Revised and Updated for Java SE 6

Core Java™

Volume II • Advanced Features

EIGHTH EDITION

Cay S. Horstmann • Gary Cornell

Chapter 10. Distributed Objects

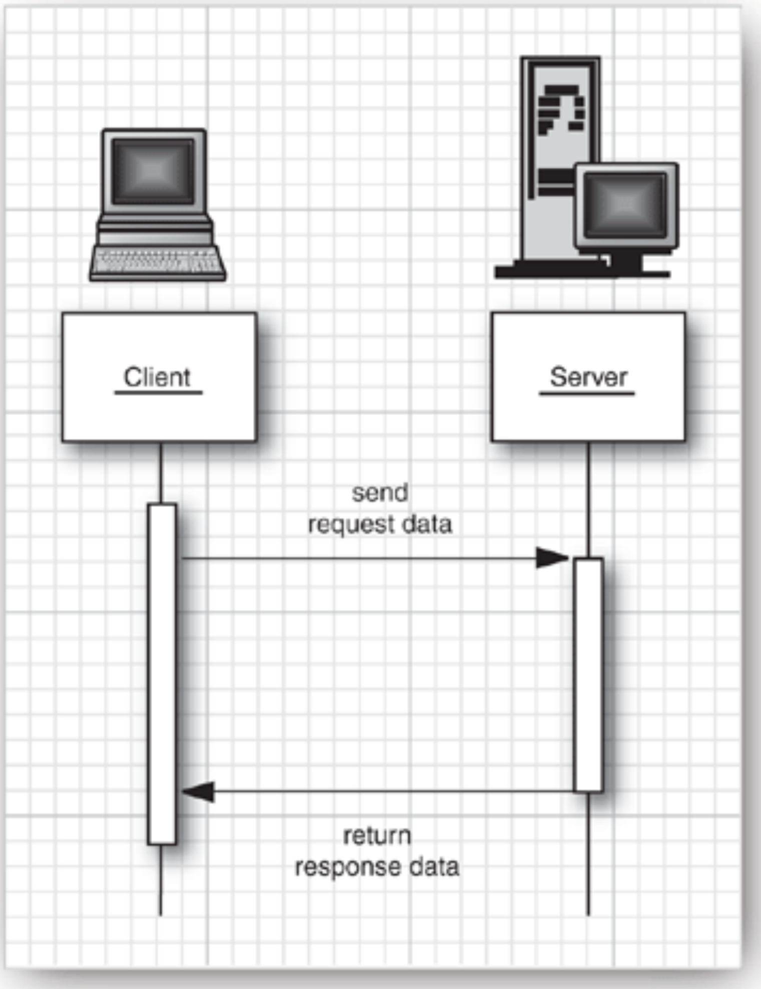
Periodically, the programming community starts thinking of "objects everywhere" as the solution to all its problems. The idea is to have a happy family of collaborating objects that can be located anywhere. When an object on one computer needs to invoke a method on an object on another computer, it sends a network message that contains the details of the request. The remote object computes a response, perhaps by accessing a database or by communicating with additional objects. Once the remote object has the answer to the client request, it sends the answer back over the network. Conceptuatlly, this process sounds quite simple, but you need to understand what goes on under the hood to use distributed objects effectively.

In this chapter, we focus on Java technologies for distributed programming, in particular the *Remote Method Invocation* (RMI) protocol for communicating between two Java virtual machines (which might run on different computers). We then briefly visit the JAX-WS technology for making remote calls to web services.

The Roles of Client and Server

The basic idea behind all distributed programming is simple. A client computer makes a request and sends the request data across a network to a server. The server processes the request and sends back a response for the client to analyze. Figure 10-1 shows the process.

Figure 10-1. Transmitting objects between client and server



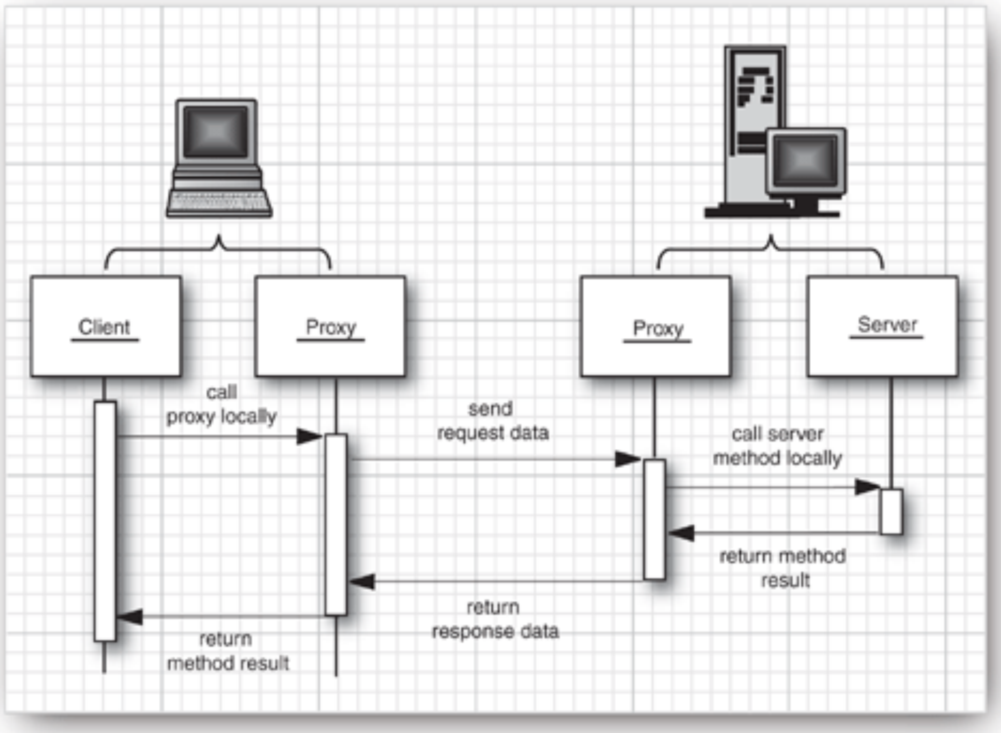
We would like to say at the outset that these requests and responses are *not* what you would see in a web application. The client is not a web browser. It can be any application that executes business rules of any complexity. The client application might or might not interact with a human user, and if it does, it can have a command-line or Swing user interface. The protocol for the request and response data allows the transfer of arbitrary objects, whereas traditional web applications are limited by using HTTP for the request and HTML for the response.

What we want is a mechanism by which the client programmer makes a regular method call, without worrying about sending data across the network or parsing the response. The solution is to install a *proxy* object on the client. The proxy is an object located in the client virtual machine that appears to the client program as if it was the remote object. The client calls the proxy, making a regular method call. The client proxy contacts the server, using a network protocol.

Similarly, the programmer who implements the service doesn't want to fuss with client communication. The solution is to install a second proxy object on the server. The server proxy communicates with the client proxy, and it makes regular method calls to the object implemeting the service (see Figure 10-2).

Figure 10-2. Remote method call with proxies

[View full size image]



How do the proxies communicate with each other? That depends on the implementation technology. There are three common choices:

- The Java RMI technology supports method calls between distributed Java objects.
- The Common Object Request Broker Architecture (CORBA) supports method calls between objects of any programming language. CORBA uses the binary Internet Inter-ORB Protocol, or IIOP, to communicate between objects.
- The web services architecture is a collection of protocols, sometimes collectively described as WS-*. It is also programming-language neutral. However, it uses XML-based communication formats. The format for transmitting objects is the Simple Object Access Protocol (SOAP).

If the communicating programs are implemented in Java code, then the full generality and complexity of CORBA or WS-* is not required. Sun developed a simple mechanism, called RMI, specifically for communication between Java applications.

It is well worth learning about RMI, even if you are not going to use it in your own programs. You will learn the mechanisms that are essential for programming distributed applications, using a straightforward architecture. Moreover, if you use enterprise Java technologies, it is very useful to have a basic understanding of RMI because that is the protocol used to communicate between enterprise Java beans (EJBs). EJBs are server-side components that are composed to make up complex applications that run on multiple servers. To make effective use of EJBs, you will want to have a good idea of the costs associated with remote calls.

Unlike RMI, CORBA and SOAP are completely language neutral. Client and server programs can be written in C, C++, C#, Java, or any other language. You supply an *interface description* to specify the signatures of the methods and the types of the data your objects can handle. These descriptions are formatted in a special language, called Interface Definition Language (IDL) for CORBA and Web Services Description Language (WSDL) for web services.

For many years, quite a few people believed that CORBA was the object model of the future. Frankly, though, CORBA has a reputationâ€”sometimes deservedâ€”for complex implementations and interoperability problems, and it has only reached modest success. We covered interoperability between Java and CORBA for five editions of this book, but dropped it for lack of interest. Our sentiments about CORBA are similar to those expressed by French president Charles De Gaulle about Brazil: It has a great future . . . and always will.

Web services had a similar amount of buzz when they first appeared, with the promise that they are simpler and, of course, founded in the goodness of the World Wide Web and XML. However, with the passing of time and the work of many committees, the protocol stack has become less simple, as it acquired more of the features that CORBA had all along. The XML protocol has the advantage of being (barely) human-readable, which helps with debugging. On the other hand, XML processing is a significant performance bottleneck. Recently, the WS-* stack has lost quite a bit of its luster and it too is gaining a reputationâ€”sometimes deservedâ€—for complex implementations and interoperability problems.

We close this chapter with an example of an application that consumes a web service. We have a look at the underlying protocol so that you can see how communication between different programming languages is implemented.

Chapter 10. Distributed Objects

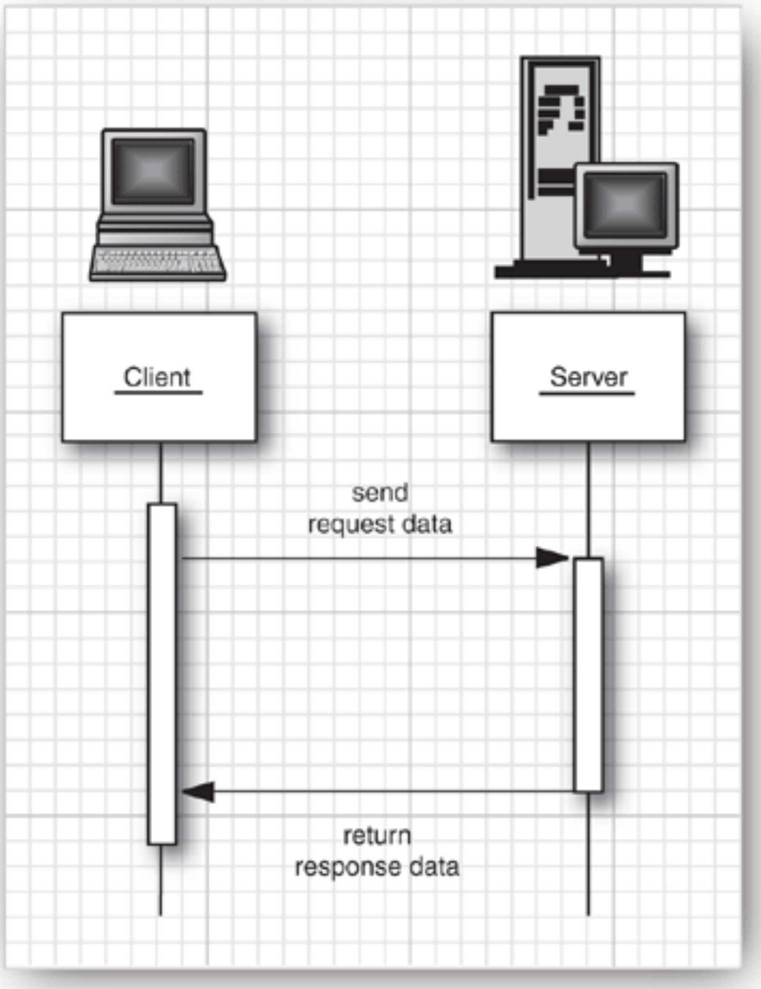
Periodically, the programming community starts thinking of "objects everywhere" as the solution to all its problems. The idea is to have a happy family of collaborating objects that can be located anywhere. When an object on one computer needs to invoke a method on an object on another computer, it sends a network message that contains the details of the request. The remote object computes a response, perhaps by accessing a database or by communicating with additional objects. Once the remote object has the answer to the client request, it sends the answer back over the network. Conceptuatlly, this process sounds quite simple, but you need to understand what goes on under the hood to use distributed objects effectively.

In this chapter, we focus on Java technologies for distributed programming, in particular the *Remote Method Invocation* (RMI) protocol for communicating between two Java virtual machines (which might run on different computers). We then briefly visit the JAX-WS technology for making remote calls to web services.

The Roles of Client and Server

The basic idea behind all distributed programming is simple. A client computer makes a request and sends the request data across a network to a server. The server processes the request and sends back a response for the client to analyze. Figure 10-1 shows the process.

Figure 10-1. Transmitting objects between client and server



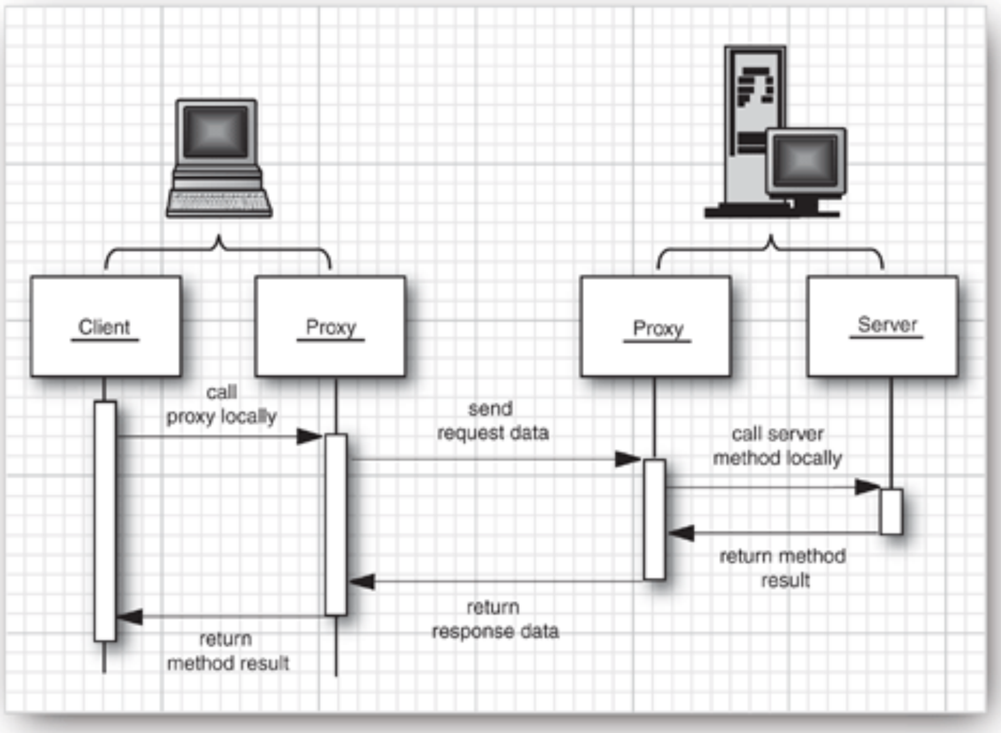
We would like to say at the outset that these requests and responses are *not* what you would see in a web application. The client is not a web browser. It can be any application that executes business rules of any complexity. The client application might or might not interact with a human user, and if it does, it can have a command-line or Swing user interface. The protocol for the request and response data allows the transfer of arbitrary objects, whereas traditional web applications are limited by using HTTP for the request and HTML for the response.

What we want is a mechanism by which the client programmer makes a regular method call, without worrying about sending data across the network or parsing the response. The solution is to install a *proxy* object on the client. The proxy is an object located in the client virtual machine that appears to the client program as if it was the remote object. The client calls the proxy, making a regular method call. The client proxy contacts the server, using a network protocol.

Similarly, the programmer who implements the service doesn't want to fuss with client communication. The solution is to install a second proxy object on the server. The server proxy communicates with the client proxy, and it makes regular method calls to the object implemeting the service (see Figure 10-2).

Figure 10-2. Remote method call with proxies

[View full size image]



How do the proxies communicate with each other? That depends on the implementation technology. There are three common choices:

- The Java RMI technology supports method calls between distributed Java objects.
- The Common Object Request Broker Architecture (CORBA) supports method calls between objects of any programming language. CORBA uses the binary Internet Inter-ORB Protocol, or IIOP, to communicate between objects.
- The web services architecture is a collection of protocols, sometimes collectively described as WS-*. It is also programming-language neutral. However, it uses XML-based communication formats. The format for transmitting objects is the Simple Object Access Protocol (SOAP).

If the communicating programs are implemented in Java code, then the full generality and complexity of CORBA or WS-* is not required. Sun developed a simple mechanism, called RMI, specifically for communication between Java applications.

It is well worth learning about RMI, even if you are not going to use it in your own programs. You will learn the mechanisms that are essential for programming distributed applications, using a straightforward architecture. Moreover, if you use enterprise Java technologies, it is very useful to have a basic understanding of RMI because that is the protocol used to communicate between enterprise Java beans (EJBs). EJBs are server-side components that are composed to make up complex applications that run on multiple servers. To make effective use of EJBs, you will want to have a good idea of the costs associated with remote calls.

Unlike RMI, CORBA and SOAP are completely language neutral. Client and server programs can be written in C, C++, C#, Java, or any other language. You supply an *interface description* to specify the signatures of the methods and the types of the data your objects can handle. These descriptions are formatted in a special language, called Interface Definition Language (IDL) for CORBA and Web Services Description Language (WSDL) for web services.

For many years, quite a few people believed that CORBA was the object model of the future. Frankly, though, CORBA has a reputationâ€"sometimes deservedâ€"for complex implementations and interoperability problems, and it has only reached modest success. We covered interoperability between Java and CORBA for five editions of this book, but dropped it for lack of interest. Our sentiments about CORBA are similar to those expressed by French president Charles De Gaulle about Brazil: It has a great future . . . and always will.

Web services had a similar amount of buzz when they first appeared, with the promise that they are simpler and, of course, founded in the goodness of the World Wide Web and XML. However, with the passing of time and the work of many committees, the protocol stack has become less simple, as it acquired more of the features that CORBA had all along. The XML protocol has the advantage of being (barely) human-readable, which helps with debugging. On the other hand, XML processing is a significant performance bottleneck. Recently, the WS-* stack has lost quite a bit of its luster and it too is gaining a reputationâ€"sometimes deservedâ€"for complex implementations and interoperability problems.

We close this chapter with an example of an application that consumes a web service. We have a look at the underlying protocol so that you can see how communication between different programming languages is implemented.

Remote Method Calls

The key to distributed computing is the *remote method call*. Some code on one machine (called the *client*) wants to invoke a method on an object on another machine (the *remote object*). To make this possible, the method parameters must somehow be shipped to the other machine, the server must be informed to locate the remote object and execute the method, and the return value must be shipped back.

Before looking at this process in detail, we want to point out that the client/server terminology applies only to a single method call. The computer that calls the remote method is the client for *that* call, and the computer hosting the object that processes the call is the server for *that* call. It is entirely possible that the roles are reversed somewhere down the road. The server of a previous call can itself become the client when it invokes a remote method on an object residing on another computer.

Stubs and Parameter Marshalling

When client code wants to invoke a method on a remote object, it actually calls an ordinary method on a proxy object called a *stub*. For example,

```
Warehouse centralWarehouse = get stub object;
double price = centralWarehouse.getPrice("Blackwell Toaster");
```

The stub resides on the client machine, not on the server. It knows how to contact the server over the network. The stub packages the parameters used in the remote method into a block of bytes. The process of encoding the parameters is called *parameter marshalling*. The purpose of parameter marshalling is to convert the parameters into a format suitable for transport from one virtual machine to another. In the RMI protocol, objects are encoded with the serialization mechanism that is described in Chapter 1. In the SOAP protocol, objects are encoded as XML.

To sum up, the stub method on the client builds an information block that consists of

- An identifier of the remote object to be used.
- A description of the method to be called.
- The parameters.

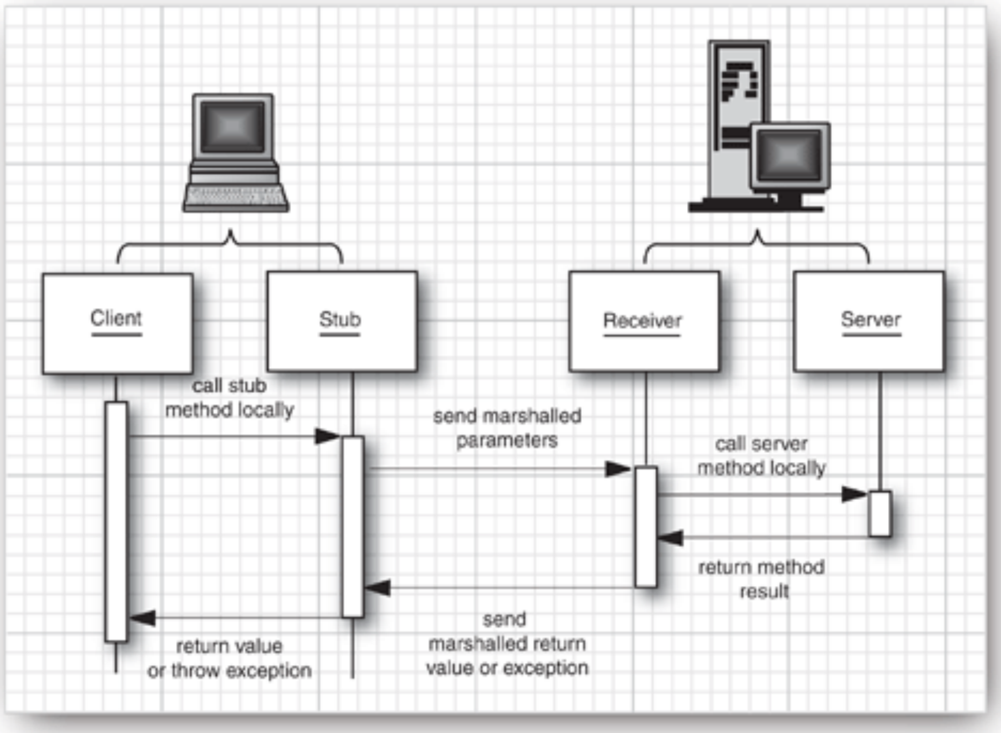
The stub then sends this information to the server. On the server side, a receiver object performs the following actions:

1. It locates the remote object to be called.
2. It calls the desired method, passing the supplied parameters.
3. It captures the return value or exception of the call.
4. It sends a package consisting of the marshalled return data back to the stub on the client.

The client stub unmarshals the return value or exception from the server. This value becomes the return value of the stub call. Or, if the remote method threw an exception, the stub rethrows it in the virtual machine of the caller. Figure 10-3 shows the information flow of a remote method invocation.

Figure 10-3. Parameter marshalling

[View full size image]


Client
Stub
Receiver
Server
call stub method locally
send marshalled parameters
call server method locally
return method result
send marshalled return value or exception
return value or throw exception

This process is obviously complex, but the good news is that it is completely automatic and, to a large extent, transparent for the programmer.

The details for implementing remote objects and for getting client stubs depend on the technology for distributed objects. In the following sections, we have a close look at RMI.

The RMI Programming Model

To introduce the RMI programming model, we start with a simple example. A remote object represents a warehouse. The client program asks the warehouse about the price of a product. In the following sections, you will see how to implement and launch the server and client programs.

Interfaces and Implementations

The capabilities of remote objects are expressed in interfaces that are shared between the client and server. For example, the interface in Listing 10-1 describes the service provided by a remote warehouse object:

Listing 10-1. `Warehouse.java`

```
import java.rmi.*;

/**
   The remote interface for a simple warehouse.
   @version 1.0 2007-10-09
   @author Cay Horstmann
*/
public interface Warehouse extends Remote
{
   double getPrice(String description) throws RemoteException;
}
```

Interfaces for remote objects must always extend the `Remote` interface defined in the `java.rmi` package. All the methods in those interfaces must also declare that they will throw a `RemoteException`. Remote method calls are inherently less reliable than local callsâ€"it is always possible that a remote call will fail. For example, the server might be temporarily unavailable, or there might be a network problem. Your client code must be prepared to deal with these possibilities. For these reasons, you must handle the `RemoteException` with *every* remote method call and specify the appropriate action to take when the call does not succeed.

Next, on the server side, you must provide the class that actually carries out the work advertised in the remote interfaceâ€"see Listing 10-2.

Listing 10-2. `WarehouseImpl.java`

Code View:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

/**
 * This class is the implementation for the remote Warehouse interface.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
   public WarehouseImpl() throws RemoteException
   {
      prices = new HashMap<String, Double>();
      prices.put("Blackwell Toaster", 24.95);
      prices.put("ZapXpress Microwave Oven", 49.95);
   }

   public double getPrice(String description) throws RemoteException
   {
      Double price = prices.get(description);
      return price == null ? 0 : price;
   }

   private Map<String, Double> prices;
}
```

Note

✓ The `WarehouseImpl` constructor is declared to throw a `RemoteException` because the superclass constructor can throw that exception. This happens when there is a problem connecting to the network service that tracks remote objects.

You can tell that the class is the target of remote method calls because it extends `UnicastRemoteObject`. The constructor of that class makes objects remotely accessible. The "path of least resistance" is to derive from `UnicastRemoteObject`, and all service implementation classes in this chapter do so.

Occasionally, you might not want to extend the `UnicastRemoteObject` class, perhaps because your implementation class already extends another class. In that situation, you need to manually instantiate the remote objects and pass them to the static `exportObject` method. Instead of extending `UnicastRemoteObject`, call

```
UnicastRemoteObject.exportObject(this, 0);
```

in the constructor of the remote object. The second parameter is 0 to indicate that any suitable port can be used to listen to client connections.

Note

✓ The term "unicast" refers to the fact that the remote object is located by making a call to a single IP address and port. This is the only mechanism that is supported in Java SE. More sophisticated distributed object systems (such as JINI) allow for "multicast" lookup of remote objects that might be on a number of different servers.

The RMI Registry

To access a remote object that exists on the server, the client needs a local stub object. How can the client request such a stub? The most common method is to call a remote method of another remote object and get a stub object as a return value. There is, however, a chicken-and-egg problem here: The *first* remote object has to be located some other way. For that purpose, the JDK provides a *bootstrap registry service*.

A server program registers at least one remote object with a bootstrap registry. To register a remote object, you need a RMI URL and a reference to the implementation object.

RMI URLs start with `rmi:` and contain an optional host name, an optional port number, and the name of the remote object that is (hopefully) unique. An example is:

```
rmi://regserver.mycompany.com:99/central_warehouse
```

By default, the host name is `localhost` and the port number is 1099. The server tells the registry at the given location to associate or "bind" the name with the object.

Here is the code for registering a `WarehouseImpl` object with the RMI registry on the same server:

```
WarehouseImpl centralWarehouse = new WarehouseImpl();
Context namingContext = new InitialContext();
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

The program in Listing 10-3 simply constructs and registers a `WarehouseImpl` object.

Listing 10-3. `WarehouseServer.java`

Code View:

```
import java.rmi.*;
import javax.naming.*;

/**
  * This server program instantiates a remote warehouse object, registers it with the naming
  * service, and waits for clients to invoke methods.
  * @version 1.12 2007-10-09
  * @author Cay Horstmann
  */

public class WarehouseServer
{
   public static void main(String[] args) throws RemoteException, NamingException
   {
      System.out.println("Constructing server implementation...");
      WarehouseImpl centralWarehouse = new WarehouseImpl();

      System.out.println("Binding server implementation to registry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:central_warehouse", centralWarehouse);

      System.out.println("Waiting for invocations from clients...");
   }
}
```

Note

✔

For security reasons, an application can bind, unbind, or rebind registry object references only if it runs on the same host as the registry. This prevents hostile clients from changing the registry information. However, any client can look up objects.

A client can enumerate all registered RMI objects by calling:

Code View:

```
Enumeration<NameClassPair> e = namingContext.list("rmi://regserver.mycompany.com");
```

`NameClassPair` is a helper class that contains both the name of the bound object and the name of its class. For example, the following code displays the names of all registered objects:

```
while (e.hasMoreElements())
   System.out.println(e.nextElement().getName());
```

A client gets a stub to access a remote object by specifying the server and the remote object name in the following way:

```
String url = "rmi://regserver.mycompany.com/central_warehouse";
Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
```

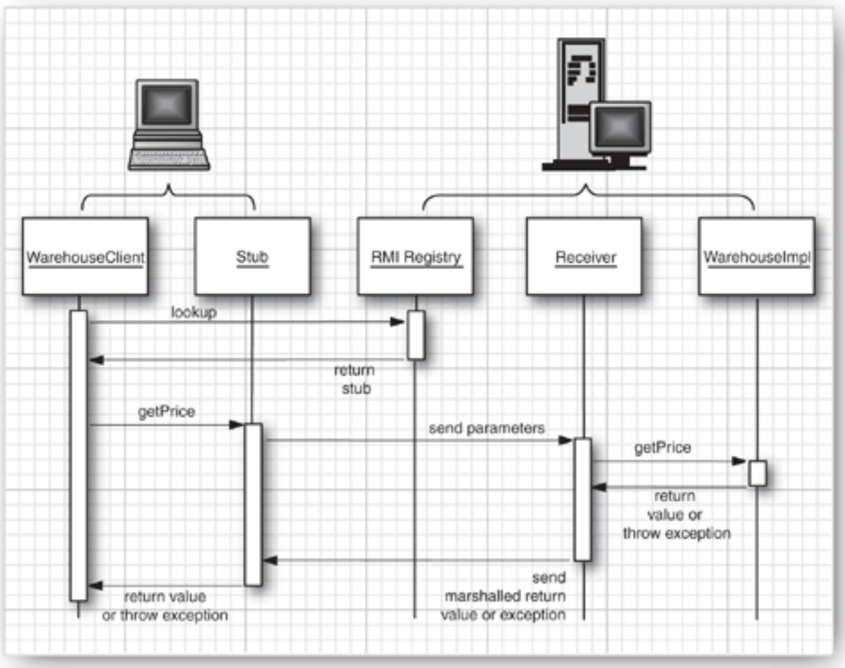
Note

Because it is notoriously difficult to keep names unique in a global registry, you should not use this technique as the general method for locating objects on the server. Instead, there should be relatively few named remote objects registered with the bootstrap service. These should be objects that can locate other objects for you.

The code in Listing 10-4 shows the client that obtains a stub to the remote warehouse object and invokes the remote `getPrice` method. Figure 10-4 shows the flow of control. The client obtains a `Warehouse` stub and invokes the `getPrice` method on it. Behind the scenes, the stub contacts the server and causes the `getPrice` method to be invoked on the `WarehouseImpl` object.

Figure 10-4. Calling the remote `getDescription` method

[View full size image]


WarehouseClient
Stub
RMI Registry
Receiver
WarehouseImpl
lookup
return stub
getPrice
send parameters
getPrice
return value or throw exception
send marshalled return value or exception
return value or throw exception

Listing 10-4. `WarehouseClient.java`

Code View:

```
import java.rmi.*;
import java.util.*;
import javax.naming.*;

/**
 * A client that invokes a remote method.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseClient
{
   public static void main(String[] args) throws NamingException, RemoteException
   {
      Context namingContext = new InitialContext();

      System.out.print("RMI registry bindings: ");
      Enumeration<NameClassPair> e = namingContext.list("rmi://localhost/");
      while (e.hasMoreElements())
         System.out.println(e.nextElement().getName());

      String url = "rmi://localhost/central_warehouse";
      Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);

      String descr = "Blackwell Toaster";
      double price = centralWarehouse.getPrice(descr);
      System.out.println(descr + ": " + price);
   }
}
```

API `javax.naming.InitialContext` 1.3

- `InitialContext()`

  constructs a naming context that can be used for accessing the RMI registry.

API `javax.naming.Context` 1.3

- `static Object lookup(String name)`

  returns the object for the given name. Throws a `NamingException` if the name is not currently bound.

- `static void bind(String name, Object obj)`

  binds `name` to the object `obj`. Throws a `NameAlreadyBoundException` if the object is already bound.

- `static void unbind(String name)`

  unbinds the name. It is legal to unbind a name that doesn't exist.

- `static void rebind(String name, Object obj)`

  binds `name` to the object `obj`. Replaces any existing binding.

- `NamingEnumeration<NameClassPair> list(String name)`

  returns an enumeration listing all matching bound objects. To list all RMI objects, call with "`rmi:`".

API `javax.naming.NameClassPair` 1.3

- `String getName()`

  gets the name of the named object.

- `String getClassName()`

  gets the name of the class to which the named object belongs.

API `java.rmi.Naming` 1.1

- `static Remote lookup(String url)`

  returns the remote object for the URL. Throws a `NotBoundException` if the name is not currently bound.

- `static void bind(String name, Remote obj)`

  binds `name` to the remote object `obj`. Throws an `AlreadyBoundException` if the object is already bound.

- `static void unbind(String name)`

  unbinds the name. Throws the `NotBound` exception if the name is not currently bound.

- `static void rebind(String name, Remote obj)`

  binds `name` to the remote object `obj`. Replaces any existing binding.

- `static String[] list(String url)`

  returns an array of strings of the URLs in the registry located at the given URL. The array contains a snapshot of the names present in the registry.

Deploying the Program

Deploying an application that uses RMI can be tricky because so many things can go wrong and the error messages that you get when something does go wrong are so poor. We have found that it really pays off to test the deployment under realistic conditions, separating the classes for client and server.

Make two separate directories to hold the classes for starting the server and client.

```
server/
   WarehouseServer.class
   Warehouse.class
   WarehouseImpl.class

client/
   WarehouseClient.class
   Warehouse.class
```

When deploying RMI applications, one commonly needs to dynamically deliver classes to running programs. One example is the RMI registry. Keep in mind that one instance of the registry will serve many different RMI applications. The RMI registry needs to have access to the class files of the service interfaces that are being registered. When the registry starts, however, one cannot predict all future registration requests. Therefore, the RMI registry dynamically loads the class files of any remote interfaces that it has not previously encountered.

Dynamically delivered class files are distributed through standard web servers. In our case, the server program needs to make the `Warehouse.class` file available to the RMI registry, so we put that file into a third directory that we call `download`.

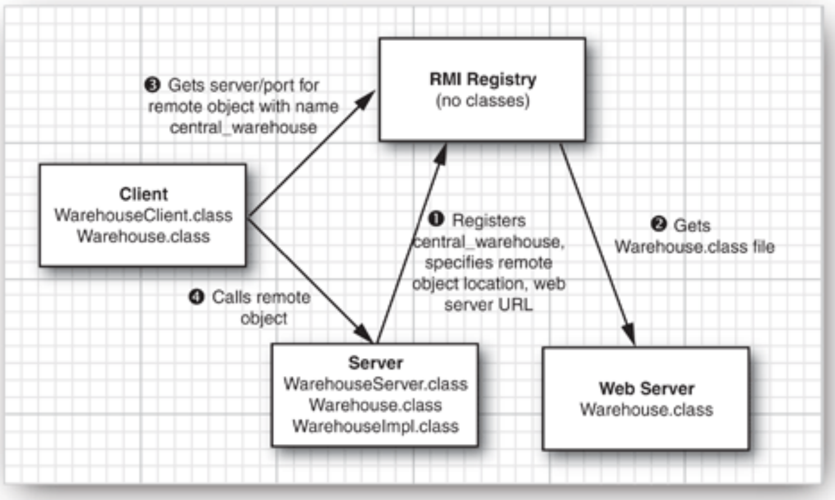
```
download/
   Warehouse.class
```

We use a web server to serve the contents of that directory.

When the application is deployed, the server, RMI registry, web server, and client can be located on four different computersâ€”see Figure 10-5. However, for testing purposes, we will use a single computer.

Figure 10-5. Server calls in the Warehouse application

[View full size image]



Note

For security reasons, the `rmiregistry` service that is part of the JDK only allows binding calls from the same host. That is, the server and `rmiregistry` process need to be located on the same computer. However, the RMI architecture allows for a more general RMI registry implementation that supports multiple servers.

To test the sample application, use the `NanoHTTPD` web server that is available from http://elonen.iki.fi/code/nanohttpd. This tiny web server is implemented in a single Java source file. Open a new console window, change to the `download` directory, and copy `NanoHTTPD.java` to that directory. Compile the source file and start the web server, using the command

```
java NanoHTTPD 8080
```

The command-line argument is the port number. Use any other available port if port 8080 is already used on your machine.

Next, open another console window, change to a directory that *contains no class files*, and start the RMI registry:

```
rmiregistry
```

Caution

Before starting the RMI registry, make sure that the CLASSPATH environment variable is not set to anything, and double-check that the current directory contains no class files. Otherwise, the RMI registry might find spurious class files, which will confuse it when it should download additional classes from a different source. There is a reason for this behavior; see http://java.sun.com/javase/6/docs/technotes/guides/rmi/codebase.html. In a nutshell, each stub object has a *codebase* entry that specifies from where it was loaded. That codebase is used to load dependent classes. If the RMI registry finds a class locally, it will set the wrong codebase.

Now you are ready to start the server. Open a third console window, change to the `server` directory, and issue the command

```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
```

The `java.rmi.server.codebase` property points to the URL for serving class files. The server program communicates this URL to the RMI registry.

Have a peek at the console window running `NanoHTTPD`. You will see a message that demonstrates that the `Warehouse.class` file has been served to the RMI registry.

Caution

It is very important that you make sure that the codebase URL *ends with a slash* (/).

Note that the server program does not exit. This seems strangeâ€”after all, the program just creates a `WarehouseImpl` object and registers it. Actually, the `main` method does exit immediately after registration, as you would expect. However, when you create an object of a class that extends `UnicastRemoteObject`, a separate thread that keeps the program alive indefinitely is started. Thus, the program stays around to allow clients to connect to it.

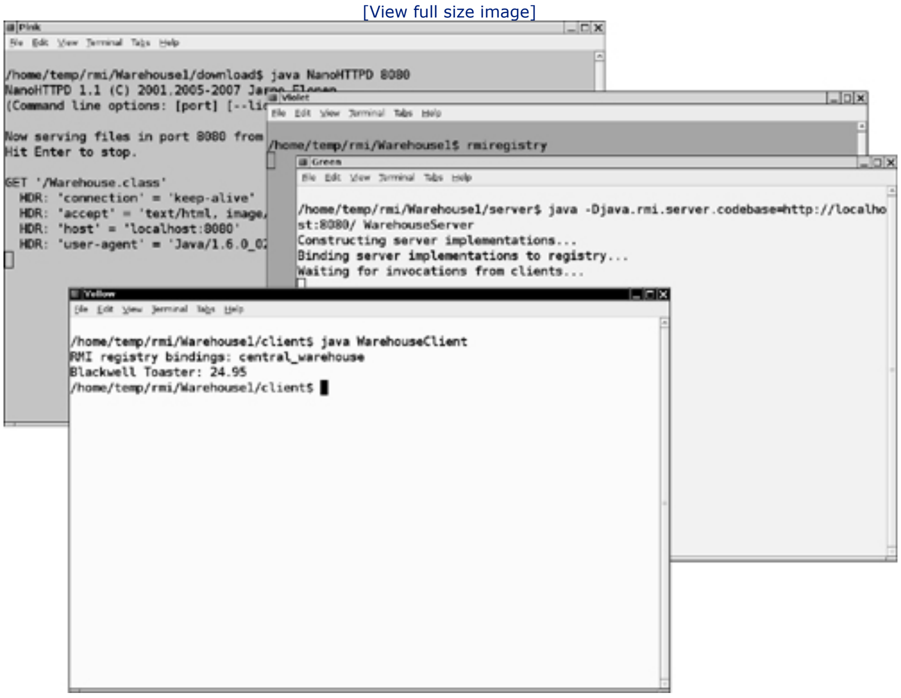
Finally, open a fourth console window, change to the `client` directory, and run

```
java WarehouseClient
```

You will see a short message, indicating that the remote method was successfully invoked (see Figure 10-6).

Figure 10-6. Testing an RMI application

[View full size image]



```
/home/temp/rmi/Warehouse1/download$ java NanoHTTPD 8080
NanoHTTPD 1.1 (C) 2001,2005-2007 Jarno Elonen
(Command line options: [port] [--lic
Now serving files in port 8080 from
Hit Enter to stop.

GET '/Warehouse.class'
  HDR: 'connection' = 'keep-alive'
  HDR: 'accept' = 'text/html, image,
  HDR: 'host' = 'localhost:8080'
  HDR: 'user-agent' = 'Java/1.6.0_0
```

```
/home/temp/rmi/Warehouse1$ rmiregistry
```

```
/home/temp/rmi/Warehouse1/server$ java -Djava.rmi.server.codebase=http://localho
st:8080/ WarehouseServer
Constructing server implementations...
Binding server implementations to registry...
Waiting for invocations from clients...
```

```
/home/temp/rmi/Warehouse1/client$ java WarehouseClient
RMI registry bindings: central_warehouse
Blackwell Toaster: 24.95
/home/temp/rmi/Warehouse1/client$
```

Note

✔ If you just want to test out basic program logic, you can put your client and server class files into the same directory. Then you can start the RMI registry, server, and client in that directory. However, because RMI class loading is the source of much grief and confusion, we felt it best to show you the correct setup for dynamic class loading right away.

Logging RMI Activity

If you start the server with the option

```
-Djava.rmi.server.logCalls=true WarehouseServer &
```

then the server logs all remote method calls on its console. Try itâ€”you'll get a good impression of the RMI traffic.

If you want to see additional logging messages, you have to configure RMI loggers, using the standard Java logging API. (See Volume I, Chapter 11 for more information on logging.)

Make a file `logging.properties` with the following content:

Code View:

```
handlers=java.util.logging.ConsoleHandler
.level=FINE
java.util.logging.ConsoleHandler.level=FINE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

You can fine-tune the settings by setting individual levels for each logger rather than setting the global level. Table 10-1 lists the RMI loggers. For example, to track the class loading activity, you can set

```
sun.rmi.loader.level=FINE
```

Table 10-1. RMI Loggers

Logger Name	Logged Activity
`sun.rmi.server.call`	Server-side remote calls
`sun.rmi.server.ref`	Server-side remote references
`sun.rmi.client.call`	Client-side remote calls
`sun.rmi.client.ref`	Client-side remote references
`sun.rmi.dgc`	Distributed garbage collection
`sun.rmi.loader`	`RMIClassLoader`
`sun.rmi.transport.misc`	Transport layer
`sun.rmi.transport.tcp`	TCP binding and connection
`sun.rmi.transport.proxy`	HTTP tunneling

Start the RMI registry with the option

```
-J-Djava.util.logging.config.file=directory/logging.properties
```

Start the client and server with

```
-Djava.util.logging.config.file=directory/logging.properties
```

Here is an example of a logging message that shows a class loading problem: The RMI registry cannot find the `Warehouse` class because the web server has been shut down.

Code View:

```
FINE: RMI TCP Connection(1)-127.0.1.1: (port 1099) op = 80
Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.1.1: interfaces = [java.rmi.Remote, Warehouse], codebase =
"http://localhost:8080/"
Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.1.1: proxy class resolution failed
java.lang.ClassNotFoundException: Warehouse
```

Parameters and Return Values in Remote Methods

At the start of a remote method invocation, the parameters need to be moved from the virtual machine of the client to the virtual machine of the server. After the invocation has completed, the return value needs to be transferred in the other direction. When a value is passed from one virtual machine to another other, we distinguish two cases: passing remote objects and passing nonremote objects. For example, suppose that a client of the `WarehouseServer` passes a `Warehouse` reference (that is, a stub through which the remote warehouse object can be called) to another remote method. That is an example of passing a remote object. However, most method parameters will be ordinary Java objects, not stubs to remote objects. An example is the `String` parameter of the `getPrice` method in our first sample application.

Transferring Remote Objects

When a reference to a remote object is passed from one virtual machine to the other, the sender and recipient of the remote object both hold a reference to the same entity. That reference is not a memory location (which is only meaningful in a single virtual machine), but it consists of a network address and a unique identifier for the remote object. This information is encapsulated in a stub object.

Conceptually, passing a remote reference is quite similar to passing local object references within a virtual machine. However, always keep in mind that a method call on a remote reference is significantly slower and potentially less reliable than a method call on a local reference.

Transferring Nonremote Objects

Consider the `String` parameter of the `getPrice` method. The string value needs to be copied from the client to the server. It is not difficult to imagine how a copy of a string can be transported across a network. The RMI mechanism can also make copies of more complex objects, provided they are *serializable*. RMI uses the serialization mechanism described in Chapter 1 to send objects across a network connection. This means that any classes that implement the `Serializable` interface can be used as parameter or return types.

Passing parameters by serializing them has a subtle effect on the semantics of remote methods. When you pass objects into a local method, object *references* are transferred. When the method applies a mutator method to a parameter object, the caller will observe that change. But if a remote method mutates a serialized parameter, it changes the copy, and the caller will never notice.

To summarize, there are two mechanisms for transferring values between virtual machines.

- Objects of classes that implement the `Remote` interface are transferred as remote references.
- Objects of classes that implement the `Serializable` interface but not the `Remote` interface are copied using serialization.

All of this is automatic and requires no programmer intervention. Keep in mind that serialization can be slow for large objects, and that the remote method cannot mutate serialized parameters. You can, of course, avoid these issues by passing around remote references. That too comes at a cost: Invoking methods on remote references is far more expensive than calling local methods. Being aware of these costs allows you to make informed choices when designing remote services.

Note

✓ Remote objects are garbage-collected automatically, just as local objects are. However, the distributed collector is signifcantly more complex. When the local garbage collector finds that there are further local uses of a remote reference, it notifies the distributed collector that the server is no longer referenced by this client. When a server is no longer used by any clients, it is marked as garbage.

Our next example program will illustrate the transfer of remote and serializable objects. We change the `Warehouse` interface as shown in Listing 10-5. Given a list of keywords, the warehouse returns the `Product` that is the best match.

Listing 10-5. `Warehouse.java`

```
import java.rmi.*;
import java.util.*;

/**
   The remote interface for a simple warehouse.
   @version 1.0 2007-10-09
   @author Cay Horstmann
*/
public interface Warehouse extends Remote
{
   double getPrice(String description) throws RemoteException;
   Product getProduct(List<String> keywords) throws RemoteException;
}
```

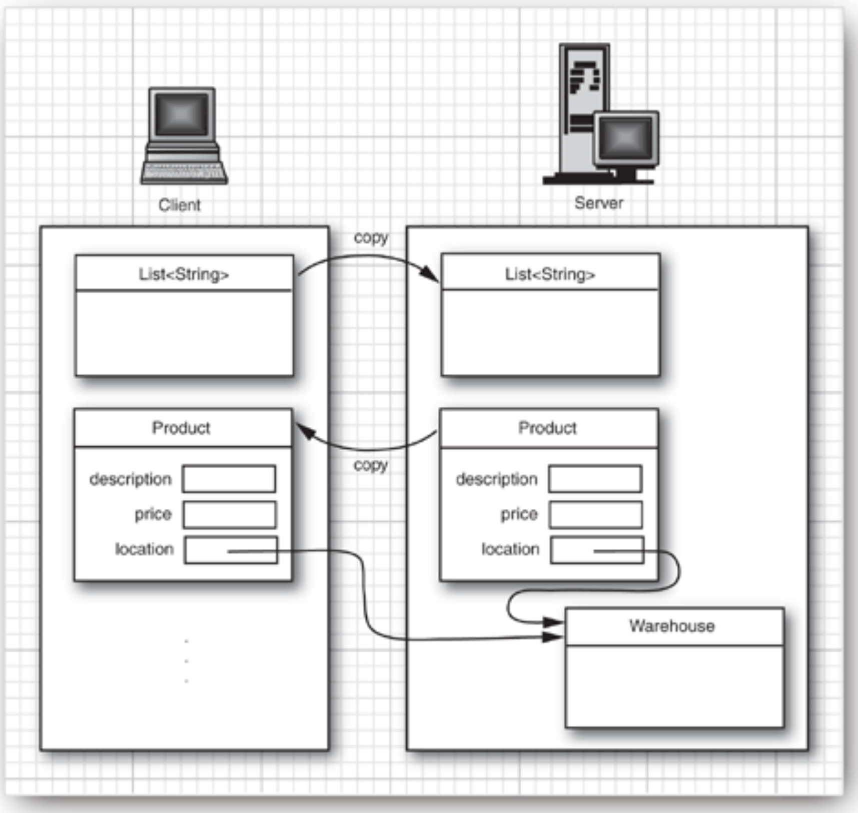
The parameter of the `getProduct` method has type `List<String>`. A parameter value must belong to a serializable class that implements the `List<String>` interface, such as `ArrayList<String>`. (Our sample client passes a value that is obtained by a call to `Arrays.asList`. Fortunately, that method is guaranteed to return a serializable list as well.)

The return type `Product` encapsulates the description, price, and location of the productâ€"see Listing 10-6.

Note that the `Product` class is serializable. The server constructs a `Product` object, and the client gets a copy (see Figure 10-7).

Figure 10-7. Copying local parameter and result objects

[View full size image]



Listing 10-6. `Product.java`

Code View:

```
import java.io.*;

public class Product implements Serializable
{
   public Product(String description, double price)
   {
      this.description = description;
      this.price = price;
   }

   public String getDescription()
   {
      return description;
   }

   public double getPrice()
   {
      return price;
   }

   public Warehouse getLocation()
   {
      return location;
   }

   public void setLocation(Warehouse location)
   {
      this.location = location;
   }

   private String description;
```

```
    private double price;
    private Warehouse location;
}
```

However, there is a subtlety. The `Product` class has an instance field of type `Warehouse`, a remote interface. The warehouse object is *not* serialized, which is just as well as it might have a huge amount of state. Instead, the client receives a stub to a remote `Warehouse` object. That stub might be different from the `centralWarehouse` stub on which the `getProduct` method was called. In our implementation, we will have two kinds of products, toasters and books, that are located in different warehouses.

Dynamic Class Loading

There is another subtlety to our next sample program. A list of keyword strings is sent to the server, and the warehouse returns an instance of a class `Product`. Of course, the client program will need the class file `Product.class` to compile. However, whenever our server program cannot find a match for the keywords, it returns the one product that is sure to delight everyone: the Core Java book. That object is an instance of the `Book` class, a subclass of `Product`.

When the client was compiled, it might have never seen the `Book` class. Yet when it runs, it needs to be able to execute `Book` methods that override `Product` methods. This demonstrates that the client needs to have the capability of loading additional classes at runtime. The client uses the same mechanism as the RMI registry. Classes are served by a web server, the RMI server class communicates the URL to the client, and the client makes an HTTP request to download the class files.

Whenever a program loads new code from another network location, there is a security issue. For that reason, you need to use a *security manager* in RMI applications that dynamically load classes. (See Chapter 9 for more information on class loaders and security managers.)

Programs that use RMI should install a security manager to control the activities of the dynamically loaded classes. You install it with the instruction

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

Note

✔ If all classes are available locally, then you do not actually need a security manager. If you know all class files of your program at deployment time, you can deploy them all locally. However, it often happens that the client or server program evolves and new classes are added over time. Then you benefit from dynamic class loading. Any time you load code from another source, you need a security manager.

By default, the `SecurityManager` restricts all code in the program from establishing network connections. However, the program needs to make network connections to three remote locations:

- The web server that loads remote classes.
- The RMI registry.
- Remote objects.

To allow these operations, you supply a policy file. (We discussed policy files in greater detail in Chapter

9.) Here is a policy file that allows an application to make any network connection to a port with port number of at least 1024. (The RMI port is 1099 by default, and the remote objects also use ports $\geq$ 1024. We use port 8080 for dowloading classes.)

```
grant
{
   permission java.net.SocketPermission
      "*:1024-65535", "connect";
};
```

You need to instruct the security manager to read the policy file by setting the `java.security.policy` property to the file name. You can use a call such as

```
System.setProperty("java.security.policy", "rmi.policy");
```

Alternatively, you can specify the system property setting on the command line:

```
-Djava.security.policy=rmi.policy
```

To run the sample application, be sure that you have killed the RMI registry, web server, and the server program from the preceding sample. Open four console windows and follow these steps.

1. Compile the source files for the interface, implementation, client, and server classes.

   ```
   javac *.java
   ```

2. Make three directories, `client`, `server`, and `download`, and populate them as follows:

   ```
   client/
      WarehouseClient.class
      Warehouse.class
      Product.class
      client.policy
   server/
      Warehouse.class
      Product.class
      Book.class
      WarehouseImpl.class
      WarehouseServer.class
      server.policy
   download
      Warehouse.class
      Product.class
      Book.class
   ```

3. In the first console window, change to a directory that has *no* class files. Start the RMI registry.

4. In the second console window, change to the `download` directory and start `NanoHTTPD`.

5. In the third console window, change to the `server` directory and start the server.

   ```
   java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
   ```

6. In the fourth console window, change to the `client` directory and run the client.

```
java WarehouseClient
```

Listing 10-7 shows the code of the `Book` class. Note that the `getDescription` method is overridden to show the ISBN. When the client program runs, it shows the ISBN for the Core Java book, which proves that the `Book` class was loaded dynamically. Listing 10-8 shows the warehouse implementation. A warehouse has a reference to a backup warehouse. If an item cannot be found in the warehouse, the backup warehouse is searched. Listing 10-9 shows the server program. Only the central warehouse is entered into the RMI registry. Note that a remote reference to the backup warehouse can be passed to the client even though it is not included in the RMI registry. This happens whenever no keyword matches and a *Core Java* book (whose `location` field references the backup warehouse) is sent to the client.

Listing 10-7. `Book.java`

```java
/**
 * A book is a product with an ISBN number.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class Book extends Product
{
   public Book(String title, String isbn, double price)
   {
      super(title, price);
      this.isbn = isbn;
   }

   public String getDescription()
   {
      return super.getDescription() + " " + isbn;
   }

   private String isbn;
}
```

Listing 10-8. `WarehouseImpl.java`

Code View:

```java
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

/**
 * This class is the implementation for the remote Warehouse interface.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
   /**
    * Constructs a warehouse implementation.
    */
   public WarehouseImpl(Warehouse backup) throws RemoteException
   {
      products = new HashMap<String, Product>();
      this.backup = backup;
   }

   public void add(String keyword, Product product)
   {
      product.setLocation(this);
      products.put(keyword, product);
   }

   public double getPrice(String description) throws RemoteException
   {
      for (Product p : products.values())
         if (p.getDescription().equals(description)) return p.getPrice();
      if (backup == null) return 0;
      else return backup.getPrice(description);
   }

   public Product getProduct(List<String> keywords) throws RemoteException
   {
      for (String keyword : keywords)
      {
         Product p = products.get(keyword);
         if (p != null) return p;
      }
      if (backup != null)
         return backup.getProduct(keywords);
      else if (products.values().size() > 0)
         return products.values().iterator().next();
      else
         return null;
   }

   private Map<String, Product> products;
   private Warehouse backup;
}
```

Listing 10-9. `WarehouseServer.java`

Code View:

```
import java.rmi.*;
import javax.naming.*;

/**
  * This server program instantiates a remote warehouse objects, registers it with the naming
  * service, and waits for clients to invoke methods.
  * @version 1.12 2007-10-09
  * @author Cay Horstmann
  */

public class WarehouseServer
{
   public static void main(String[] args) throws RemoteException, NamingException
   {
      System.setProperty("java.security.policy", "server.policy");
      System.setSecurityManager(new SecurityManager());

      System.out.println("Constructing server implementation...");
      WarehouseImpl backupWarehouse = new WarehouseImpl(null);
      WarehouseImpl centralWarehouse = new WarehouseImpl(backupWarehouse);

      centralWarehouse.add("toaster", new Product("Blackwell Toaster", 23.95));
      backupWarehouse.add("java", new Book("Core Java vol. 2", "0132354799", 44.95));

      System.out.println("Binding server implementation to registry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:central_warehouse", centralWarehouse);

      System.out.println("Waiting for invocations from clients...");
   }
}
```

Remote References with Multiple Interfaces

A remote class can implement multiple interfaces. Consider a remote interface `ServiceCenter`.

```
public interface ServiceCenter extends Remote
{
   int getReturnAuthorization(Product prod) throws RemoteException;
}
```

Now suppose a `WarehouseImpl` class implements this interface as well as the `Warehouse` interface. When a remote reference to such a service center is transferred to another virtual machine, the recipient obtains a stub that has access to the remote methods in both the `ServiceCenter` and the `Warehouse` interface. You can use the `instanceof` operator to find out whether a particular remote object implements an interface. Suppose you receive a remote object through a variable of type `Warehouse`.

```
Warehouse location = product.getLocation();
```

The remote object might or might not be a service center. To find out, use the test

```
if (location instanceof ServiceCenter)
```

If the test passes, you can cast `location` to the `ServiceCenter` type and invoke the `getReturnAuthorization` method.

Remote Objects and the `equals`, `hashCode`, and `clone` Methods

Objects inserted in sets must override the `equals` method. In the case of a hash set or hash map, the `hashCode` method must be defined as well. However, there is a problem when trying to compare remote objects. To find out if two remote objects have the same contents, the call to `equals` would need to contact the servers containing the objects and compare their contents. Like any remote call, that call could fail. But the `equals` method in the class `Object` is not declared to throw a `RemoteException`, whereas all methods in a remote interface must throw that exception. Because a subclass method cannot throw more exceptions than the superclass method it replaces, you cannot define an `equals` method in a remote interface. The same holds for `hashCode`.

Instead, the `equals` and `hashCode` methods on stub objects simply look at the location of the remote objects. The `equals` method deems two stubs equal if they refer to the same remote object. Two stubs that refer to different remote objects are never equal, even if those objects have identical contents. Similarly, the hash code is computed only from the object identifier.

For the same technical reasons, remote references do not have a `clone` method. If `clone` were to make a remote call to tell the server to clone the implementation object, then the `clone` method would need to throw a `RemoteException`. However, the `clone` method in the `Object` superclass promised never to throw any exception other than `CloneNotSupportedException`.

To summarize, you can use remote references in sets and hash tables, but you must remember that equality testing and hashing do not take into account the contents of the remote objects. You simply cannot clone remote references.

Remote Object Activation

In the preceding sample programs, we used a server program to instantiate and register objects so that clients could make remote calls on them. However, in some cases, it might be wasteful to instantiate lots of remote objects and have them wait for connections, whether or not client objects use them. The *activation* mechanism lets you delay the object construction so that a remote object is only constructed when at least one client invokes a remote method on it.

To take advantage of activation, the client code is completely unchanged. The client simply requests a remote reference and makes calls through it.

However, the server program is replaced by an activation program that constructs *activation descriptors* of the objects that are to be constructed at a later time, and binds receivers for remote method calls with the naming service. When a call is made for the first time, the information in the activation descriptor is used to construct the object.

A remote object that is used in this way should extend the `Activatable` class instead of the `UnicastRemoteObject` class. Of course, it also implements one or more remote interfaces. For example,

```
class WarehouseImpl
   extends Activatable
   implements Warehouse
{
   . . .
}
```

Because the object construction is delayed until a later time, it must happen in a standardized form. Therefore, you must provide a constructor that takes two parameters:

- An activation ID (which you simply pass to the superclass constructor).
- A single object containing all construction information, wrapped in a `MarshalledObject`.

If you need multiple construction parameters, you must package them into a single object. You can always use an `Object[]` array or an `ArrayList` for this purpose.

When you build the activation descriptor, you will construct a `MarshalledObject` from the construction information like this:

```
MarshalledObject<T> param = new MarshalledObject<T>(constructionInfo);
```

In the constructor of the implementation object, use the `get` method of the `MarshalledObject` class to obtain the deserialized construction information.

```
T constructionInfo = param.get();
```

To demonstrate activation, we modify the `WarehouseImpl` class so that the construction information is a map of descriptions and prices. That information is wrapped into a `MarshalledObject` and unwrapped in the constructor:

Code View:

```
public WarehouseImpl(ActivationID id, MarshalledObject<Map<String, Double>> param)
      throws RemoteException, ClassNotFoundException, IOException
{
   super(id, 0);
   prices = param.get();
   System.out.println("Warehouse implementation constructed.");
}
```

By passing 0 as the second parameter of the superclass constructor, we indicate that the RMI library should assign a suitable port number to the listener port.

This constructor prints a message so that you can see that the warehouse object is activated on demand.

Note

✓

Your remote objects don't actually have to extend the `Activatable` class. If they don't, then place the static method call

```
Activatable.exportObject(this, id, 0)
```

in the constructor of the server class.

Now let us turn to the activation program. First, you need to define an activation group. An activation group describes common parameters for launching the virtual machine that contains the remote objects. The most important parameter is the security policy.

Construct an activation group descriptor as follows:

```
Properties props = new Properties();
props.put("java.security.policy", "/path/to/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);
```

The second parameter describes special command options. We don't need any for this example, so we pass a `null` reference.

Next, create a group ID with the call

```
ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);
```

Now you are ready to construct activation descriptors. For each object that should be constructed on demand, you need the following:

- The activation group ID for the virtual machine in which the object should be constructed.
- The name of the class (such as `"WarehouseImpl"` or `"com.mycompany.MyClassImpl"`).
- The URL string from which to load the class files. This should be the base URL, not including package paths.
- The marshalled construction information.

For example,

Code View:

```
MarshalledObject param = new MarshalledObject(constructionInfo);
ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl",
                                         "http://myserver.com/download/", param);
```

Pass the descriptor to the static `Activatable.register` method. It returns an object of some class that implements the remote interfaces of the implementation class. You can bind that object with the naming service:

```
Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

Unlike the server programs of the preceding examples, the activation program exits after registering and binding the activation receivers. The remote objects are constructed only when the first remote method call occurs.

Listings 10-10 and 10-11 show the code for the activation program and the activatable warehouse implementation. The warehouse interface and the client program are unchanged.

To launch this program, follow these steps:

1. Compile all source files.

2. Distribute class files as follows:

   ```
   client/
      WarehouseClient.class
      Warehouse.class
   server/
      WarehouseActivator.class
      Warehouse.class
      WarehouseImpl.class
      server.policy
   download/
      Warehouse.class
      WarehouseImpl.class
   rmi/
      rmid.policy
   ```

3. Start the RMI registry in the `rmi` directory (which contains no class files).

4. Start the RMI activation daemon in the `rmi` directory.

   ```
   rmid -J-Djava.security.policy=rmid.policy
   ```

   The `rmid` program listens to activation requests and activates objects in a separate virtual machine. To launch a virtual machine, the `rmid` program needs certain permissions. These are specified in a policy file (see Listing 10-12). You use the `-J` option to pass an option to the virtual machine running the activation daemon.

5. Start the `NanoHTTPD` web server in the `download` directory.

6. Run the activation program from the `server` directory.

   ```
   java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseActivator
   ```

   The program exits after the activation receivers have been registered with the naming service. (You might wonder why you need to specify the codebase as it is also provided in the constructor of the activation descriptor. However, that information is only processed by the RMI activation daemon. The RMI registry still needs the codebase to load the remote interface classes.)

7. Run the client program from the `client` directory.

```
java WarehouseClient
```

The client will print the familiar product description. When you run the client for the first time, you will also see the constructor messages in the shell window of the activation daemon.

Listing 10-10. `WarehouseActivator.java`

Code View:

```
import java.io.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;
import javax.naming.*;

/**
  * This server program instantiates a remote warehouse object, registers it with the naming
  * service, and waits for clients to invoke methods.
  * @version 1.12 2007-10-09
  * @author Cay Horstmann
  */

public class WarehouseActivator
{
   public static void main(String[] args) throws RemoteException, NamingException,
         ActivationException, IOException
   {
      System.out.println("Constructing activation descriptors...");

      Properties props = new Properties();
      // use the server.policy file in the current directory
      props.put("java.security.policy", new File("server.policy").getCanonicalPath());
      ActivationGroupDesc group = new ActivationGroupDesc(props, null);
      ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);

      Map<String, Double> prices = new HashMap<String, Double>();
      prices.put("Blackwell Toaster", 24.95);
      prices.put("ZapXpress Microwave Oven", 49.95);

      MarshalledObject<Map<String, Double>> param = new MarshalledObject<Map<String, Double>>(
            prices);

      String codebase = "http://localhost:8080/";

      ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl", codebase, param);

      Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);

      System.out.println("Binding activable implementation to registry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:central_warehouse", centralWarehouse);
      System.out.println("Exiting...");
   }
}
```

Listing 10-11. `WarehouseImpl.java`

Code View:

```
import java.io.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;

/**
 * This class is the implementation for the remote Warehouse interface.
 * @version 1.0 2007-10-20
 * @author Cay Horstmann
 */
public class WarehouseImpl extends Activatable implements Warehouse
{
   public WarehouseImpl(ActivationID id, MarshalledObject<Map<String, Double>> param)
         throws RemoteException, ClassNotFoundException, IOException
   {
      super(id, 0);
      prices = param.get();
      System.out.println("Warehouse implementation constructed.");
   }

   public double getPrice(String description) throws RemoteException
   {
      Double price = prices.get(description);
      return price == null ? 0 : price;
   }

   private Map<String, Double> prices;
}
```

Listing 10-12. `rmid.policy`

```
grant
{
   permission com.sun.rmi.rmid.ExecPermission
      "${java.home}${/}bin${/}java";
   permission com.sun.rmi.rmid.ExecOptionPermission
      "-Djava.security.policy=*";
}
```

API `java.rmi.activation.Activatable` 1.2

- `protected Activatable(ActivationID id, int port)`

  constructs the activatable object and establishes a listener on the given port. Use 0 for the port to have a port assigned automatically.

- `static Remote exportObject(Remote obj, ActivationID id, int port)`

  makes a remote object activatable. Returns the activation receiver that should be made available to remote callers. Use 0 for the port to have a port assigned automatically.

- `static Remote register(ActivationDesc desc)`

  registers the descriptor for an activatable object and prepares it for receiving remote calls. Returns the activation receiver that should be made available to remote callers.

API java.rmi.MarshalledObject 1.2

- `MarshalledObject(Object obj)`

  constructs an object containing the serialized data of a given object.

- `Object get()`

  deserializes the stored object data and returns the object.

API java.rmi.activation.ActivationGroupDesc 1.2

- `ActivationGroupDesc(Properties props, ActivationGroupDesc.CommandEnvironment env)`

  constructs an activation group descriptor that specifies virtual machine properties for a virtual machine that hosts activated objects. The `env` parameter contains the path to the virtual machine executable and command-line options, or it is `null` if no special settings are required.

API java.rmi.activation.ActivationGroup 1.2

- `static ActivationSystem getSystem()`

  returns a reference to the activation system.

API java.rmi.activation.ActivationSystem 1.2

- `ActivationGroupID registerGroup(ActivationGroupDesc group)`

  registers an activation group and returns the group ID.

API java.rmi.activation.ActivationDesc 1.2

- `ActivationDesc(ActivationGroupID id, String className, String classFileURL, MarshalledObject data)`

  constructs an activation descriptor.