[an error occurred while processing this directive]

Hello world!

This is an abbreviated version of Sun's tutorial titled *Getting Started Using RMI*. It is a distributed version of Hello World using Java RMI. In the distributed Hello World example, a client makes a remote method call to the server, to retrieve the message "Hello world!". When the client runs, "Hello world!" is output to the client's *System.out* stream. To accomplish this, we need to:

1. **Write** the Java sources
2. **Compile** and deploy class files
3. **Start** the RMI registry, server, and client

Note: The terms "remote object implementation", "object implementation," and "implementation" may be used interchangeably to refer to the class, examples.hello.HelloImpl, which implements Hello, an extension of Remote.

Write the Java Source

Define the functions of the remote class as a Java interface

Your remote interface declares each of the methods that you want to call remotely. Remote interfaces have the following characteristics:

- It extends the java.rmi.Remote interface.
- Each method must declare java.rmi.RemoteException (or a superclass of RemoteException) in its throws clause.

```
import java.rmi.*;

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

Implementation the Remote server class

At a minimum, a remote object class must:

- Implement at least one Remote interface.
- Define a constructor for the remote object.

A "server" class in this context, is the class which has a **`main`** method that:

- creates an instance of the remote object implementation, and
- binds that instance to a name in the `rmiregistry`.

The class that contains this `main` method could be the implementation class itself, or **another** class.

In this example, the `main` method is part of `examples.hello.HelloImpl`. The server program needs to:

- Instantiate the remote object.
- **Register** the remote object with the rmiregistry.

An explanation of each of these steps follows the source for `HelloImpl.java:`

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello
{
    public HelloImpl() throws RemoteException {}

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

    public static void main(String args[])
    {
        try
        {
            HelloImpl obj = new HelloImpl();
            // Bind this object instance to the name "HelloServer"
            Naming.rebind("HelloServer", obj);
        }
        catch (Exception e)
        {
            System.out.println("HelloImpl err: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Define the constructor for the remote object

Your remote object instance is **exported**: **make it available to accept incoming remote method invocations by listening for incoming calls to the remote object on an anonymous port**.

- When you extend `java.rmi.server.UnicastRemoteObject`, your class is automatically exported upon creation.

  Because the object export could potentially throw a `java.rmi.RemoteException`, you ***must*** define a constructor that throws a `RemoteException`, even if the constructor does nothing else. The no-argument constructor for the superclass, UnicastRemoteObject, declares the exception RemoteException in its throws clause, so your constructor must also declare that it can throw RemoteException. A RemoteException can occur during construction if the attempt to export the object fails--due to, for example, communication resources being unavailable or the appropriate stub class not being found.

Instantiate a remote object

The `main` method of the server creates an instance of the remote object implementation:

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

The constructor exports the remote object: Once created, the remote object is ready to accept incoming calls.

Register the remote object

For a client to invoke a method on a remote object, it must get a reference to the remote object.

The RMI system provides a remote object registry that allows you to bind a URL-formatted name of the form "`//host/objectname`" to the remote object, where `objectname` is a simple string name.

The RMI registry is a simple server-side name server that allows remote clients to get a reference to a remote object.

It typically is used to locate only the **first** remote object an RMI client needs to talk to. Then, that first object in turn, provides **application-specific** support getting references for other objects.

For example, the reference can be obtained as a parameter to, or a return value from, another remote method call.

Once a remote object is registered in the rmiregistry, clients can:

- obtain a remote object reference (e.g., by looking it up in the rmiregistry)
- remotely invoke methods on the object.

For example, the following code binds the name "HelloServer" to a reference for the remote object:

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

Write a client program that uses the remote service

The client part of the distributed Hello World example remotely invokes the `sayHello` method in order to get the string "Hello world!", which is output when the client runs. Here is the code for the client:

```
import java.rmi.RMISecurityManager;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloClient
{
    public static void main(String arg[])
    {
        String message = "blank";

        // I download server's stubs ==> must set a SecurityManager
        System.setSecurityManager(new RMISecurityManager());

        try
        {
           Hello obj = (Hello) Naming.lookup( "//" +
                "lysander.cs.ucsb.edu" +
                "/HelloServer");         //objectname in registry
           System.out.println(obj.sayHello());
        }
        catch (Exception e)
        {
           System.out.println("HelloClient exception: " + e.getMessage());
           e.printStackTrace();
        }
    }
}
```

- Set the security manager, so that the client can download the stub code.
- Get a reference to the remote object implementation (advertised as "HelloServer") from the server host's `rmiregistry`.
- Invoke the remote `sayHello` method on the server's remote object

The constructed URL-string that is passed as a parameter to the `Naming.lookup` *method must include the server's hostname.*

Compile & Deploy Class Files

The source code for the Hello World example is now complete and the `$HOME/mysrc/examples/hello` directory has 3 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 client.
- `HelloClient.java`, which is the source code for the client.

When you use the `javac` and `rmic` compilers, specify where the resulting class files should reside.

Compile the Java source files

Make sure that the deployment directory `$HOME/myclasses` and the development directory `$HOME/mysrc/examples/hello` are each accessible through the local `CLASSPATH` on the development machine, before attempting to compile.

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

```
javac -d $HOME/myclasses
        Hello.java HelloImpl.java HelloClient.java
```

1. It creates the directory `examples/hello` (if it does not exist) in `$HOME/myclasses`.
2. It then writes to that directory the files `Hello.class`, `HelloImpl.class`, and `HelloClient.class`.

For an explanation of `javac` options, please refer to the Solaris `javac` manual page or the Win32 `javac` manual page.

Use `rmic` to generate skeletons and/or stubs

The `rmic` command takes one or more class names as an argument and produces 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 HelloImpl
```

1. `HelloImpl_Stub.class`
2. `HelloImpl_Skel.class`

Start the RMI registry, server, & client

Start the RMI registry

Note: *Before you start the rmiregistry, you must make sure that the shell or window in which you will run the* `registry`*, either has no CLASSPATH set or has a CLASSPATH that does not include the path to any classes that you want downloaded to your client, including the stubs for your remote object implementation classes.*

If you start the `rmiregistry`*, and it* can *find your stub classes in its CLASSPATH, it will ignore the server's* `java.rmi.server.codebase` *property, and as a result, your client(s) will not be able to download the stub*

code for your remote object.

- To start the registry on the server, execute the `rmiregistry` command.
- This command produces no output and is typically run in the background.
- For more on the `rmiregistry`, please refer to the Solaris `rmiregistry` manual page or the Win32 `rmiregistry` manual page.

For example, on Solaris:

```
rmiregistry &
```

For example, on Windows 95 or Windows NT:

```
start rmiregistry
```

Start the server

1. For Solaris:

```
java  HelloImpl &
```

2. For Windows:

```
java  HelloImpl
```

Run the client

Once the registry and server are running, the client can be run as follows:

```
java HelloClient
```

After running the client, you will see "Hello world!". [an error occurred while processing this directive]