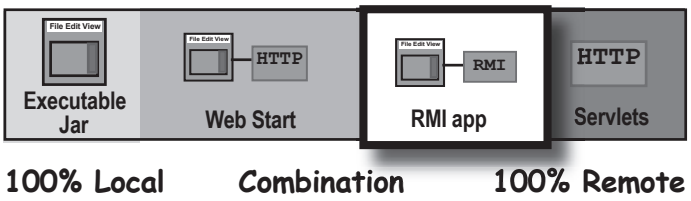
Distributed Computing

Being remote doesn't have to be a bad thing. Sure, things *are* easier when all the parts of your application are in one place, in one heap, with one JVM to rule them all. But that's not always possible. Or desirable. What if your application handles powerful computations, but the end-users are on a wimpy little Java-enabled device? What if your app needs data from a database, but for security reasons, only code on your server can access the database? Imagine a big e-commerce back-end, that has to run within a transaction-management system? Sometimes, part of your app *must* run on a server, while another part (usually a client) must run on a *different* machine. In this chapter, we'll learn to use Java's amazingly simple Remote Method Invocation (RMI) technology. We'll also take a quick peek at Servlets, Enterprise Java Beans (EJB) , and Jini, and look at the ways in which EJB and Jini *depend* on RMI. We'll end the book by writing one of the coolest things you can make in Java, a *universal service browser*.

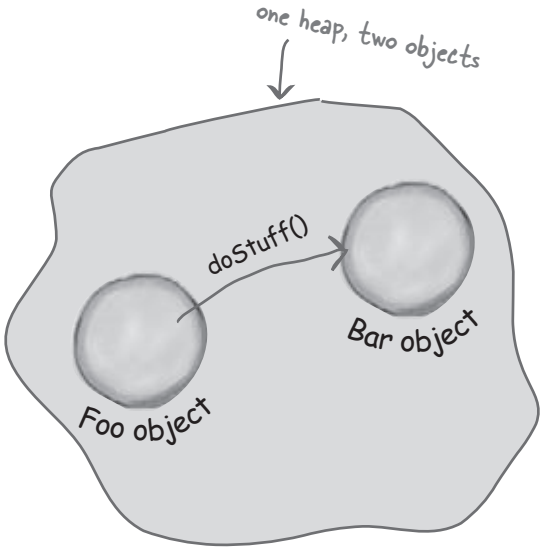


Method calls are always between two objects on the same heap.

So far in this book, every method we've invoked has been on an object running in the same virtual machine as the caller. In other words, the calling object and the callee (the object we're invoking the method on) live on the same heap.

```
class Foo {
    void go() {
        Bar b = new Bar();
        b.doStuff();
    }
    public static void main (String[] args) {
       Foo f = new Foo();
       f.go();
    }
}
```

In the code above, we know that the Foo instance referenced by *f* and the Bar object referenced by *b* are both on the same heap, run by the same JVM. Remember, the JVM is responsible for stuffing bits into the reference variable that represent *how to get to an object on the heap.* The JVM always knows where each object is, and how to get to it. But the JVM can know about references on only its *own* heap! You can't, for example, have a JVM running on one machine knowing about the heap space of a JVM running on a *different* machine. In fact, a JVM running on one machine can't know anything about a different JVM running on the *same* machine. It makes no difference if the JVMs are on the same or different physical machines; it matters only that the two JVMs are, well, two different invocations of the JVM.



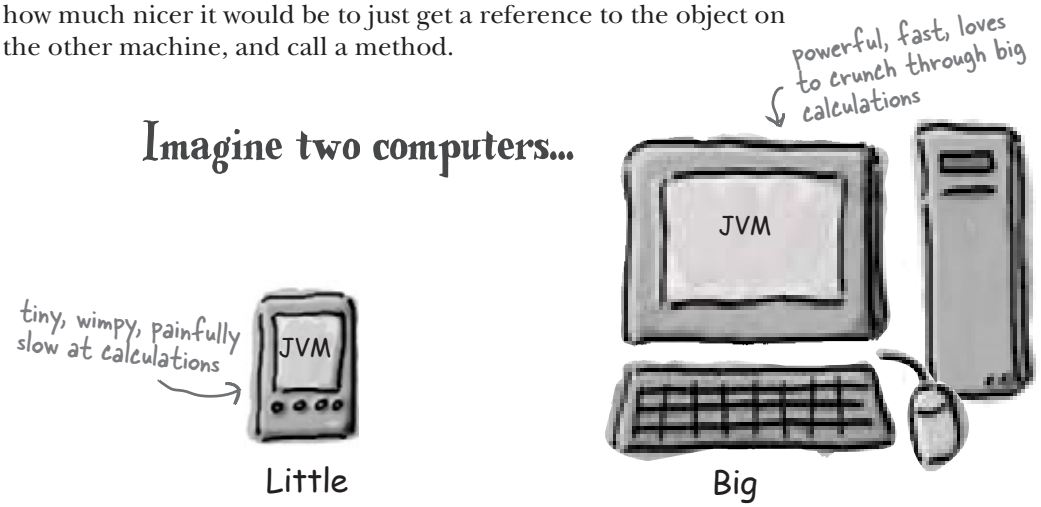
In most applications, when one object calls a method on another, both objects are on the same heap. In other words, both are running within the same JVM.

What if you want to invoke a method on an object running on another machine?

We know how to get information from one machine to another—with Sockets and I/O. We open a Socket connection to another machine, and get an OutputStream and write some data to it.

But what if we actually want to *call a method* on something running in another machine... another JVM? Of course we could always build our own protocol, and when you send data to a ServerSocket the server could parse it, figure out what you meant, do the work, and send back the result on another stream. What a pain, though. Think how much nicer it would be to just get a reference to the object on the other machine, and call a method.

Imagine two computers...



Big has something Little wants.
Compute power.

Little wants to send some data to Big, so that Big can do the heavy computing.
Little wants simply to call a method...

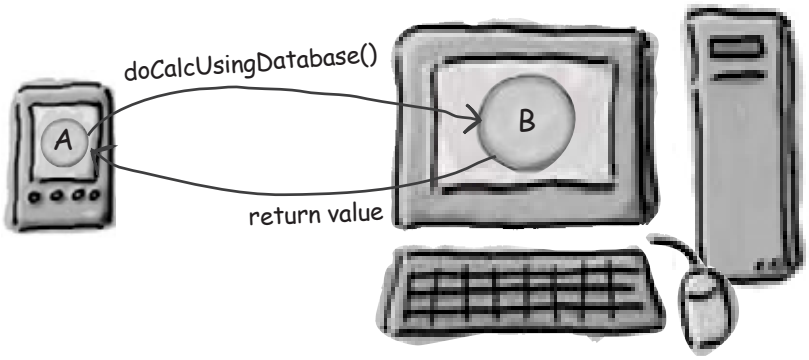
```
double doCalcUsingDatabase(CalcNumbers numbers)
```

and get back the result.

But how can Little get a reference to an object on Big?

Object A, running on Little, wants to call a method on Object B, running on Big.

The question is, how do we get an object on one machine (which means a different heap/JVM) to call a method on another machine?



But you can't do that.

Well, not directly anyway. You can't get a reference to something on another heap. If you say:

Dog d = ???

Whatever *d* is referencing must be in the same heap space as the code running the statement.

But imagine you want to design something that will use Sockets and I/O to communicate your intention (a method invocation on an object running on another machine), yet still *feel* as though you were making a local method call.

In other words, you want to cause a method invocation on a *remote* object (i.e., an object in a heap somewhere else), but with code that lets you *pretend* that you're invoking a method on a local object. The ease of a plain old everyday method call, but the power of remote method invocation. That's our goal.

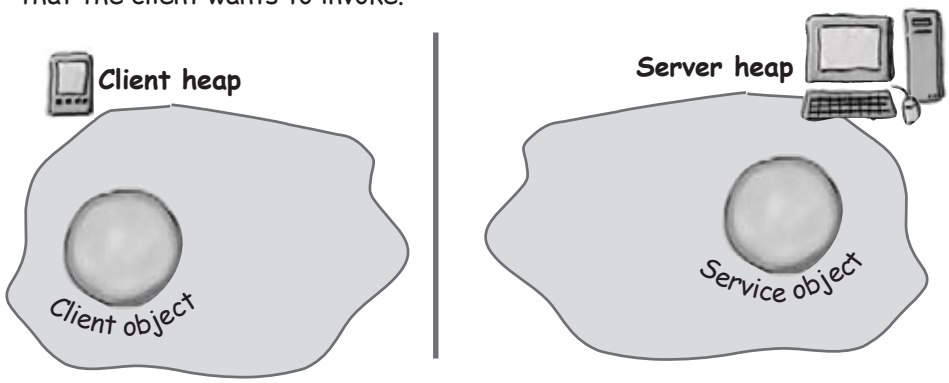
That's what RMI (Remote Method Invocation) gives you!

But let's step back and imagine how you would design RMI if you were doing it yourself. Understanding what you'd have to build yourself will help you learn how RMI works.

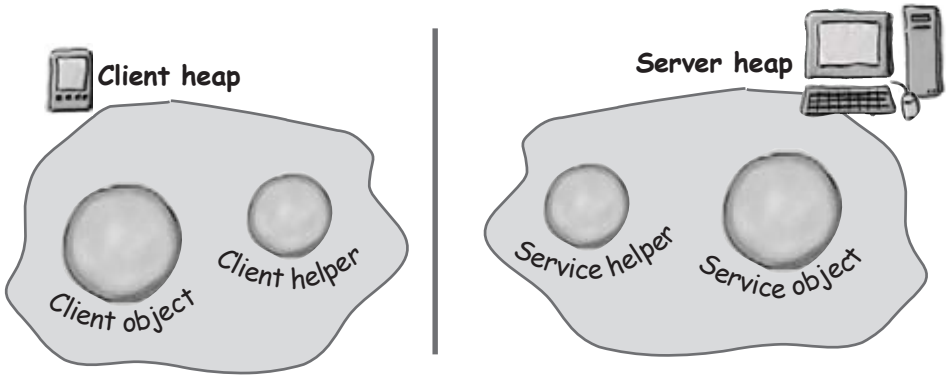
A design for remote method calls

Create four things: server, client, server helper, client helper

1. Create client and server apps. The server app is the **remote service** that has an object with the method that the client wants to invoke.



2. Create client and server 'helpers'. They'll handle all the low-level networking and I/O details so your client and service can pretend like they're in the same heap.



The role of the 'helpers'

The 'helpers' are the objects that actually do the communicating. They make it possible for the client to *act* as though its calling a method on a local object. In fact, it *is*. The client calls a method on the client helper, *as if the client helper were the actual service. The client helper is a proxy for the Real Thing.*

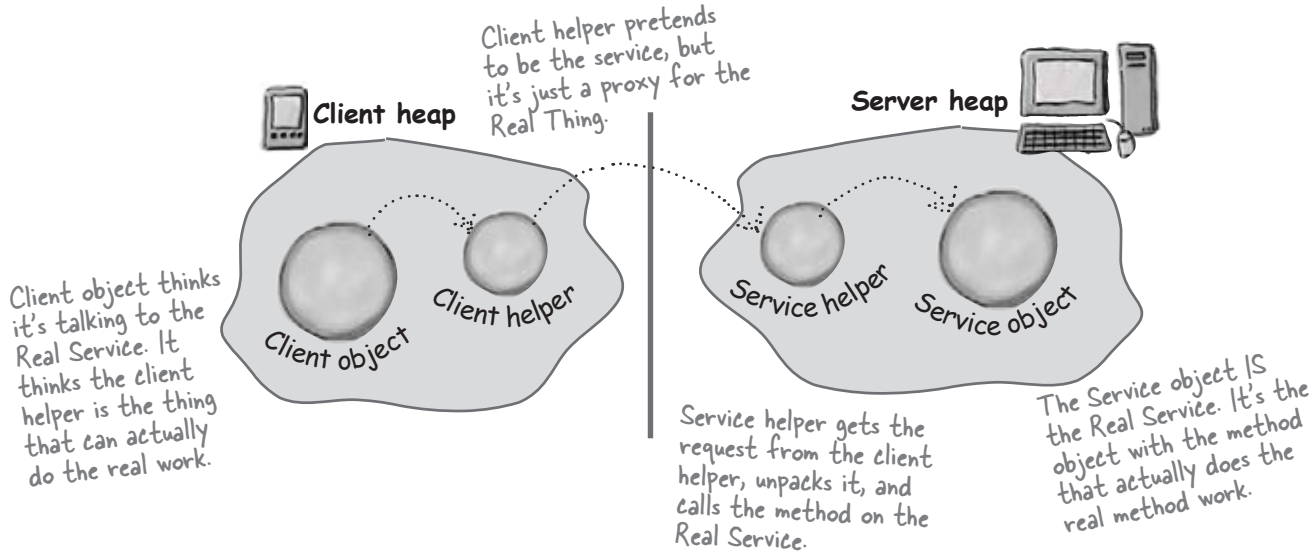
In other words, the client object *thinks* it's calling a method on the remote service, because the client helper is *pretending* to be the service object. ***Pretending to be the thing with the method the client wants to call!***

But the client helper isn't really the remote service. Although the client helper *acts* like it (because it has the same method that the service is advertising), the client helper doesn't have any of the actual method logic the client is expecting. Instead, the client helper contacts the server, transfers information about the method call (e.g., name of the method, arguments, etc.), and waits for a return from the server.

On the server side, the service helper receives the request from the client helper (through a Socket connection), unpacks the information about the call, and then invokes the *real* method on the *real* service object. So to the service object, the call is local. It's coming from the service helper, not a remote client.

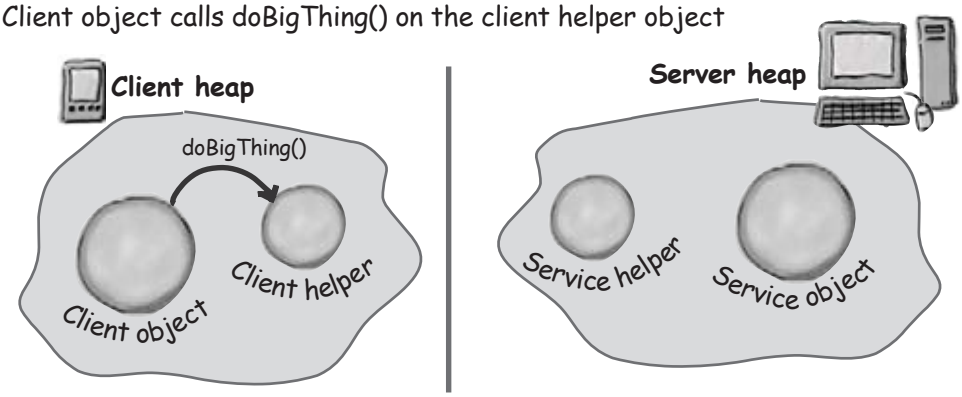
The service helper gets the return value from the service, packs it up, and ships it back (over a Socket's output stream) to the client helper. The client helper unpacks the information and returns the value to the client object.

Your client object gets to act like it's making remote method calls. But what it's really doing is calling methods on a heap-local 'proxy' object that handles all the low-level details of Sockets and streams.

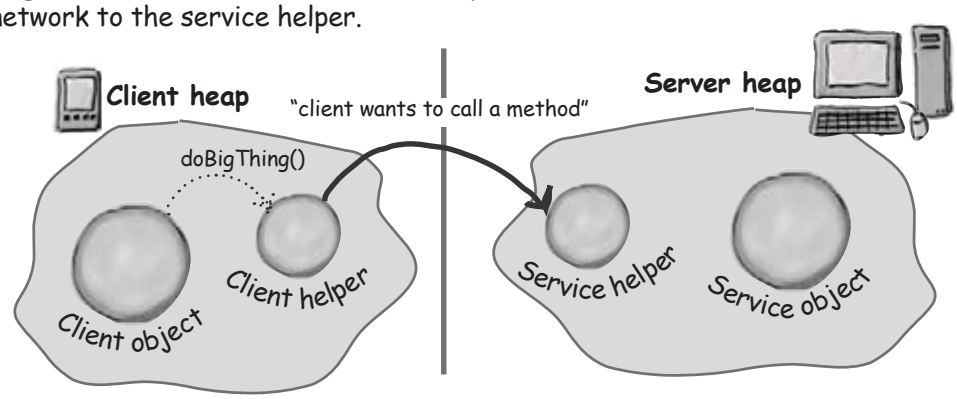


How the method call happens

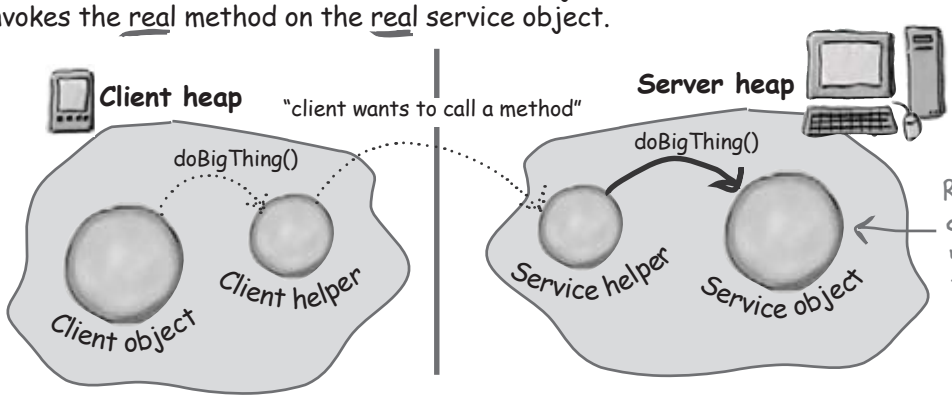
1. Client object calls doBigThing() on the client helper object



2. Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.



3. Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the real method on the real service object.



Java RMI gives you the client and service helper objects!

In Java, RMI builds the client and service helper objects for you, and it even knows how to make the client helper look like the Real Service. In other words, RMI knows how to give the client helper object the same methods you want to call on the remote service.

Plus, RMI provides all the runtime infrastructure to make it work, including a lookup service so that the client can find and get the client helper (the proxy for the Real Service).

With RMI, you don't write *any* of the networking or I/O code yourself. The client gets to call remote methods (i.e. the ones the Real Service has) just like normal method calls on objects running in the client's own local JVM.

Almost.

There is one difference between RMI calls and local (normal) method calls. Remember that even though to the client it looks like the method call is local, the client helper sends the method call across the network. So there is networking and I/O. And what do we know about networking and I/O methods?

They're risky!

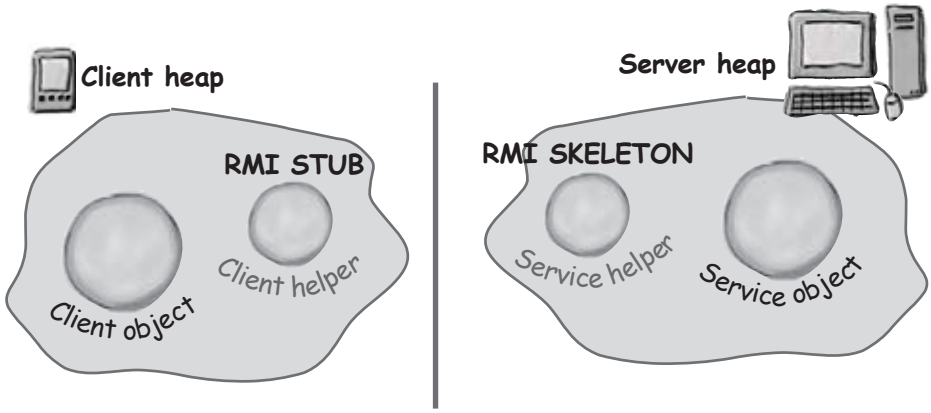
They throw exceptions all over the place.

So, the client does have to acknowledge the risk. The client has to acknowledge that when it calls a remote method, even though to the client it's just a local call to the proxy/helper object, the call *ultimately* involves Sockets and streams. The client's original call is *local*, but the proxy turns it into a *remote* call. A remote call just means a method that's invoked on an object on another JVM. *How* the information about that call gets transferred from one JVM to another depends on the protocol used by the helper objects.

With RMI, you have a choice of protocols: JRMP or IIOP. JRMP is RMI's 'native' protocol, the one made just for Java-to-Java remote calls. IIOP, on the other hand, is the protocol for CORBA (Common Object Request Broker Architecture), and lets you make remote calls on things which aren't necessarily Java objects. CORBA is usually *much* more painful than RMI, because if you don't have Java on both ends, there's an awful lot of translation and conversion that has to happen.

But thankfully, all we care about is Java-to-Java, so we're sticking with plain old, remarkably easy RMI.

In RMI, the client helper is a 'stub' and the server helper is a 'skeleton'.



Making the Remote Service

This is an **overview** of the five steps for making the remote service (that runs on the server). Don't worry, each step is explained in detail over the next few pages.

Step one:

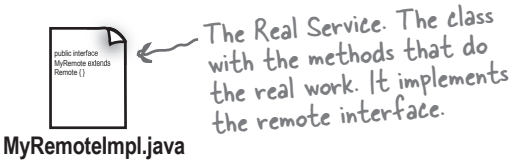
Make a Remote Interface

The remote interface defines the methods that a client can call remotely. It's what the client will use as the polymorphic class type for your service. Both the Stub and actual service will implement this!

Step two:

Make a Remote Implementation

This is the class that does the Real Work. It has the real implementation of the remote methods defined in the remote interface. It's the object that the client wants to call methods on.



Step three:

Generate the stubs and skeletons using rmic

These are the client and server 'helpers'. You don't have to create these classes or ever look at the source code that generates them. It's all handled automatically when you run the rmic tool that ships with your Java development kit.

Step four:

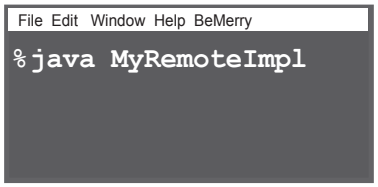
Start the RMI registry (rmiregistry)

The *rmiregistry* is like the white pages of a phone book. It's where the user goes to get the proxy (the client stub/helper object).

Step five:

Start the remote service

You have to get the service object up and running. Your service implementation class instantiates an instance of the service and registers it with the RMI registry. Registering it makes the service available for clients.



Step one: Make a Remote Interface

MyRemote.java

1 Extend java.rmi.Remote

Remote is a 'marker' interface, which means it has no methods. It has special meaning for RMI, though, so you must follow this rule. Notice that we say 'extends' here. One interface is allowed to *extend* another interface.

```
public interface MyRemote extends Remote {
```

Your interface has to announce that it's for remote method calls. An interface can't implement anything, but it can extend other interfaces.

2 Declare that all methods throw a RemoteException

The remote interface is the one the client uses as the polymorphic type for the service. In other words, the client invokes methods on something that implements the remote interface. That something is the stub, of course, and since the stub is doing networking and I/O, all kinds of Bad Things can happen. The client has to acknowledge the risks by handling or declaring the remote exceptions. If the methods in an interface declare exceptions, any code calling methods on a reference of that type (the interface type) must handle or declare the exceptions.

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

the Remote interface is in java.rmi

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

Every remote method call is considered 'risky'. Declaring RemoteException on every method forces the client to pay attention and acknowledge that things might not work.

3 Be sure arguments and return values are primitives or Serializable

Arguments and return values of a remote method must be either primitive or Serializable. Think about it. Any argument to a remote method has to be packaged up and shipped across the network, and that's done through Serialization. Same thing with return values. If you use primitives, Strings, and the majority of types in the API (including arrays and collections), you'll be fine. If you are passing around your own types, just be sure that you make your classes implement Serializable.

```
public String sayHello() throws RemoteException;
```

This return value is gonna be shipped over the wire from the server back to the client, so it must be Serializable. That's how args and return values get packaged up and sent.

Step two: Make a Remote Implementation

MyRemoteImpl.java

① Implement the Remote interface

Your service has to implement the remote interface—the one with the methods your client is going to call.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    // more code in class
}
```

The compiler will make sure that you've implemented all the methods from the interface you implement. In this case, there's only one.

② Extend UnicastRemoteObject

In order to work as a remote service object, your object needs some functionality related to 'being remote'. The simplest way is to extend UnicastRemoteObject (from the java.rmi.server package) and let that class (your superclass) do the work for you.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

③ Write a no-arg constructor that declares a RemoteException

Your new superclass, UnicastRemoteObject, has one little problem—its constructor throws a RemoteException. The only way to deal with this is to declare a constructor for your remote implementation, just so that you have a place to declare the RemoteException. Remember, when a class is instantiated, its superclass constructor is always called. If your superclass constructor throws an exception, you have no choice but to declare that your constructor also throws an exception.

```
public MyRemoteImpl() throws RemoteException { }
```

You don't have to put anything in the constructor. You just need a way to declare that your superclass constructor throws an exception.

④ Register the service with the RMI registry

Now that you've got a remote service, you have to make it available to remote clients. You do this by instantiating it and putting it into the RMI registry (which must be running or this line of code fails). When you register the implementation object, the RMI system actually puts the *stub* in the registry, since that's what the client really needs. Register your service using the static rebind() method of the java.rmi.Naming class.

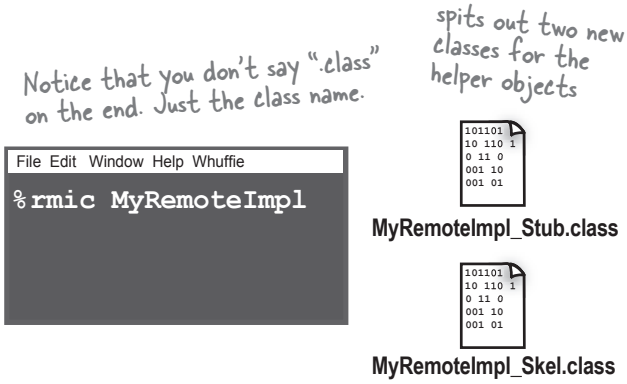
```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("Remote Hello", service);
} catch(Exception ex) {...}
```

Give your service a name (that clients can use to look it up in the registry) and register it with the RMI registry. When you bind the service object, RMI swaps the service for the stub and puts the stub in the registry.

Step three: generate stubs and skeletons

1 Run rmic on the remote implementation class (not the remote interface)

The rmic tool, that comes with the Java software development kit, takes a service implementation and creates two new classes, the stub and the skeleton. It uses a naming convention that is the name of your remote implementation, with either _Stub or _Skeleton added to the end. There are other options with rmic, including not generating skeletons, seeing what the source code for these classes looked like, and even using IIOP as the protocol. The way we're doing it here is the way you'll usually do it. The classes will land in the current directory (i.e. whatever you did a cd to). Remember, *rmic* must be able to see your implementation class, so you'll probably run rmic from the directory where your remote implementation is. (We're deliberately not using packages here, to make it simpler. In the Real World, you'll need to account for package directory structures and fully-qualified names).



Step four: run rmiregistry

1 Bring up a terminal and start the rmiregistry.

Be sure you start it from a directory that has access to your classes. The simplest way is to start it from your 'classes' directory.

Step five: start the service

1 Bring up another terminal and start your service

This might be from a main() method in your remote implementation class, or from a separate launcher class. In this simple example, we put the starter code in the implementation class, in a main method that instantiates the object and registers it with RMI registry.

Complete code for the server side

The Remote interface:

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

RemoteException and Remote interface are in java.rmi package

Your interface MUST extend java.rmi.Remote

All of your remote methods must declare a RemoteException

The Remote service (the implementation):

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

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

   public String sayHello() {
       return "Server says, 'Hey'";
   }

   public MyRemoteImpl() throws RemoteException { }

   public static void main (String[] args) {

       try {
           MyRemote service = new MyRemoteImpl();
           Naming.rebind("Remote Hello", service);
       } catch(Exception ex) {
           ex.printStackTrace();
       }
   }
}
```

UnicastRemoteObject is in the java.rmi.server package

extending UnicastRemoteObject is the easiest way to make a remote object

You have to implement all the interface methods, of course. But notice that you do NOT have to declare the RemoteException.

you MUST implement your remote interface!!

your superclass constructor (for UnicastRemoteObject) declares an exception, so YOU must write a constructor, because it means that your constructor is calling risky code (its super constructor)

Make the remote object, then 'bind' it to the rmiregistry using the static Naming.rebind(). The name you register it under is the name clients will need to look it up in the rmi registry.

How does the client get the stub object?

The client has to get the stub object, since that's the thing the client will call methods on. And that's where the RMI registry comes in. The client does a 'lookup', like going to the white pages of a phone book, and essentially says, "Here's a name, and I'd like the stub that goes with that name."

lookup() is a static method of the Naming class

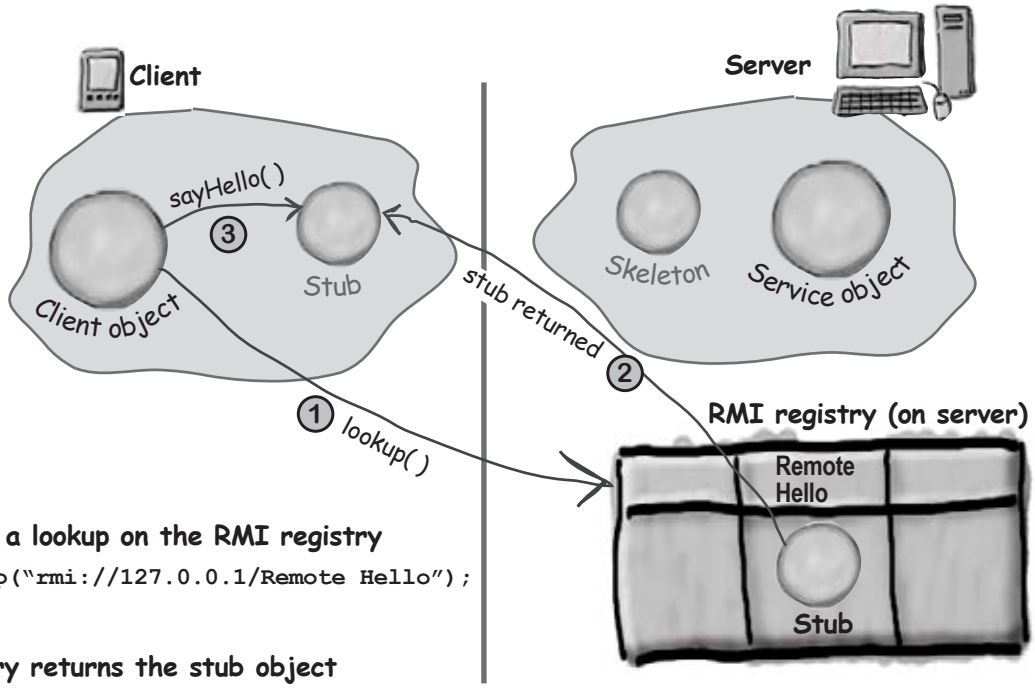
This must be the name that the service was registered under

```
MyRemote service = (MyRemote)   Naming.lookup("rmi://127.0.0.1/Remote Hello");
```

The client always uses the remote implementation as the type of the service. In fact, the client never needs to know the actual class name of your remote service.

You have to cast it to the interface, since the lookup method returns type Object.

your host name or IP address goes here



1 Client does a lookup on the RMI registry

```
Naming.lookup("rmi://127.0.0.1/Remote Hello");
```

2 RMI registry returns the stub object

(as the return value of the lookup method) and RMI deserializes the stub automatically. You MUST have the stub class (that rmic generated for you) on the client or the stub won't be deserialized.

3 Client invokes a method on the stub, as though the stub IS the real service

How does the client get the stub class?

Now we get to the interesting question. Somehow, someway, the client must have the stub class (that you generated earlier using rmic) at the time the client does the lookup, or else the stub won't be deserialized on the client and the whole thing blows up. In a simple system, you can simply hand-deliver the stub class to the client.

There's a much cooler way, though, although it's beyond the scope of this book. But just in case you're interested, the cooler way is called "dynamic class downloading". With dynamic class downloading, a stub object (or really any Serialized object) is 'stamped' with a URL that tells the RMI system on the client where to find the class file for that object. Then, in the process of deserializing an object, if RMI can't find the class locally, it uses that URL to do an HTTP Get to retrieve the class file. So you'd need a simple Web server to serve up class files, and you'd also need to change some security parameters on the client. There are a few other tricky issues with dynamic class downloading, but that's the overview.

Complete client code

```
import java.rmi.*;

public class MyRemoteClient {
   public static void main (String[] args) {
        new MyRemoteClient().go();
   }

  public void go() {

     try {
        MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/Remote Hello");

        String s = service.sayHello();

        System.out.println(s);
      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

The Naming class (for doing the remiregistry lookup) is in the java.rmi package

It comes out of the registry as type Object, so don't forget the cast

you need the IP address or hostname

and the name used to bind/rebind the service

It looks just like a regular old method call! (Except it must acknowledge the RemoteException)

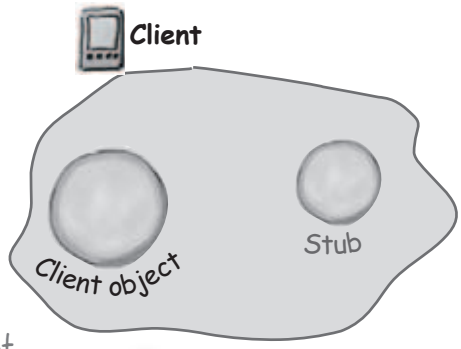
Be sure each machine has the class files it needs.

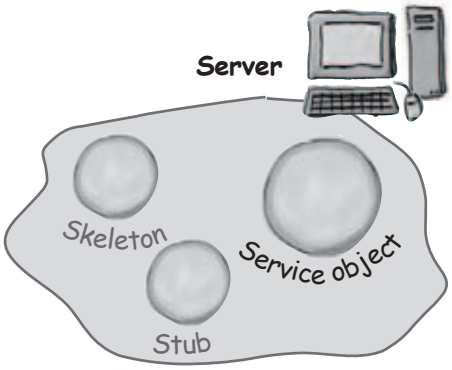
The top three things programmers do wrong with RMI are:

1) Forget to start rmiregistry before starting remote service (when you register the service using Naming.rebind(), the rmiregistry must be running!)

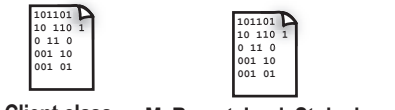
2) Forget to make arguments and return types serializable (you won't know until runtime; this is not something the compiler will detect.)

3) Forget to give the stub class to the client.

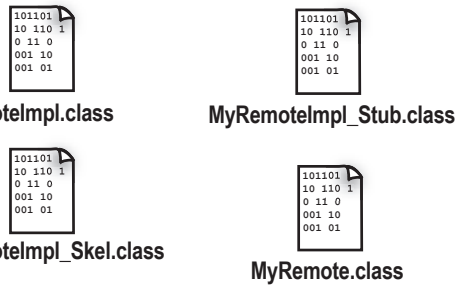




Don't forget, the client uses the interface to call methods on the stub. The client JVM needs the stub class, but the client never refers to the stub class in code. The client always uses the remote interface, as though the remote interface WERE the actual remote object.



The remote interface.



Server needs both the Stub and Skeleton classes, as well as the service and the remote interface. It needs the stub class because remember, the stub is substituted for the real service, when the real service is bound to the RMI registry.

Sharpen your pencil

What's First?

Look at the sequence of events below, and place them in the order in which they occur in a Java RMI application.

1.
2.
3.
4.
5.
6.
7.

The stub sends the method call to the server

The client gets the stub from the RMI registry

The client invokes a method on the stub

The client does a lookup on the RMI Registry

The remote service is registered with the RMI registry

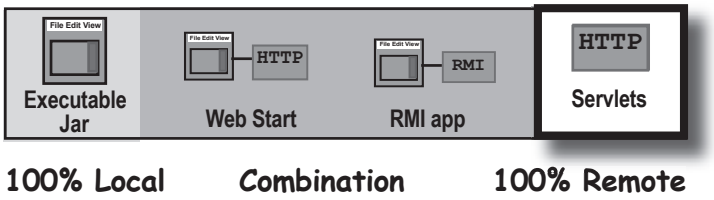
The RMI registry is started

The remote service (remote implementation) is instantiated

BULLET POINTS

- An object on one heap cannot get a normal Java reference to an object on a different heap (which means running on a different JVM)
- Java Remote Method Invocation (RMI) makes it *seem* like you're calling a method on a remote object (i.e. an object in a different JVM), but you aren't.
- When a client calls a method on a remote object, the client is really calling a method on a *proxy* of the remote object. The proxy is called a 'stub'.
- A stub is a client helper object that takes care of the low-level networking details (sockets, streams, serialization, etc.) by packaging and sending method calls to the server.
- To build a remote service (in other words, an object that a remote client can ultimately call methods on), you must start with a remote interface.
- A remote interface must extend the java.rmi.Remote interface, and all methods must declare RemoteException.
- Your remote service implements your remote interface.
- Your remote service should extend UnicastRemoteObject. (Technically there are other ways to create a remote object, but extending UnicastRemoteObject is the simplest).
- Your remote service class must have a constructor, and the constructor must declare a RemoteException (because the superclass constructor declares one).
- Your remote service must be instantiated, and the object registered with the RMI registry.
- To register a remote service, use the static Naming.rebind("Service Name", serviceInstance);
- The RMI registry must be running on the same machine as the remote service, before you try to register a remote object with the RMI registry.
- The client looks up your remote service using the static Naming.lookup("rmi://MyHostName/ServiceName");
- Almost everything related to RMI can throw a RemoteException (checked by the compiler). This includes registering or looking up a service in the reigstry, and *all* remote method calls from the client to the stub.

Yeah, but who really *uses* RMI?



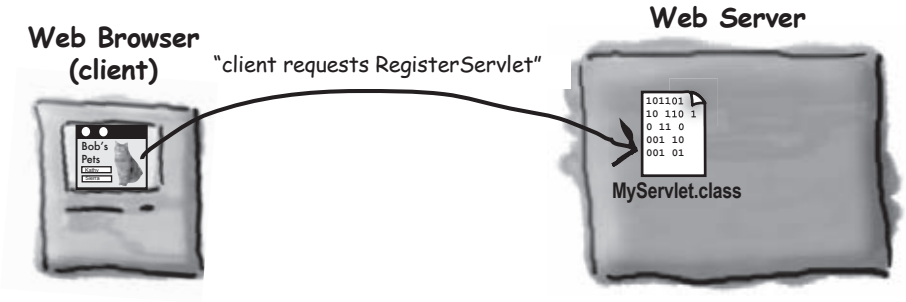
What about Servlets?

Servlets are Java programs that run on (and with) an HTTP web server. When a client uses a web browser to interact with a web page, a request is sent back to the web server. If the request needs the help of a Java servlet, the web server runs (or calls, if the servlet is already running) the servlet code. Servlet code is simply code that runs on the server, to do work as a result of whatever the client requests (for example, save information to a text file or database on the server). If you're familiar with CGI scripts written in Perl, you know exactly what we're talking about. Web developers use CGI scripts or servlets to do everything from sending user-submitted info to a database, to running a web-site's discussion board.

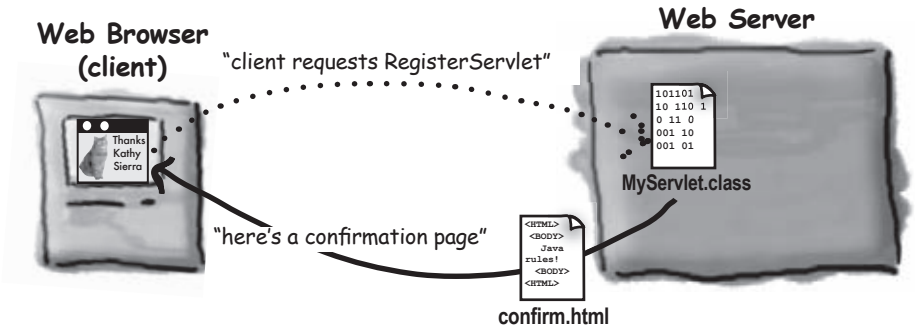
And even servlets can use RMI!

By far, the most common use of J2EE technology is to mix servlets and EJBs together, where servlets are the client of the EJB. And in that case, *the servlet is using RMI to talk to the EJBs.* (Although the way you use RMI with EJB is a *little* different from the process we just looked at.)

1. Client fills out a registration form and clicks 'submit'. The HTTP server (i.e. web server) gets the request, sees that it's for a servlet, and sends the request to the servlet.



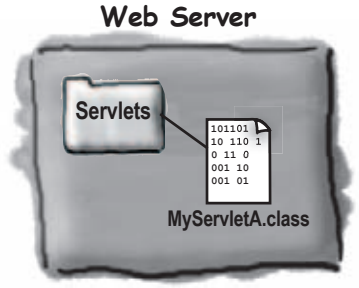
2. Servlet (Java code) runs, adds data to the database, composes a web page (with custom info) and sends it back to the client where it displays in the browser.



Step for making and running a servlet

1 Find out where your servlets need to be placed.

For these examples, we'll assume that you already have a web server up and running, and that it's already configured to support servlets. The most important thing is to find out exactly where your servlet class files have to be placed in order for your server to 'see' them. If you have a web site hosted by an ISP, the hosting service can tell you where to put your servlets, just as they'll tell you where to place your CGI scripts.



2 Get the servlets.jar and add it to your classpath

Servlets aren't part of the standard Java libraries; you need the servlets classes packaged into the servlets.jar file. You can download the servlets classes from java.sun.com, or you can get them from your Java-enabled web server (like Apache Tomcat, at the apache.org site). Without these classes, you won't be able to compile your servlets.

3 Write a servlet class by extending HttpServlet

A servlet is just a Java class that extends HttpServlet (from the javax.servlet.http package). There are other types of servlets you can make, but most of the time we care only about HttpServlet.


```
public class MyServletA extends HttpServlet { ... }
```

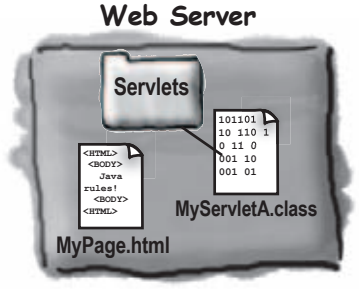
4 Write an HTML page that invokes your servlet

When the user clicks a link that references your servlet, the web server will find the servlet and invoke the appropriate method depending on the HTTP command (GET, POST, etc.)


```
<a href="servlets/MyServletA">This is the most amazing servlet.</a>
```

5 Make your servlet and HTML page available to your server

This is completely dependent on your web server (and more specifically, on which *version* of Java Servlets that you're using). Your ISP may simply tell you to drop it into a "Servlets" directory on your web site. But if you're using, say, the latest version of Tomcat, you'll have a lot more work to do to get the servlet (and web page) into the right location. (We just happen to have a book on this too .)



A very simple Servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServletA extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
                                        throws  ServletException, IOException  {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        String message = "If you're reading this, it worked!";

        out.println("<HTML><BODY>");
        out.println("<H1>" + message + "</H1>");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

Besides io, we need to import two of the servlet packages. Remember, these two packages are NOT part of the Java standard libraries -- you have to download them separately

Most 'normal' servlets will extend HttpServlet, then override one or more methods.

Override the doGet for simple HTTP GET messages.

The web server calls this method, handing you the client's request (you can get data out of it) and a 'response' object that you'll use to send back a response (a page).

This tells the server (and browser) what kind of 'thing' is coming back from the server as a reuslt of this servlet running.

The response object gives us an output stream to 'write' information back out to the server.

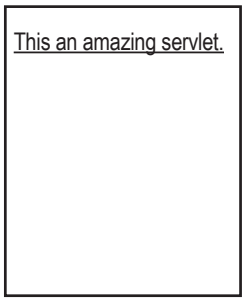
What we 'write' is an HTML page! The page gets delivered through the server back to the browser, just like any other HTML page, even though this is a page that never existed until now. In other words, there's no .html file somewhere with this stuff in it.

HTML page with a link to this servlet

```
<HTML>
  <BODY>
     <a href="servlets/MyServletA">This is an amazing servlet.</a>
  </BODY>
</HTML>
```

What the web page looks like:

click the link to trigger the servlet →



BULLET POINTS

- Servlets are Java classes that run entirely on (and/or within) an HTTP (web) server.
- Servlets are useful for running code on the server as a result of client interaction with a web page. For example, if a client submits information in a web page form, the servlet can process the information, add it to a database, and send back a customized, confirmation response page.
- To compile a servlet, you need the servlet packages which are in the servlets.jar file. The servlet classes are not part of the Java standard libraries, so you need to download the servlets.jar from java.sun.com or get them from a servlet-capable web server. (Note: the Servlet library is included with the Java 2 Enterprise Edition (J2EE))
- To run a servlet, you must have a web server capable of running servlets, such as the Tomcat server from apache.org.
- Your servlet must be placed in a location that's specific to your particular web server, so you'll need to find that out before you try to run your servlets. If you have a web site hosted by an ISP that supports servlets, the ISP will tell you which directory to place your servlets in.
- A typical servlet extends HttpServlet and overrides one or more servlet methods, such as doGet() or doPost().
- The web server starts the servlet and calls the appropriate method (doGet(), etc.) based on the client's request.
- The servlet can send back a response by getting a PrintWriter output stream from the response parameter of the doGet() method.
- The servlet 'writes' out an HTML page, complete with tags).

there are no Dumb Questions

Q: What's a JSP, and how does it relate to servlets?

A: JSP stands for Java Server Pages. In the end, the web server turns a JSP into a servlet, but the difference between a servlet and a JSP is what YOU (the developer) actually create. With a servlet, you write a Java *class* that contains *HTML* in the output statements (if you're sending back an HTML page to the client). But with a JSP, it's the opposite—you write an *HTML* page that contains *Java* code!

This gives you the ability to have dynamic web pages where you write the page as a normal HTML page, except you embed Java code (and other tags that "trigger" Java code at runtime) that gets processed at runtime. In other words, part of the page is customized at runtime when the Java code runs.

The main benefit of JSP over regular servlets is that it's just a lot easier to write the HTML part of a servlet as a JSP page than to write HTML in the torturous print out statements in the servlet's response. Imagine a reasonably complex HTML page, and now imagine formatting it within println statements. Yikes!

But for many applications, it isn't necessary to use JSPs because the servlet doesn't need to send a dynamic response, or the HTML is simple enough not to be such a big pain. And, there are still many web servers out there that support servlets but do not support JSPs, so you're stuck.

Another benefit of JSPs is that you can separate the work by having the Java developers write the servlets and the web page developers write the JSPs. That's the promised benefit, anyway. In reality, there's still a Java learning curve (and a tag learning curve) for anyone writing a JSP, so to think that an HTML web page designer can bang out JSPs is not realistic. Well, not without tools. But that's the good news—authoring tools are starting to appear, that help web page designers create JSPs without writing the code from scratch.

Q: Is this all you're gonna say about servlets? After such a *huge* thing on RMI?

A: Yes. RMI is part of the Java language, and all the classes for RMI are in the standard libraries. Servlets and JSPs are *not* part of the Java language; they're considered *standard extensions.* You can run RMI on any modern JVM, but Servlets and JSPs require a properly configured web server with a servlet "container". This is our way of saying, "it's beyond the scope of this book." But you can read much more in the lovely *Head First Servlets & JSP.*

Just for fun, let's make the Phrase-O-Matic work as a servlet

Now that we told you that we won't say any more about servlets, we can't resist servletizing (yes, we *can* verbify it) the Phrase-O-Matic from chapter 1. A servlet is still just Java. And Java code can call Java code from other classes. So a servlet is free to call a method on the Phrase-O-Matic. All you have to do is drop the Phrase-O-Matic class into the same directory as your servlet, and you're in business. (The Phrase-O-Matic code is on the next page).


```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class KathyServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response)
                                          throws ServletException, IOException  {

        String title = "PhraseOMatic has generated the following phrase.";

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println("PhraseOmatic");
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>" + PhraseOMatic.makePhrase());
        out.println("<P><a href=\"KathyServlet\">make another phrase</a></p>");
        out.println("</BODY></HTML>");

        out.close();
    }
}
```

See? Your servlet can call methods on another class. In this case, we're calling the static makePhrase() method of the PhraseOMatic class (on the next page)

Phrase-O-Matic code, servlet-friendly

This is a slightly different version from the code in chapter one. In the original, we ran the entire thing in a main() method, and we had to rerun the program each time to generate a new phrase at the command-line. In this version, the code simply returns a String (with the phrase) when you invoke the static makePhrase() method. That way, you can call the method from any other code and get back a String with the randomly-composed phrase.

Please note that these long String[] array assignments are a victim of word-processing here—don't type in the hyphens! Just keep on typing and let your code editor do the wrapping. And whatever you do, don't hit the return key in the middle of a String (i.e. something between double quotes).

```
public class PhraseOMatic {
   public static String makePhrase() {

     // make three sets of words to choose from
     String[] wordListOne = {"24/7","multi-Tier","30,000 foot","B-to-B","win-win","front-
end", "web-based","pervasive", "smart", "six-sigma","critical-path", "dynamic"};

     String[] wordListTwo = {"empowered", "sticky", "valued-added", "oriented", "centric",
"distributed", "clustered", "branded","outside-the-box", "positioned", "networked", "fo-
cused", "leveraged", "aligned", "targeted", "shared", "cooperative", "accelerated"};

     String[] wordListThree = {"process", "tipping point", "solution", "architecture",
"core competency", "strategy", "mindshare", "portal", "space", "vision", "paradigm", "mis-
sion"};

    // find out how many words are in each list
    int oneLength = wordListOne.length;
    int twoLength = wordListTwo.length;
    int threeLength = wordListThree.length;

    // generate three random numbers, to pull random words from each list
    int rand1 = (int) (Math.random() * oneLength);
    int rand2 = (int) (Math.random() * twoLength);
    int rand3 = (int) (Math.random() * threeLength);

    // now build a phrase
    String phrase = wordListOne[rand1] + " " + wordListTwo[rand2] + " " +
wordListThree[rand3];

    // now return it
    return ("What we need is a " + phrase);
  }
}
```

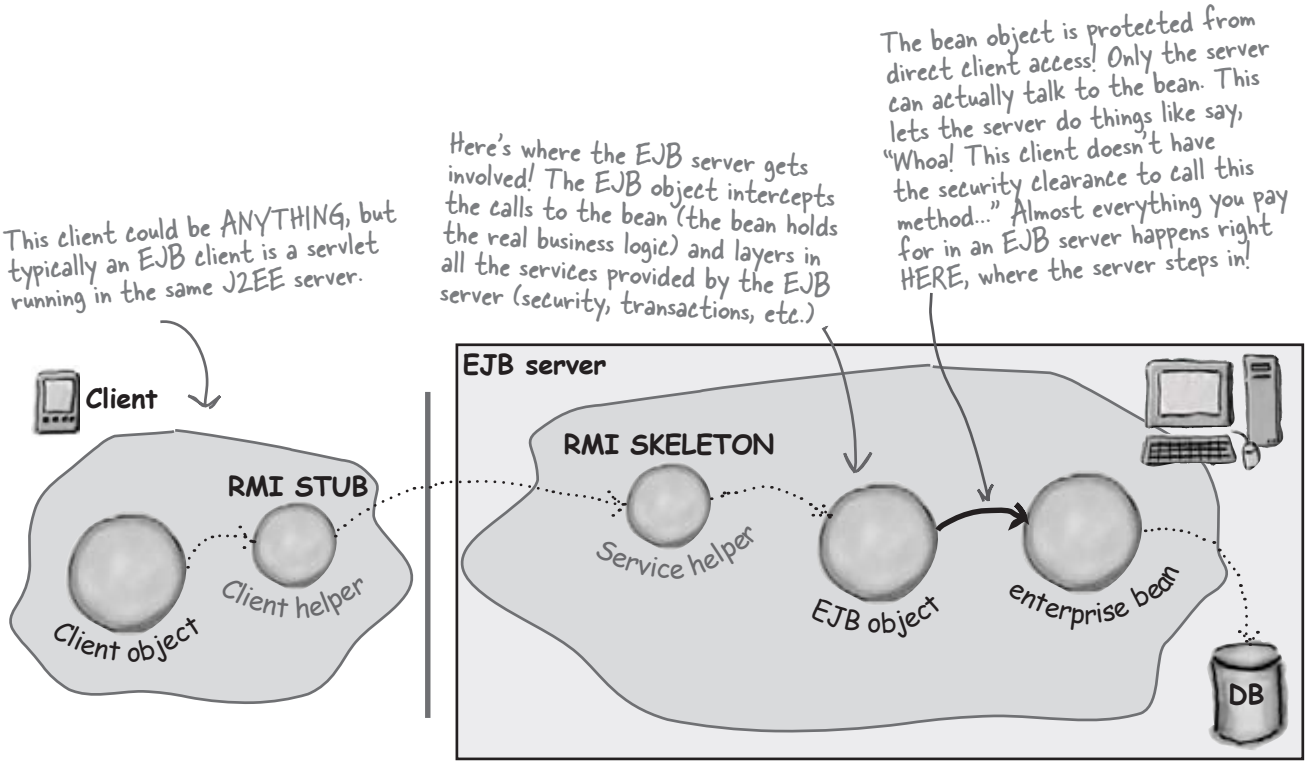
Enterprise JavaBeans: RMI on steroids

RMI is great for writing and running remote services. But you wouldn't run something like an Amazon or eBay on RMI alone. For a large, deadly serious, enterprise application, you need something more. You need something that can handle transactions, heavy concurrency issues (like a gazillion people are hitting your server at once to buy those organic dog kibbles), security (not just anyone should hit your payroll database), and data management. For that, you need an *enterprise application server*.

In Java, that means a Java 2 Enterprise Edition (J2EE) server. A J2EE server includes both a web server and an Enterprise JavaBeans (EJB) server, so that you can deploy an application that includes both servlets and EJBs. Like servlets, EJB is way beyond the scope of this book, and there's no way to show "just a little" EJB example with code, but we *will* take a quick look at how it works. (For a much more detailed treatment of EJB, we can recommend the lively Head First EJB certification study guide.)

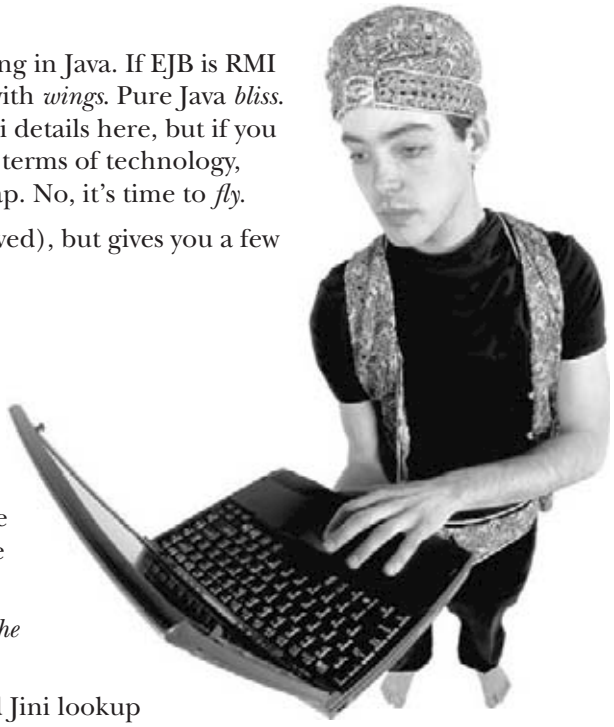
An EJB server adds a bunch of services that you don't get with straight RMI. Things like transactions, security, concurrency, database management, and networking.

An EJB server steps into the middle of an RMI call and layers in all of the services.



This is only a small part of the EJB picture!

For our final trick... a little Jini



We love Jini. We think Jini is pretty much the best thing in Java. If EJB is RMI on steroids (with a bunch of managers), Jini is RMI with *wings*. Pure Java *bliss*. Like the EJB material, we can't get into any of the Jini details here, but if you know RMI, you're three-quarters of the way there. In terms of technology, anyway. In terms of *mindset*, it's time to make a big leap. No, it's time to *fly*.

Jini uses RMI (although other protocols can be involved), but gives you a few key features including:

Adaptive discovery

Self-healing networks

With RMI, remember, the client has to know the name and location of the remote service. The client code for the lookup includes the IP address or hostname of the remote service (because that's where the RMI registry is running) *and* the logical name the service was registered under.

But with Jini, the client has to know only one thing: *the* ***interface*** *implemented by the service!* That's it.

So how do you find things? The trick revolves around Jini lookup services. Jini lookup services are far more powerful and flexible than the RMI registry. For one thing, Jini lookup services announce themselves to the network, *automatically*. When a lookup service comes online, it sends a message (using IP multicast) out to the network saying, "I'm here, if anyone's interested."

But that's not all. Let's say you (a client) come online *after* the lookup service has already announced itself, *you* can send a message to the entire network saying, "Are there any lookup services out there?"

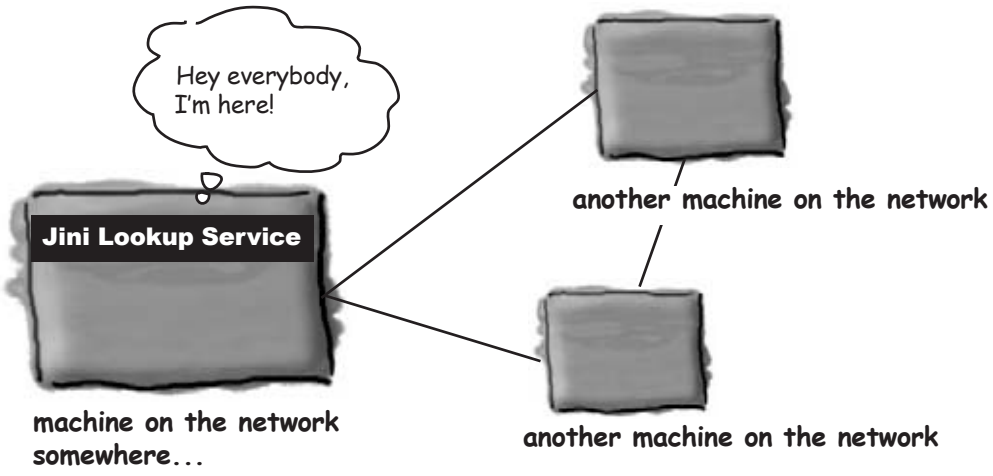
Except that you're not really interested in the lookup service *itself*—you're interested in the services that are *registered* with the lookup service. Things like RMI remote services, other serializable Java objects, and even devices such as printers, cameras, and coffee-makers.

And here's where it gets even more fun: when a service comes online, it will dynamically discover (and *register* itself with) any Jini lookup services on the network. When the service registers with the lookup service, the service sends a serialized object to be placed in the lookup service. That serialized object can be a stub to an RMI remote service, a driver for a networked device, or even the whole service itself that (once you get it from the lookup service) runs locally on your machine. And instead of registering by *name*, the service registers by the *interface* it implements.

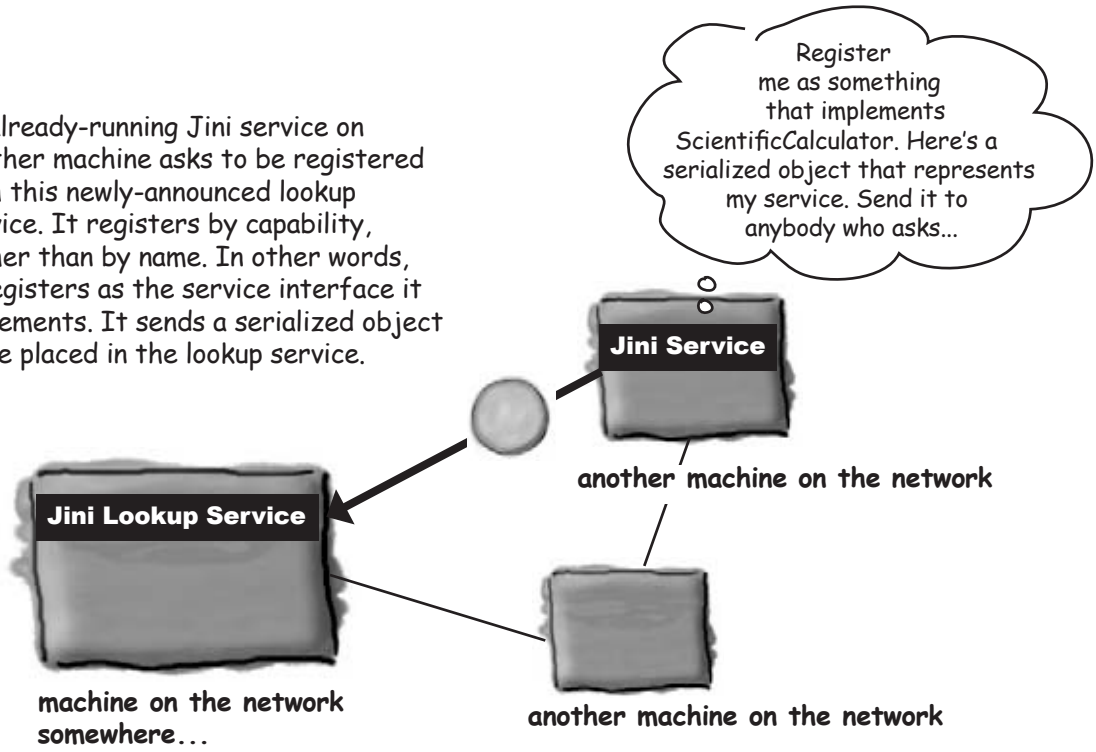
Once you (the client) have a reference to a lookup service, you can say to that lookup service, "Hey, do you have anything that implements ScientificCalculator?" At that point, the lookup service will check its list of registered interfaces, and assuming it finds a match, says back to you, "Yes I *do* have something that implements that interface. Here's the serialized object the ScientificCalculator service registered with me."

Adaptive discovery in action

1. Jini lookup service is launched somewhere on the network, and announces itself using IP multicast.

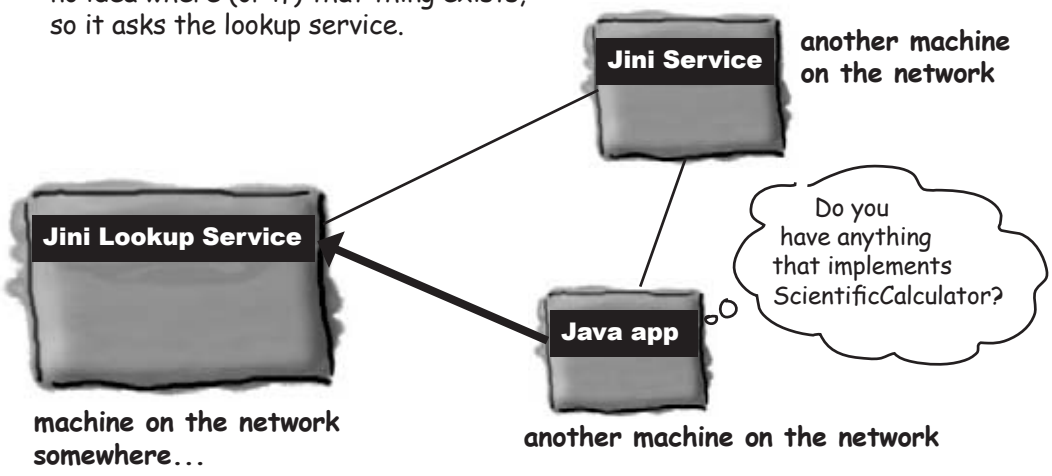


2. An already-running Jini service on another machine asks to be registered with this newly-announced lookup service. It registers by capability, rather than by name. In other words, it registers as the service interface it implements. It sends a serialized object to be placed in the lookup service.

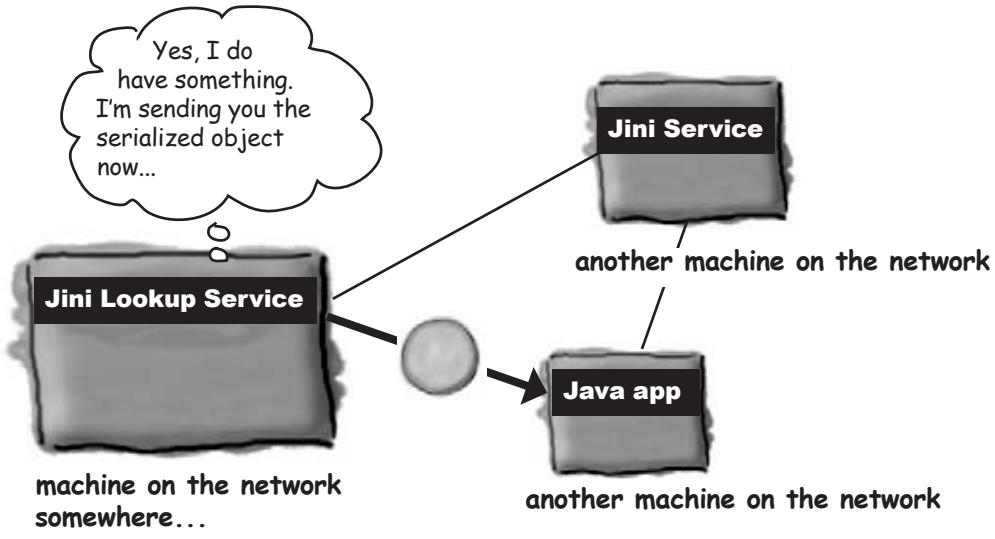


Adaptive discovery in action, continued...

3 A client on the network wants something that implements the ScientificCalculator interface. It has no idea where (or if) that thing exists, so it asks the lookup service.

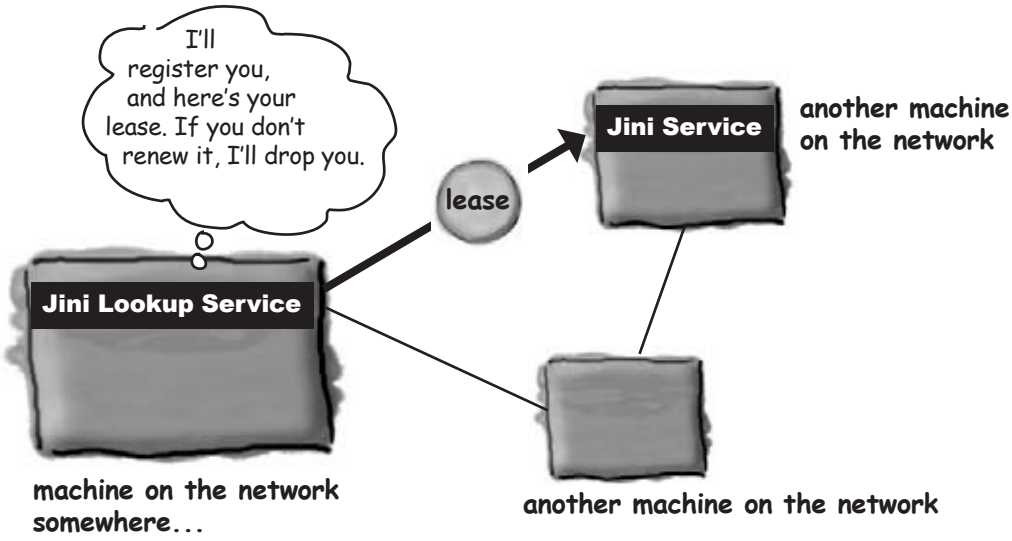


4 The lookup service responds, since it does have something registered as a ScientificCalculator interface.

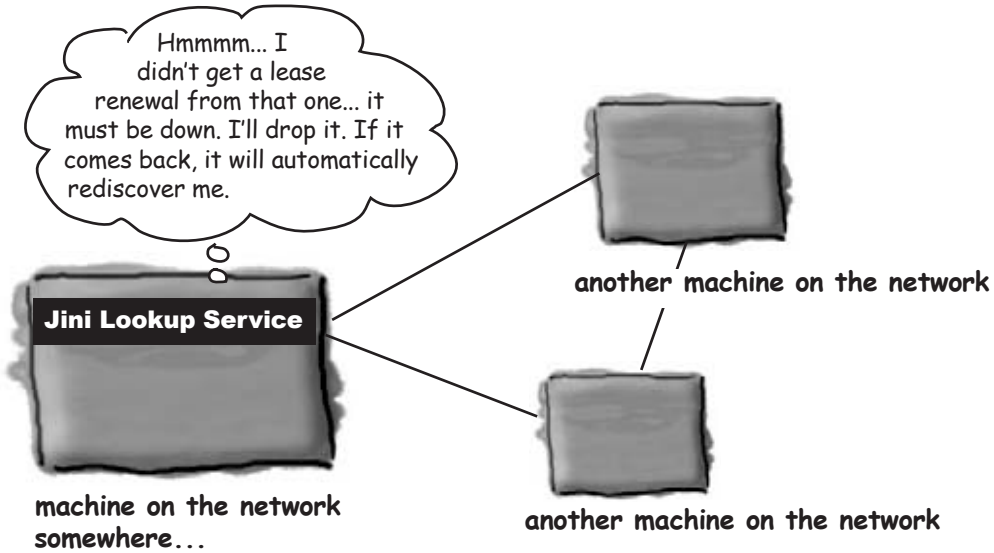


Self-healing network in action

1. A Jini Service has asked to register with the lookup service. The lookup service responds with a "lease". The newly-registered service must keep renewing the lease, or the lookup service assumes the service has gone offline. The lookup service wants always to present an accurate picture to the rest of the network about which services are available.



2. The service goes offline (somebody shuts it down), so it fails to renew its lease with the lookup service. The lookup service drops it.



Final Project: the Universal Service browser

We're going to make something that isn't Jini-enabled, but quite easily could be. It will give you the flavor and feeling of Jini, but using straight RMI. In fact the main difference between our application and a Jini application is how the service is discovered. Instead of the Jini lookup service, which automatically announces itself and lives anywhere on the network, we're using the RMI registry which must be on the same machine as the remote service, and which does not announce itself automatically.

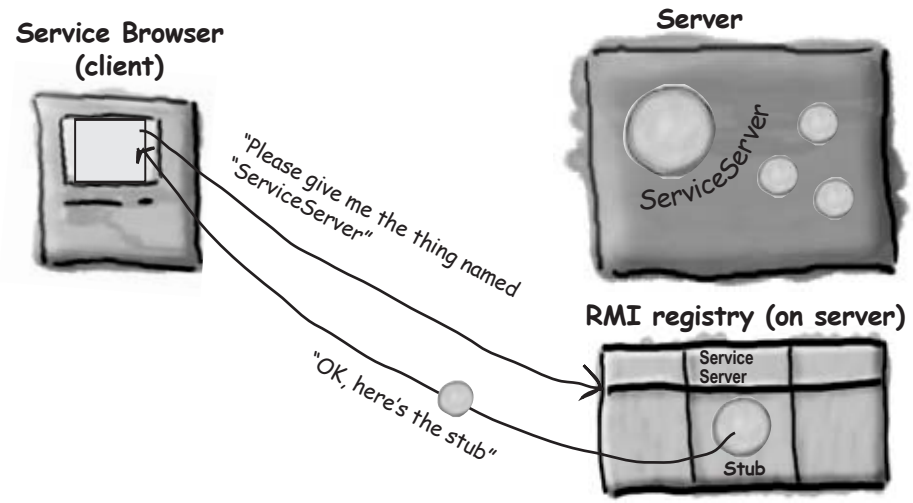
And instead of our service registering itself automatically with the lookup service, *we* have to register it in the RMI registry (using Naming.rebind()).

But once the client has found the service in the RMI registry, the rest of the application is almost identical to the way we'd do it in Jini. (The main thing missing is the *lease* that would let us have a self-healing network if any of the services go down.)

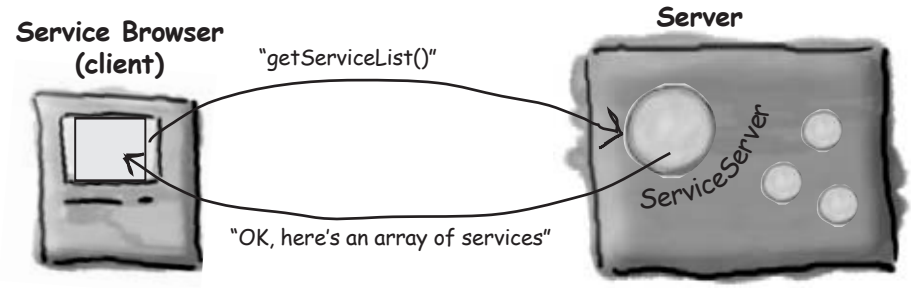
The universal service browser is like a specialized web browser, except instead of HTML pages, the service browser downloads and displays interactive Java GUIs that we're calling *universal services*.

How it works:

1. Client starts up and does a lookup on the RMI registry for the service called "ServiceServer", and gets back the stub.



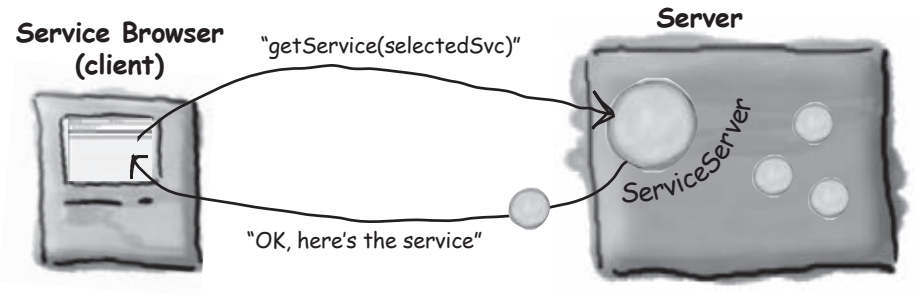
2. Client calls getServiceList() on the stub. The ServiceServer returns an array of services



3. Client displays the list of services in a GUI

How it works, continued...

4. User selects from the list, so client calls the getService() method on the remote service. The remote service returns a serialized object that is an actual service that will run inside the client browser.



5. Client calls the getGuiPanel() on the serialized service object it just got from the remote service. The GUI for that service is displayed inside the browser, and the user can interact with it locally. At this point, we don't need the remote service unless/until the user decides to select another service.

The classes and interfaces:

1. **interface ServiceServer implements Remote**
A regular old RMI remote interface for the remote service (the remote service has the method for getting the service list and returning a selected service).

2. **class ServiceServerImpl implements ServiceServer**
The actual RMI remote service (extends UnicastRemoteObject). Its job is to instantiate and store all the services (the things that will be shipped to the client), and register the server itself (ServiceServerImpl) with the RMI registry.

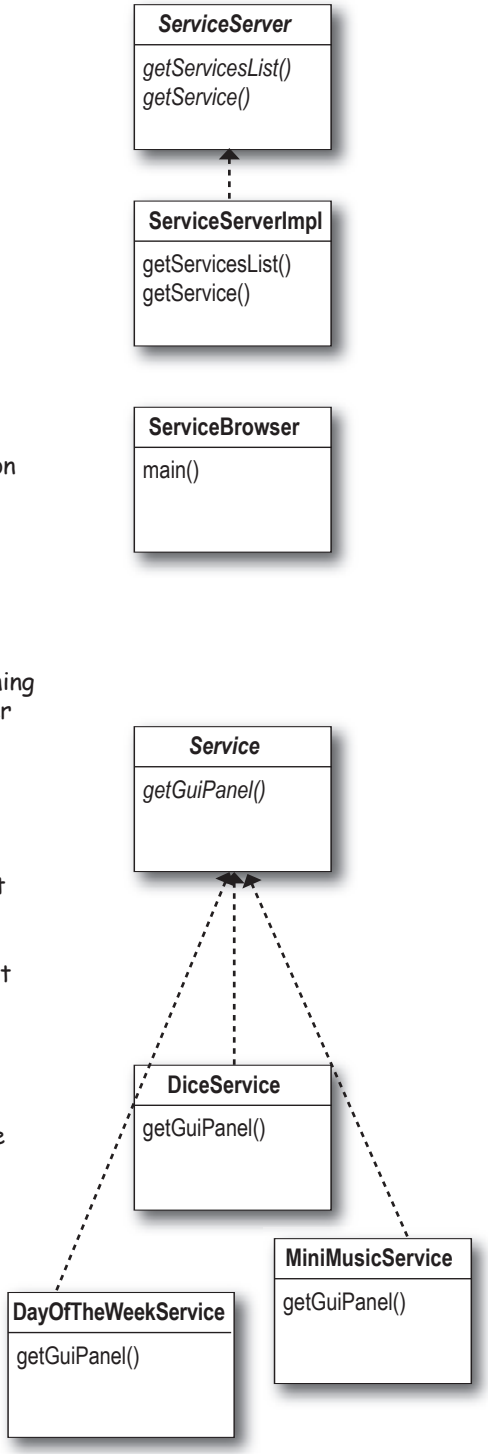
3. **class ServiceBrowser**
The client. It builds a very simple GUI, does a lookup in the RMI registry to get the ServiceServer stub, then calls a remote method on it to get the list of services to display in the GUI list.

4. **interface Service**
This is the key to everything. This very simple interface has just one method, getGuiPanel(). Every service that gets shipped over to the client must implement this interface. This is what makes the whole thing UNIVERSAL! By implementing this interface, a service can come over even though the client has no idea what the actual class (or classes) are that make up that service. All the client knows is that whatever comes over, it implements the Service interface, so it MUST have a getGuiPanel() method.
The client gets a serialized object as a result of calling getService(selectedSvc) on the ServiceServer stub, and all the client says to that object is, "I don't know who or what you are, but I DO know that you implement the Service interface, so I know I can call getGuiPanel() on you. And since getGuiPanel() returns a JPanel, I'll just slap it into the browser GUI and start interacting with it!

5. **class DiceService implements Service**
Got dice? If not, but you need some, use this service to roll anywhere from 1 to 6 virtual dice for you.

6. **class MiniMusicService implements Service**
Remember that fabulous little 'music video' program from the first GUI Code Kitchen? We've turned it into a service, and you can play it over and over and over until your roommates finally leave.

7. **class DayOfTheWeekService implements Service**
Were you born on a Friday? Type in your birthday and find out.



interface ServiceServer (the remote interface)

```
import java.rmi.*;

public interface ServiceServer extends Remote {

    Object[] getServiceList() throws RemoteException;

    Service getService(Object serviceKey) throws RemoteException;
}
```

A normal RMI remote interface, defines the two methods the remote service will have.

interface Service (what the GUI services implement)

```
import javax.swing.*;
import java.io.*;

public interface Service extends Serializable {
    public JPanel getGuiPanel();
}
```

A plain old (i.e. non-remote) interface, that defines the one method that any universal service must have—getGuiPanel(). The interface extends Serializable, so that any class implementing the Service interface will automatically be Serializable.

That's a must, because the services get shipped over the wire from the server, as a result of the client calling getService() on the remote ServiceServer.

class ServiceServerImpl (the remote implementation)

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

public class ServiceServerImpl extends UnicastRemoteObject implements ServiceServer  {

    HashMap serviceList;

    public ServiceServerImpl() throws RemoteException {
       setUpServices();
    }

    private void setUpServices() {
       serviceList = new HashMap();
       serviceList.put("Dice Rolling Service", new DiceService());
       serviceList.put("Day of the Week Service", new DayOfTheWeekService());
       serviceList.put("Visual Music Service", new MiniMusicService());
    }

    public Object[] getServiceList() {
       System.out.println("in remote");
       return serviceList.keySet().toArray();

    }

    public Service getService(Object serviceKey) throws RemoteException {
       Service theService = (Service) serviceList.get(serviceKey);
       return theService;
    }

    public static void main (String[] args) {
       try {
         Naming.rebind("ServiceServer", new ServiceServerImpl());
       } catch(Exception ex) {
           ex.printStackTrace();
      }
       System.out.println("Remote service is running");
    }
}
```

A normal RMI implementation

The services will be stored in a HashMap collection. Instead of putting ONE object in the collection, you put TWO -- a key object (like a String) and a value object (whatever you want). (see appendix B for more on HashMap)

When the constructor is called, initialize the actual universal services (DiceService, MiniMusicService, etc.)

Make the services (the actual service objects) and put them into the HashMap, with a String name (for the 'key').

Client calls this in order to get a list of services to display in the browser (so the user can select one). We send an array of type Object (even though it has Strings inside) by making an array of just the KEYS that are in the HashMap. We won't send an actual Service object unless the client asks for it by calling getService().

Client calls this method after the user selects a service from the displayed list of services (that it got from the method above). This code uses the key (the same key originally sent to the client) to get the corresponding service out of the HashMap.

class ServiceBrowser (the client)

```
import java.awt.*;
import javax.swing.*;
import java.rmi.*;
import java.awt.event.*;


public class ServiceBrowser {

   JPanel mainPanel;
   JComboBox serviceList;
   ServiceServer server;


   public void buildGUI() {
      JFrame frame = new JFrame("RMI Browser");
      mainPanel = new JPanel();
      frame.getContentPane().add(BorderLayout.CENTER, mainPanel);

      Object[] services = getServicesList();

      serviceList = new JComboBox(services);

      frame.getContentPane().add(BorderLayout.NORTH, serviceList);

      serviceList.addActionListener(new MyListListener());

      frame.setSize(500,500);
      frame.setVisible(true);

  }

   void loadService(Object serviceSelection) {
       try {
          Service svc = server.getService(serviceSelection);

          mainPanel.removeAll();
          mainPanel.add(svc.getGuiPanel());
          mainPanel.validate();
          mainPanel.repaint();
        } catch(Exception ex) {
           ex.printStackTrace();
        }
   }
```

this method does the RMI registry lookup, gets the stub, and calls getServiceList(). (The actual method is on the next page).

Add the services (an array of Objects) to the JComboBox (the list). The JComboBox knows how to make displayable Strings out of each thing in the array.

Here's where we add the actual service to the GUI, after the user has selected one. (This method is called by the event listener on the JComboBox). We call getService() on the remote server (the stub for ServiceServer) and pass it the String that was displayed in the list (which is the SAME String we originally got from the server when we called getServiceList()). The server returns the actual service (serialized), which is automatically deserialized (thanks to RMI) and we simply call the getGuiPanel() on the service and add the result (a JPanel) to the browser's mainPanel.

```
    Object[] getServicesList() {
       Object obj = null;
       Object[] services = null;

       try {

          obj = Naming.lookup("rmi://127.0.0.1/ServiceServer");

       }
       catch(Exception ex) {
        ex.printStackTrace();
       }
       server = (ServiceServer) obj;


       try {

         services = server.getServiceList();

       } catch(Exception ex) {
          ex.printStackTrace();
       }
       return services;

    }

    class MyListListener implements ActionListener {
       public void actionPerformed(ActionEvent ev) {

           Object selection =  serviceList.getSelectedItem();
           loadService(selection);
       }
     }

    public static void main(String[] args) {
       new ServiceBrowser().buildGUI();
    }
}
```

Do the RMI lookup, and get the stub

Cast the stub to the remote interface type, so that we can call getServiceList() on it

getServiceList() gives us the array of Objects, that we display in the JComboBox for the user to select from.

If we're here, it means the user made a selection from the JComboBox list. So, take the selection they made and load the appropriate service. (see the loadService method on the previous page, that asks the server for the service that corresponds with this selection)

DiceService code

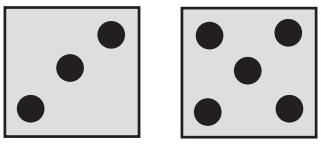
class DiceService (a universal service, implements Service)


```
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

public class DiceService implements Service {

    JLabel label;
    JComboBox numOfDice;



    public JPanel getGuiPanel() {
       JPanel panel = new JPanel();
       JButton button = new JButton("Roll 'em!");
       String[] choices = {"1", "2", "3", "4", "5"};
       numOfDice = new JComboBox(choices);
       label = new JLabel("dice values here");
       button.addActionListener(new RollEmListener());
       panel.add(numOfDice);
       panel.add(button);
       panel.add(label);
       return panel;
    }

   public class RollEmListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
         // roll the dice
         String diceOutput = "";
         String selection = (String)  numOfDice.getSelectedItem();
         int numOfDiceToRoll = Integer.parseInt(selection);
         for (int i = 0; i < numOfDiceToRoll; i++) {
            int r = (int) ((Math.random() * 6) + 1);
            diceOutput += (" " + r);
         }
        label.setText(diceOutput);

      }
    }
```

Here's the one important method! The method of the Service interface-- the one the client's gonna call when this service is selected and loaded. You can do whatever you want in the getGuiPanel() method, as long as you return a JPanel, so it builds the actual dice-rolling GUI.

Sharpen your pencil

Think about ways to improve the DiceService. One suggestion: using what you learned in the GUI chapters, make the dice graphical. Use a rectangle, and draw the appropriate number of circles on each one, corresponding to the roll for that particular die.



class MiniMusicService (a universal service, implements Service)

```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiniMusicService implements Service {

    MyDrawPanel myPanel;

    public JPanel getGuiPanel() {
       JPanel mainPanel = new JPanel();
       myPanel = new MyDrawPanel();
       JButton playItButton = new JButton("Play it");
       playItButton.addActionListener(new PlayItListener());
       mainPanel.add(myPanel);
       mainPanel.add(playItButton);
       return mainPanel;
    }

    public class PlayItListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {

        try {

         Sequencer sequencer = MidiSystem.getSequencer();
         sequencer.open();

         sequencer.addControllerEventListener(myPanel, new int[] {127});
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

         for (int i = 0; i < 100; i+= 4) {

            int  rNum = (int) ((Math.random() * 50) + 1);
            if (rNum < 38) {  // so now only do it if num <38 (75% of the time)
              track.add(makeEvent(144,1,rNum,100,i));
              track.add(makeEvent(176,1,127,0,i));
              track.add(makeEvent(128,1,rNum,100,i + 2));
            }
         } // end loop

         sequencer.setSequence(seq);
         sequencer.start();
         sequencer.setTempoInBPM(220);
      } catch (Exception ex) {ex.printStackTrace();}

     } // close actionperformed
    } // close inner class
```

The service method! All it does is display a button and the drawing service (where the rectangles will eventually be painted).

This is all the music stuff from the Code Kitchen in chapter 12, so we won't annotate it again here.

class MiniMusicService, continued...

```
public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
     MidiEvent event = null;
        try {
           ShortMessage a = new ShortMessage();
           a.setMessage(comd, chan, one, two);
           event = new MidiEvent(a, tick);

         }catch(Exception e) { }
     return event;
     }


class MyDrawPanel extends JPanel implements ControllerEventListener {

     // only if we got an event do we want to paint
     boolean msg = false;

     public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
    }

     public Dimension getPreferredSize() {
       return new Dimension(300,300);
     }

     public void paintComponent(Graphics g) {
        if (msg) {

          Graphics2D g2 = (Graphics2D) g;

          int r = (int) (Math.random() * 250);
          int gr = (int) (Math.random() * 250);
          int b = (int) (Math.random() * 250);

          g.setColor(new Color(r,gr,b));

          int ht = (int) ((Math.random() * 120) + 10);
          int width = (int) ((Math.random() * 120) + 10);

          int x = (int) ((Math.random() * 40) + 10);
          int y =  (int) ((Math.random() * 40) + 10);

          g.fillRect(x,y,ht, width);
          msg = false;

       } // close if
     } // close method
   }  // close inner class
} // close class
```

Nothing new on this entire page. You've seen it all in the graphics CodeKitchen. If you want another exercise, try annotating this code yourself, then compare it with the CodeKitchen in the "A very graphic story" chapter.

class DayOfTheWeekService (a universal service, implements Service)


```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class DayOfTheWeekService implements Service {

    JLabel outputLabel;
    JComboBox month;
    JTextField day;
    JTextField year;

    public JPanel getGuiPanel() {
       JPanel panel = new JPanel();
       JButton button = new JButton("Do it!");
       button.addActionListener(new DoItListener());
       outputLabel = new JLabel("date appears here");
       DateFormatSymbols dateStuff = new DateFormatSymbols();
       month = new JComboBox(dateStuff.getMonths());
       day = new JTextField(8);
       year = new JTextField(8);
       JPanel inputPanel = new JPanel(new GridLayout(3,2));
       inputPanel.add(new JLabel("Month"));
       inputPanel.add(month);
       inputPanel.add(new JLabel("Day"));
       inputPanel.add(day);
       inputPanel.add(new JLabel("Year"));
       inputPanel.add(year);
       panel.add(inputPanel);
       panel.add(button);
       panel.add(outputLabel);
       return panel;
    }

    public class DoItListener implements ActionListener {
       public void actionPerformed(ActionEvent ev) {
            int monthNum = month.getSelectedIndex();
            int dayNum = Integer.parseInt(day.getText());
            int yearNum = Integer.parseInt(year.getText());
            Calendar c = Calendar.getInstance();
            c.set(Calendar.MONTH, monthNum);
            c.set(Calendar.DAY_OF_MONTH, dayNum);
            c.set(Calendar.YEAR, yearNum);
            Date date = c.getTime();
            String dayOfWeek = (new SimpleDateFormat("EEEE")).format(date);
            outputLabel.setText(dayOfWeek);
      }
    }
}
```

The Service interface method that builds the GUI

Refer to chapter 10 if you need a reminder of how number and date formatting works. This code is slightly different, however, because it uses the Calendar class. Also, the SimpleDateFormat lets us specify a pattern for how the date should print out.

Congratulations!
You made it to the end.

Of course, there's still the two appendices.
And the index.
And then there's the web site...
There's no escape, really.