Getting Started Using Java™ RMI

This tutorial shows you the steps to follow to create a distributed version of the classic Hello World program using Java™ Remote Method Invocation (Java RMI). While you work through this example, you will probably come up with a number of related questions. You are encouraged to look for answers in the Java RMI FAQ and Preservation of Mail Lists and Documentation External to the River Project.

The distributed Hello World example uses a simple client to make a remote method invocation to a server which may be running on a remote host. The client receives the "Hello, world!" message from the server.

This tutorial has the following steps:

- Define the remote interface
- Implement the server
- Implement the client
- Compile the source files
- Start the Java RMI registry, server, and client

The files needed for this tutorial are:

- `Hello.java` - a remote interface
- `Server.java` - a remote object implementation that implements the remote interface
- `Client.java` - a simple client that invokes a method of the remote interface

Note: For the remainder of this tutorial, the terms "remote object implementation" and "implementation class" are used interchangeably to refer to the class `example.hello.Server`, which implements a remote interface.

Define the remote interface

A remote object is an instance of a class that implements a *remote interface*. A remote interface extends the interface `java.rmi.Remote` and declares a set of *remote methods*. Each *remote method* must declare `java.rmi.RemoteException` (or a superclass of `RemoteException`) in its `throws` clause, in addition to any application-specific exceptions.

Here is the interface definition for the remote interface used in this example, `example.hello.Hello`. It declares just one method, `sayHello`, which returns a string to the caller:

```
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Remote method invocations can fail in many additional ways compared to local method invocations (such as network-related communication problems and server problems), and remote methods will report such failures by throwing a `java.rmi.RemoteException`.

Implement the server

A "server" class, in this context, is the class which has a `main` method that creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name in a Java RMI *registry*. The class that contains this `main` method could be the implementation class itself, or another class entirely.

In this example, the `main` method for the server is defined in the class `Server` which also implements the remote interface `Hello`. The server's `main` method does the following:

- Create and export a remote object
- Register the remote object with a Java RMI registry

Here is the source code for the class `Server`. Descriptions for writing this server class follow the source code:

```
package example.hello;
```

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {

    public Server() {}

    public String sayHello() {
        return "Hello, world!";
    }

    public static void main(String args[]) {

        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

The implementation class `Server` implements the remote interface `Hello`, providing an implementation for the remote method `sayHello`. The method `sayHello` does not need to declare that it throws any exception because the method implementation itself does not throw `RemoteException` nor does it throw any other checked exceptions.

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 export a remote object

The `main` method of the server needs to create the remote object that provides the service. Additionally, the remote object must be *exported* to the Java RMI runtime so that it may receive incoming remote calls. This can be done as follows:

```
Server obj = new Server();
Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
```

The static method `UnicastRemoteObject.exportObject` exports the supplied remote object to receive incoming remote method invocations on an anonymous TCP port and returns the stub for the remote object to pass to clients. As a result of the `exportObject` call, the runtime may begin to listen on a new server socket or may use a shared server socket to accept incoming remote calls for the remote object. The returned stub implements the same set of remote interfaces as the remote object's class and contains the host name and port over which the remote object can be contacted.

Note: As of the J2SE 5.0 release, stub classes for remote objects no longer need to be pregenerated using the `rmic` stub compiler, unless the remote object needs to support clients running in pre-5.0 VMs. If your application needs to support such clients, you will need to generate stub classes for the remote objects used in the application and deploy those stub classes for clients to download. For details on how to generate stub classes, see the tools documentation for `rmic` [Solaris, Windows]. For details on how to deploy your application along with pregenerated stub classes, see the codebase tutorial.

Register the remote object with a Java RMI registry

For a caller (client, peer, or applet) to be able to invoke a method on a remote object, that caller must first obtain a stub for the remote object. For bootstrapping, Java RMI provides a registry API for applications to bind a name to a remote object's stub and for clients to look up remote objects by name in order to obtain their stubs.

A Java RMI registry is a simplified name service that allows clients to get a reference (a stub) to a remote object. In general, a registry is used (if at all) only to locate the first remote object a client needs to use. Then, typically, that first object would in turn provide application-specific support for finding other objects. For example, the reference can be obtained as a parameter to, or a return value from, another remote method call. For a discussion on how this works, please take a look at *Applying the Factory Pattern to Java RMI*.

Once a remote object is registered on the server, callers can look up the object by name, obtain a remote object reference, and then invoke remote methods on the object.

The following code in the server obtains a stub for a registry on the local host and default registry port and then uses the registry stub to bind the name "Hello" to the remote object's stub in that registry:

```
Registry registry = LocateRegistry.getRegistry();
registry.bind("Hello", stub);
```

The static method `LocateRegistry.getRegistry` that takes no arguments returns a stub that implements the remote interface `java.rmi.registry.Registry` and sends invocations to the registry on server's local host on the default registry port of `1099`. The `bind` method is then invoked on the `registry` stub in order to bind the remote object's stub to the name "`Hello`" in the registry.

Note: The call to `LocateRegistry.getRegistry` simply returns an appropriate stub for a registry. The call does not check to see if a registry is actually running. If no registry is running on TCP port 1099 of the local host when the `bind` method is invoked, the server will fail with a `RemoteException`.

Implement the client

The client program obtains a stub for the registry on the server's host, looks up the remote object's stub by name in the registry, and then invokes the `sayHello` method on the remote object using the stub.

Here is the source code for the client:

```
package example.hello;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    private Client() {}

    public static void main(String[] args) {

        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

This client first obtains the stub for the registry by invoking the static `LocateRegistry.getRegistry` method with the hostname specified on the command line. If no hostname is specified, then `null` is used as the hostname indicating that the local host address should be used.

Next, the client invokes the remote method `lookup` on the registry stub to obtain the stub for the remote object from the server's registry.

Finally, the client invokes the `sayHello` method on the remote object's stub, which causes the following actions to happen:

- The client-side runtime opens a connection to the server using the host and port information in the remote object's stub and then serializes the call data.
- The server-side runtime accepts the incoming call, dispatches the call to the remote object, and serializes the result (the reply string "Hello, world!") to the client.
- The client-side runtime receives, deserializes, and returns the result to the caller.

The response message returned from the remote invocation on the remote object is then printed to `System.out`.

Compile the source files

The source files for this example can be compiled as follows:

```
javac -d destDir Hello.java Server.java Client.java
```

where ***destDir*** is the destination directory to put the class files in.

Note: If the server needs to support clients running on pre-5.0 VMs, then a stub class for the remote object implementation class needs to be pregenerated using the `rmic` compiler, and that stub class needs to be made available for clients to download. See the codebase tutorial for more details.

Start the Java RMI registry, server, and client

To run this example, you will need to do the following:

- Start the Java RMI registry
- Start the server
- Run the client

Start the Java RMI registry

To start the registry, run the `rmiregistry` command on the server's host. This command produces no output (when successful) and is typically run in the background. For more information, see the tools documentation for `rmiregistry` [Solaris, Windows].

For example, on the Solaris(tm) Operating System:

```
rmiregistry &
```

Or, on Windows platforms:

```
start rmiregistry
```

By default, the registry runs on TCP port 1099. To start a registry on a different port, specify the port number from the command line. For example, to start the registry on port 2001 on a Windows platform:

```
start rmiregistry 2001
```

If the registry will be running on a port other than 1099, you'll need to specify the port number in the calls to `LocateRegistry.getRegistry` in the `Server` and `Client` classes. For example, if the registry is running on port 2001 in this example, the call to `getRegistry` in the server would be:

```
Registry registry = LocateRegistry.getRegistry(2001);
```

Start the server

To start the server, run the `Server` class using the `java` command as follows:

On the Solaris Operating System:

```
java -classpath classDir -Djava.rmi.server.codebase=file:classDir/ example.hello.Server &
```

On Windows platforms:

```
start java -classpath classDir -Djava.rmi.server.codebase=file:classDir/ example.hello.Server
```

where ***classDir*** is the root directory of the class file tree (see ***destDir*** in the section "Compiling the source files"). Setting the `java.rmi.server.codebase` system property ensures that the registry can load the remote interface definition (note that the trailing slash is important); for more information about using this property, see the codebase tutorial.

The output from the server should look like this:

```
Server ready
```

The server remains running until the process is terminated by the user (typically by killing the process).

Run the client

Once the server is ready, the client can be run as follows:

```
java  -classpath classDir example.hello.Client
```

where ***classDir*** is the root directory of the class file tree (see ***destDir*** in the section "Compiling the source files").

The output from the client is the following message:

```
response: Hello, world!
```