center> <h1>Hello World</h1> </center>

The message from the HelloServer is:
<p>
<applet codebase="../.."
    code="examples.hello.HelloApplet"
    width=500 height=120>
</applet>
</HTML>
```

Note the following:

- There needs to be an HTTP server running on the machine from which you want to download classes. The applet's `codebase` attribute indicates the URL, as shown here:

  ```
  codebase="../.."
  ```

  The codebase in this example specifies a directory two levels above the directory from which the web page was itself loaded. Using this kind of relative path is usually a good idea.

- The applet's `code` attribute specifies the fully package qualified name of the applet, in this example `examples.hello.HelloApplet`:

  ```
  code="examples.hello.HelloApplet"
  ```

1.2 Compile and Deploy Class Files and HTML Files

The source code for the Hello World example is now complete and the `$HOME/java/mysrc/hello` directory has four files:

- `Hello.java`, which contains the source code for the Hello remote interface.
- `HelloImpl.java`, which is the source code for the HelloImpl remote object implementation, the server for the Hello World applet.
- `HelloApplet.java`, which is the source code for the applet.
- `index.html`, which is the web page that references the Hello World applet.

In this section, you compile the `.java` source files to create `.class` files. You then run the `rmic` compiler to create stubs and skeletons. A stub is a client-side proxy for a remote object which forwards RMI calls to the server-side skeleton, which in turn forwards the call to the actual remote object implementation.

When you use the `javac` and `rmic` compilers, you must specify where the resulting class files should reside. For applets, all files should be in the applet's codebase directory. In this chapter, this is `$HOME/public_html/codebase`.

Note – Some Web servers allow accessing a user's public_html diretory via an HTTP URL constructed as "http://host/~username/". If your Web server does not support this convention, you may use a file URL of the form "file://home/username/public_html".

1.2.1 Compile the Java Source Files

Make sure that the deployment directory `$HOME/public_html/codebase` and the development directory `$HOME/java/mysrc/examples/hello` are each visible via the local `CLASSPATH` on the development machine.

To compile the Java source files, run the `javac` command as follows:

```
javac -d $HOME/public_html/codebase
   Hello.java HelloImpl.java HelloApplet.java
```

This command creates the directory `examples/hello` (if it does not already exist) in the directory `$HOME/public_html/codebase`. The command then writes to that directory the files `Hello.class`, `HelloImpl.class`, and `HelloApplet.class`. These are the remote interface, the server, and the applet respectively.

1.2.2 Generate Stubs and Skeletons

To create stub and skeleton files, run the `rmic` compiler on the names of compiled class files that contain remote object implementations. `rmic` takes one or more class names as input and produces as output class files of the form *`myImpl`*`_Skel.class` and *`myImpl`*`_Stub.class`.

For example, to create the stub and skeleton for the HelloImpl remote object implementation, run `rmic` like this:

```
rmic -d $HOME/public_html/codebase examples.hello.HelloImpl
```

The `-d` option indicates the root directory in which to place the compiled stub and skeleton files. So the preceding command creates the following files in the directory `$HOME/public_html/codebase/examples/hello`:

- `HelloImpl_Stub.class`
- `HelloImpl_Skel.class`

Note that the generated stub implements exactly the same set of remote interfaces as the remote object itself. This means that a client can use the Java language's built-in operators for casting and type checking. It also means that Java remote objects support true object-oriented polymorphism.

1.2.3 Move the HTML File to the Deployment Directory

To make the web page that references the applet visible to clients, the `index.html` file must be moved from the development directory to the `codebase` directory. For example:

```
mv $HOME/java/mysrc/examples/hello/index.html
    $HOME/public_html/codebase/examples/hello
```

1.2.4 Set Paths for Runtime

Make sure that the `$HOME/public_html/codebase` directory is available via the server's local `CLASSPATH` when you run the HelloImpl server.

1.3 Start the Remote Object Registry, Server, and Applet

1.3.1 Start the RMI Bootstrap Registry

The RMI registry is a simple server-side bootstrap name server that allows remote clients to get a reference to a remote object. It is typically used only to locate the first remote object an application needs to talk to. That object in turn will provide application specific support for finding other objects.

To start the registry on the server, execute the `rmiregistry` command. This command produces no output and is typically run in the background. For example, on Windows 95 or Windows NT:

```
start rmiregistry
```

(use `javaw` if start is not available)

And on Solaris:

```
rmiregistry &
```

The registry by default runs on port 1099. To start the registry on a different port, specify the port number in the command. For example, to start the registry on port 2001 on Windows NT:

```
start rmiregistry 2001
```

If the registry is running on a port other than the default, you need to specify the port number in the URL-based methods of the `java.rmi.Naming` class when making calls to the registry. For example, if the registry is running on port 2001 in the Hello World example, here is the call required to bind the URL of the HelloServer to the remote object reference:

```
Naming.rebind("//myhost:2001/HelloServer", obj);
```

Similarly, the URL stored on the web page needs to specify the non-default port number, or else the applet's attempt to lookup the server in the registry will fail:

```
<PARAM name="url" value="//myhost:2001/HelloServer">
```

Note – You must stop and restart the registry any time you modify a remote interface or use modified/additional remote interfaces in a remote object implementation. Otherwise, the class bound in the registry will not match the modified class.

1.3.2 Start the Server

When starting the server, the `java.rmi.server.codebase` property must be specified, so that references to the remote objects created by the server can include the URL from which the stub class can be dynamically downloaded to the client.

The following command shows how to start the HelloImpl server, specifying this property:

```
java -Djava.rmi.server.codebase=http://myhost/~myusrname/codebase/
    examples.hello.HelloImpl &
```

Note – The trailing / in the codebase URL must be specified.

A stub class is dynamically loaded into a client's virtual machine only when the class is not already available locally.

1.3.3 Run the Applet

Note – At Prebeta, the applet must be run in theJDK1.0.2 appletviewer or the special RMI version of HotJava.

Once the registry and server are running, the applet can be run. An applet is run by loading its web page into a browser or appletviewer, as shown here:

```
appletviewer
    http://myhost/~myusrname/codebase/examples/hello/index.html &
```

After running the appletviewer, you will see output similar to the following on your display: