

Tutorial: Getting Started Using RMI-IIOP

This tutorial shows you the steps to follow to create a distributed version of the classic "Hello World" program using Java™ Remote Method Invocation (RMI) over Internet Inter-ORB Protocol (IIOP). RMI-IIOP adds CORBA (Common Object Request Broker Architecture) capability to Java RMI, providing standards-based interoperability and connectivity to many other programming languages and platforms. RMI-IIOP enables distributed Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard IIOP defined by the Object Management Group. Runtime components include a Java ORB for distributed computing using IIOP communication.

[RMI-IIOP](#) is for Java programmers who want to program to the RMI interfaces, but use IIOP as the underlying transport. RMI-IIOP provides interoperability with other CORBA objects implemented in various languages - but only if all the remote interfaces are originally defined as Java RMI interfaces. It is of particular interest to programmers using Enterprise JavaBeans (EJB), since the remote object model for EJBs is RMI-based.

Other options for creating distributed applications are:

- [Java™ Interface Definition Language \(IDL\)](#)

Java IDL is for CORBA programmers who want to program in the Java programming language based on interfaces defined in CORBA Interface Definition Language (IDL). This is "business as usual" CORBA programming, supporting Java in exactly the same way as other languages like C++ or COBOL.

- [Java™ Remote Method Invocation \(RMI\)](#).

The Java RMI system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM. RMI provides for remote communication between programs written in the Java programming language via the Java Remote Method Protocol (JRMP).

Tutorial: The Hello World Application

The distributed Hello World example uses a client application to make a remote method call via IIOP to a server running on the host from which the client was downloaded. When the client runs, "Hello from MARS!" is displayed.

This tutorial is organized as follows:

1. [The steps to write the source files](#)
2. [The steps to compile the example](#)
3. [The steps to run the example](#)



Each step in the tutorial is indicated by this symbol.

Write the Source Files

There are three tasks to complete in this section:

1. [Define the functions of the remote class as an interface written in the Java programming language](#)
2. [Write the implementation class](#)
3. [Write the server class](#)
4. [Write a client program that uses the remote service](#)

The source files created in this tutorial are:

- [HelloInterface.java](#) - a remote interface
- [HelloImpl.java](#) - a remote object implementation that implements HelloInterface
- [HelloServer.java](#) - an RMI server that creates an instance of the remote object implementation and binds that instance to a name in the Naming Service
- [HelloClient.java](#) - a client application that invokes the remote method, sayHello()

Define the functions of the remote class as an interface written in the Java programming language

In the Java programming language, a remote object is an instance of a class that implements a `Remote` interface. Your remote interface will declare each of the methods that you would like to call from other machines. Remote interfaces have the following characteristics:

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

For this example, create all of the source files in the same directory, for example, `$HOME/mysrc/helloWorld`.



Create the file `HelloInterface.java`. The following code is the interface definition for the remote interface, `HelloInterface`, which contains just one method, `sayHello`:

```
//HelloInterface.java  
  
import java.rmi.Remote;  
  
public interface HelloInterface extends java.rmi.Remote {  
    public void sayHello( String from ) throws java.rmi.RemoteException;  
}
```

Because remote method invocations can fail in very different ways from local method invocations (due to network-related communication problems and server problems), remote methods will report communication failures by throwing a `java.rmi.RemoteException`. If you want more information on failure and recovery in distributed systems, you may wish to read [A Note on Distributed Computing](#).

Write The Implementation Class

At a minimum, a remote object implementation class, `HelloImpl.java` must:

- [Declare that it implements at least one remote interface](#)
- [Define the constructor for the remote object](#)
- [Provide implementations for the methods that can be invoked remotely](#)



Create the file `HelloImpl.java`. The code for this file follows. An explanation of each of the preceding steps follows the source code:

```
//HelloImpl.java

import javax.rmi.PortableRemoteObject;

public class HelloImpl extends PortableRemoteObject implements HelloInterface
{
    public HelloImpl() throws java.rmi.RemoteException {
        super();
        // invoke rmi linking and remote object initialization
    }

    public void sayHello( String from ) throws java.rmi.RemoteException {
        System.out.println( "Hello from " + from + "!!" );
        System.out.flush();
    }
}
```

Implement a remote interface

In the Java programming language, when a class declares that it implements an interface, a contract is formed between the class and the compiler. By entering into this contract, the class is promising that it will provide method bodies, or definitions, for each of the method signatures declared in that interface. Interface methods are implicitly `public` and `abstract`, so if the implementation class doesn't fulfill its contract, it becomes *by definition* an `abstract` class, and the compiler will point out this fact if the class was not declared `abstract`.

The implementation class in this example is `HelloImpl`. The implementation class declares which remote interface(s) it is implementing. Here is the `HelloImpl` class declaration:

```
public class HelloImpl extends PortableRemoteObject
    implements HelloInterface{
```

As a convenience, the implementation class can extend a remote class, which in this example is `javax.rmi.PortableRemoteObject`. By extending `PortableRemoteObject`, the `HelloImpl` class can be used to create a remote object that uses IIOP-based transport for communication.

Define the constructor for the remote object

The constructor for a remote class provides the same functionality as the constructor for a non-remote class: it initializes the variables of each newly created instance of the class, and returns an instance of the class to the program which called the constructor.

In addition, the remote object instance will need to be "exported". Exporting a remote object makes it available to accept incoming remote method requests, by listening for incoming calls to the remote object on an anonymous port. When you extend `javax.rmi.PortableRemoteObject`, your class will be exported automatically 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. If you forget the constructor, `javac` will produce the following error message:

```
HelloImpl.java:3: unreported exception java.rmi.RemoteException; must be caught or declared to be thrown.
```

```
public class HelloImpl extends PortableRemoteObject implements
HelloInterface{
    ^
    1 error
```

To review: *The implementation class for a remote object needs to:*

- *Implement a remote interface*
- *Export the object so that it can accept incoming remote method calls*
- *Declare its constructor(s) to throw at least a `java.rmi.RemoteException`*

Here is the constructor for the `HelloImpl` class:

```
public HelloImpl() throws java.rmi.RemoteException {
    super();
}
```

Note the following:

- The `super` method call invokes the no-argument constructor of `javax.rmi.PortableRemoteObject`, which exports the remote object.
- The constructor must throw `java.rmi.RemoteException`, because RMI's attempt to export a remote object during construction might fail if communication resources are not available.

If you are interested in why `java.rmi.RemoteException` is a checked exception rather than runtime exception, please refer to the archives of the `rmi-users` email list:

<http://archives.java.sun.com/archives/rmi-users.html>

Although the call to the superclass's no-argument constructor, `super()`, occurs by default (even if omitted), it is included in this example to make clear the fact that the superclass will be constructed 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 from MARS!!" to the caller:

```
public void sayHello( String from ) throws java.rmi.RemoteException {
    System.out.println( "Hello from " + from + "!!");
    System.out.flush();
}
```

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

- By default, local objects are passed by copy, which means that all data members (or fields) of an object are copied, except those marked as `static` or `transient`. Please refer to the [Java Object Serialization Specification](#) for information on how to alter the default serialization behavior.
- 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 the [Java Remote Method Invocation Specification](#). We'll create them later in this tutorial in the section: [Use `rmic` to generate stubs and skeletons](#).

Write The Server Class

A server class 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 Naming Service. The class that contains this `main` method could be the implementation class itself, or another class entirely.

In this example, the `main` method is part of `HelloServer.java`, which does the following:

- [Creates an instance of the servant](#)
- [Publishes the object reference](#)



Create the file `HelloServer.java`. The source code for this file follows. An explanation of each of the preceding steps follows the source code:

```
//HelloServer.java
import javax.naming.InitialContext;
import javax.naming.Context;

public class HelloServer {
    public static void main(String[] args) {
        try {
            // Step 1: Instantiate the Hello servant
            HelloImpl helloRef = new HelloImpl();

            // Step 2: Publish the reference - the Naming Service
            // using JNDI API
            Context initialNamingContext = new InitialContext();
            initialNamingContext.rebind("HelloService", helloRef );

            System.out.println("Hello Server: Ready...");

        } catch (Exception e) {
            System.out.println("Trouble: " + e);
            e.printStackTrace();
        }
    }
}
```

Create an instance of a remote object

The `main` method of the server needs to create an instance of the remote object implementation, or *Servant*. For example:

```
HelloImpl helloRef = new HelloImpl();
```

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

Publish the object reference

For a caller (client, peer, or client application) to be able to invoke a method on a remote object, that caller must first obtain a reference to the remote object.

Once a remote object is registered on the server, callers can look up the object by name (using a naming service), obtain a remote object reference, and then remotely invoke methods on the object. In this example, we use the **Naming Service that is part of the Object Request Broker Daemon (orbd)**.

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

```
// Step 2: Publish the reference in the Naming Service
// using JNDI API
Context initialNamingContext = new InitialContext();
initialNamingContext.rebind("HelloService", helloRef );
```

Note the following about the arguments to the `rebind` method call:

- The first argument, "HelloService", is a `java.lang.String`, representing the name of the remote object to bind
- The second argument, `helloRef` is the object id of the remote object to bind

Write a client program that uses the remote service

The client application in this example remotely invokes the `sayHello` method in order to get the string "Hello from MARS!!" to display when the client application runs.



Create the file `HelloClient.java`. Here is the source code for the client application:

```
//HelloClient.java
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import javax.rmi.*;
import java.util.Vector;
import javax.naming.NamingException;
import javax.naming.InitialContext;
import javax.naming.Context;

public class HelloClient {

    public static void main( String args[] ) {
        Context ic;
        Object objref;
        HelloInterface hi;

        try {
            ic = new InitialContext();

            // STEP 1: Get the Object reference from the Name Service
            // using JNDI call.
            objref = ic.lookup("HelloService");
            System.out.println("Client: Obtained a ref. to Hello server.");

            // STEP 2: Narrow the object reference to the concrete type and
            // invoke the method.
            hi = (HelloInterface) PortableRemoteObject.narrow(
                objref, HelloInterface.class);
            hi.sayHello( " MARS " );

        } catch( Exception e ) {
            System.err.println( "Exception " + e + "Caught" );
            e.printStackTrace( );
            return;
        }
    }
}
```

First, the client application gets a reference to the remote object implementation (advertised as "HelloService") from the Naming Service using Java Naming and Directory Interface [TM] (JNDI) calls. Like the `Naming.rebind` method, the `Naming.lookup` method takes `java.lang.String` value representing the name of the object to look up. You supply `Naming.lookup()` the name of the object you want to look up, and it returns the object bound to that name. `Naming.lookup()` returns the stub for the remote implementation of the `Hello` interface to its caller (`HelloClient`).

- The client application invokes the remote `sayHello()` method on the server's remote object, causing the string "Hello from MARS!!" to be displayed on the command line.

Compile the Example

The source code for this example is now complete and the directory contains **four files**:

- `HelloInterface.java` contains the source code for the remote interface
- `HelloImpl.java` contains the source code for the remote object implementation
- `HelloServer.java` contains the source code for the server
- `HelloClient.java` contains the source code for the client application

In this section, you compile the remote object implementation file, `HelloImpl.java`, in order to create the `.class` files needed to run `rmic`. 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-IIOP calls to the server-side dispatcher, which in turn forwards the call to the actual remote object implementation. The last task is to compile the remaining `.java` source files to create `.class` files.

The following tasks will be completed in this section:

1. [Compile the remote object implementation](#)
2. [Use `rmic` to generate stubs and skeletons](#)
3. [Compile the source files](#)

Compile the remote object implementation

To create stub and skeleton files, the `rmic` compiler must be run on the fully-qualified package names of compiled class files that contain remote object implementations. In this example, the file that contains the remote object implementations is `HelloImpl.java`. To generate the stubs and skeletons:



Compile `HelloImpl.java`, as follows:

```
javac -d . -classpath . HelloImpl.java
```

The "-d ." option indicates that the generated files should be placed in the directory from which you are running the compiler. The "-classpath ." option indicates that files on which HelloImpl.java is dependent can be found in this directory.

Use `rmic` to generate skeletons and stubs

To create CORBA-compatible stub and skeleton files, run the `rmic` compiler with the `-iiop` option. The `rmic -iiop` command takes one or more class names as an argument and produces class files of the form `_HelloImpl_Tie.class` and `_HelloInterface_Stub.class`. The remote implementation file, `HelloImpl.class`, is the class name to pass in this example.

For an explanation of `rmic` options, you can refer to the [Solaris `rmic` manual page](#) or the [Windows `rmic` manual page](#).



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

```
rmic -iiop HelloImpl
```

The preceding command creates the following files:

- `_HelloInterface_Stub.class` - the client stub
- `_HelloImpl_Tie.class` - the server skeleton

Compile the source files



Compile the source files as follows:

```
javac -d . -classpath . HelloInterface.java HelloServer.java HelloClient.java
```

This command creates the class files `HelloInterface.class`, `HelloServer.class`, and `HelloClient.class`. These are the remote interface, the server, and the client application respectively. For an explanation of `javac` options, you can refer to the [Solaris `javac` manual page](#) or the [Windows `javac` manual page](#).

Run the Example

The following tasks will be completed in this section:

1. [Start the Naming Service](#)
2. [Start the server](#)
3. [Run the client application](#)

Start the Naming Service

For this example, we will use the Object Request Broker Daemon, `orbd`, which includes both a Transient and a Persistent Naming Service, and is available with every download of J2SE 1.4 and higher.

For a caller (client, peer, or client application) to be able to invoke a method on a remote object, that caller must first obtain a reference to the remote object.

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.



Start the Naming Service by running `orbd` from the command line.

For this example, on the Solaris operating system:

```
orbd -ORBInitialPort 1050&
```

or, on the Windows operating system:

```
start orbd -ORBInitialPort 1050
```

You must specify a port on which to run `orbd`. For this example the port of 1050 is chosen because in the Solaris operating environment, a user must become root to start a process on a port under 1024. For more on the `orbd` tool, you can refer to the [orbd manual page](#).

You must stop and restart the server any time you modify a remote interface or use modified/additional remote interfaces in a remote object implementation. Otherwise, the type of the object reference bound in the Naming Service will not match the modified class.

Start the server

Open another terminal window and change to the directory containing the source files for this example. The command for running the client has been spread out below to make it easier to read, but should be typed without returns between the lines. The following command shows how to start the `HelloServer` server. If you used a port other than 1050 or a host other than `localhost` when starting the `orbd` tool, replace those values in the command below with the actual values used to start `orbd`.



Start the Hello server, as follows:

```
java
  -classpath .
  -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
  -Djava.naming.provider.url=iiop://localhost:1050
  HelloServer
```

For an explanation of `java` options, you can refer to the [Solaris java manual page](#) or the [Windows java manual page](#).

The output should look like this:

```
Hello Server: Ready ...
```

Run the client application

Once the Naming Service and server are running, the client application can be run. From a new terminal window, go to the source code directory, and run the client application from the command line, as shown below. The command for running the client has been spread out below to make it easier to read, but should be typed without returns between the lines. If you used a port other than 1050 or a host other than `localhost` when starting the `orbd` tool, replace those values in the command below with the actual values used to start `orbd`.



Start the client application, as follows:

```
java
  -classpath .
  -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
  -Djava.naming.provider.url=iiop://localhost:1050
  HelloClient
```

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

Client: Obtained a ref. to Hello server.

Hello from MARS

ORBD and the Hello server will continue to run until they are explicitly stopped. On Solaris, you can stop these processes using the `pkill orbd` and `pkill HelloServer` commands from a terminal window. On Windows, you can type `Ctrl+C` in a prompt window to kill the process.

This completes the basic RMI-IIOP tutorial. If you are ready to move on to more complicated applications, here are some sources that may help:

- <http://forum.java.sun.com/forum.jsp?forum=59>, The Sun Developer's Forum for RMI-IIOP. You must register with the Java Developer Connection to access this site.
- EJB clients interact with the J2EE EJB tier using the RMI-IIOP protocol. For more information on using RMI-IIOP in this way, see the Java 2 Platform, Enterprise Edition [Blueprints](#) or the [EJB Tutorials](#).
- Another [RMI-IIOP tutorial](#) includes support for the Portable Object Adapter (POA). POA support for RMI-IIOP is non-standard.
- The Java IDL tutorial includes an example for running the client and server on different machines. The concepts of [Running the Hello World Example on Two Machines](#) apply to this example as well.

[RMI-IIOP Documentation Home](#)

Send questions or comments to: rmi-iiop@sun.com