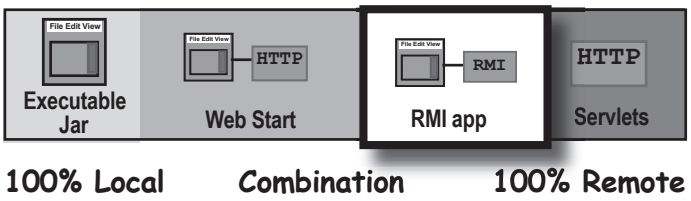
Distributed Computing

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*.

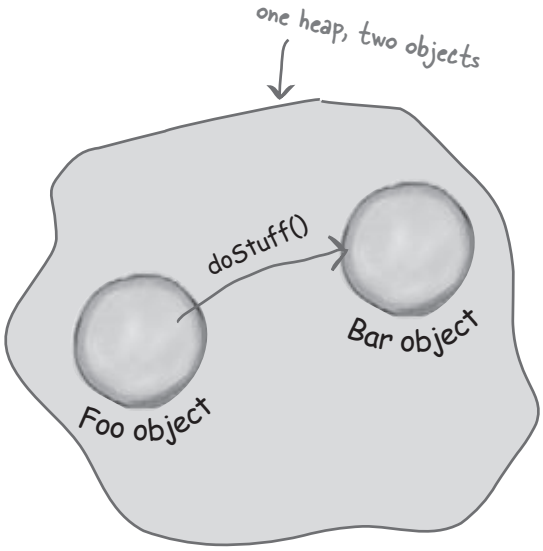


Method calls are always between two objects on the same heap.

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.



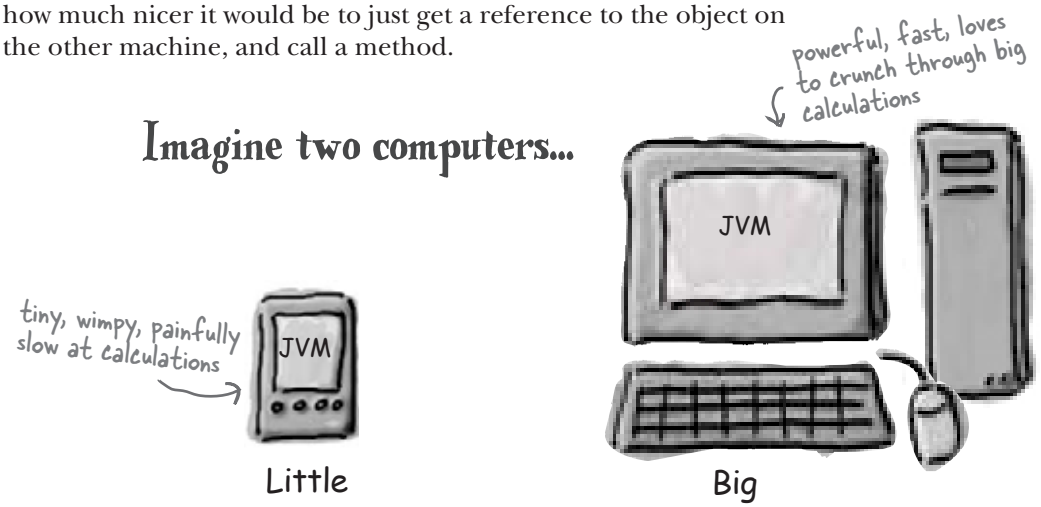
In most applications, when one object calls a method on another, both objects are on the same heap. In other words, both are running within the same JVM.

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...

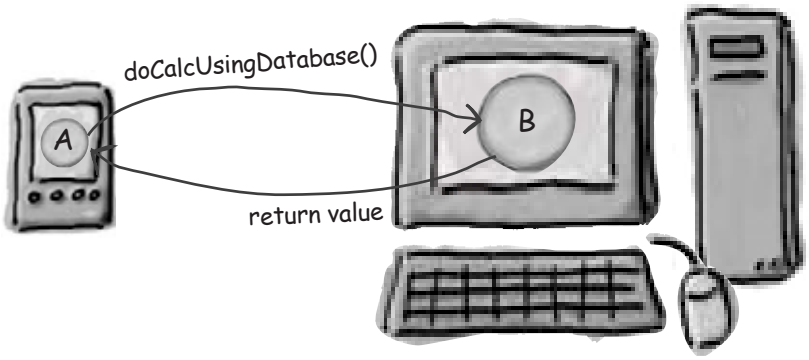
```
double doCalcUsingDatabase(CalcNumbers numbers)
```

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.

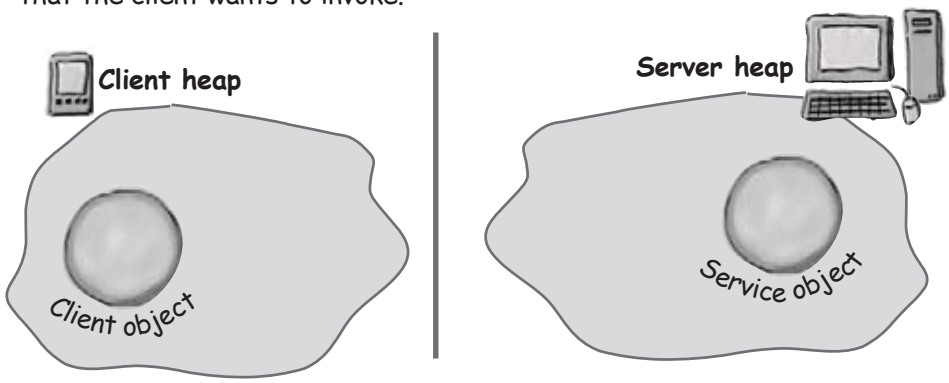
That's what RMI (Remote Method Invocation) gives you!

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.

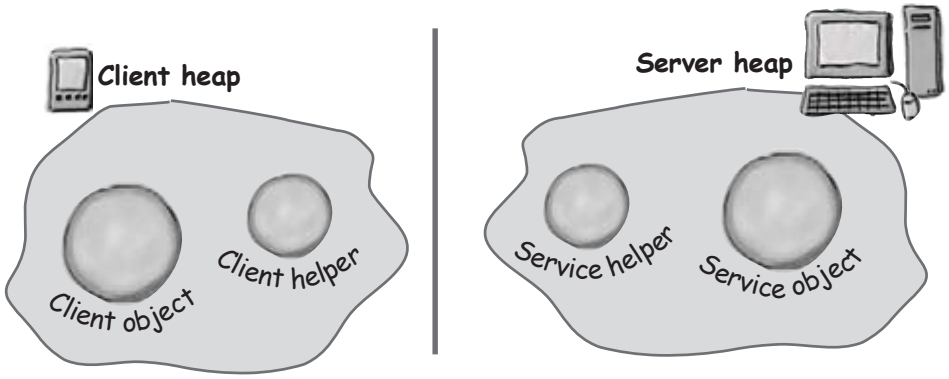
A design for remote method calls

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.*

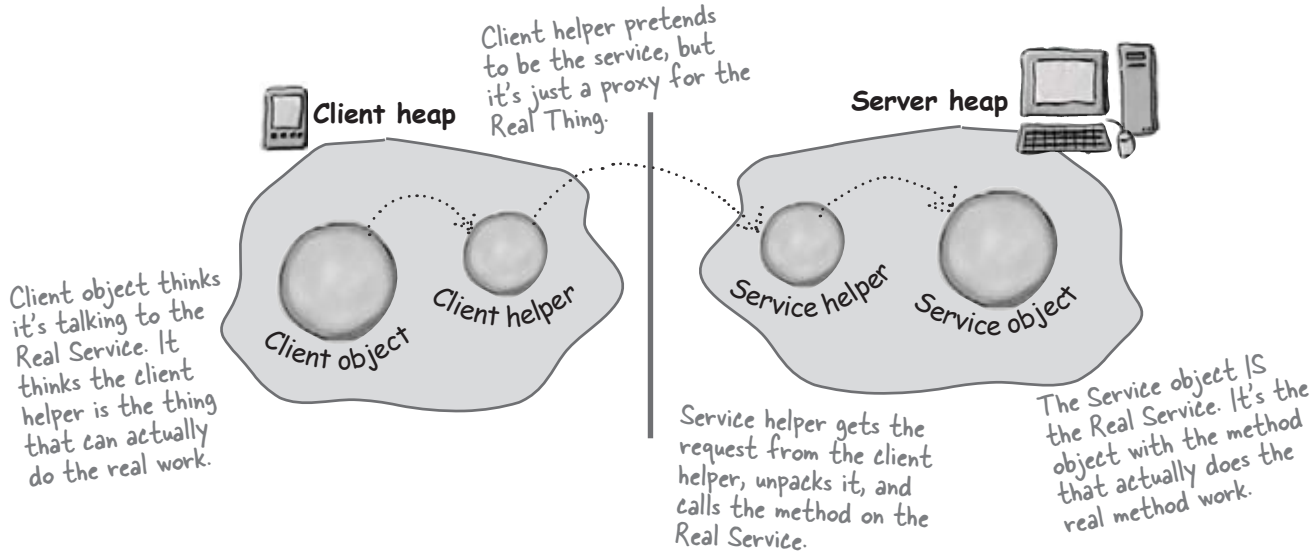
In other words, the client object *thinks* it's calling a method on the remote service, because the client helper is *pretending* to be the service object. ***Pretending to be the thing with the method the client wants to call!***

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.

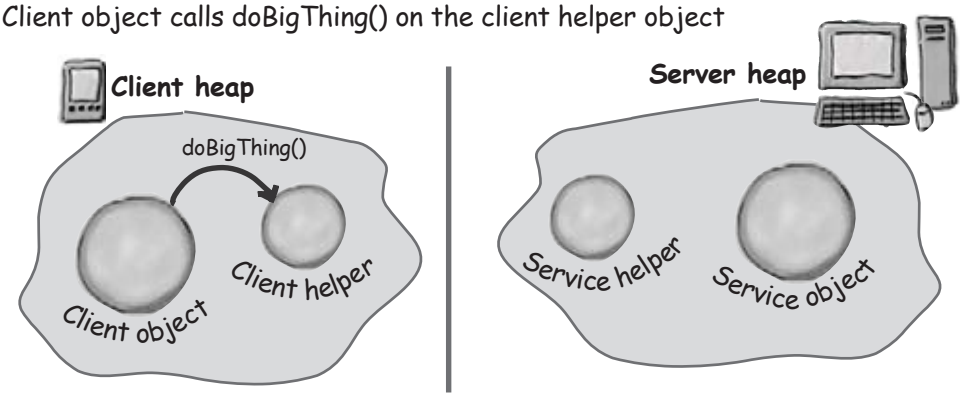
The service helper gets the return value from the service, packs it up, and ships it back (over a Socket's output stream) to the client helper. The client helper unpacks the information and returns the value to the client object.

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.

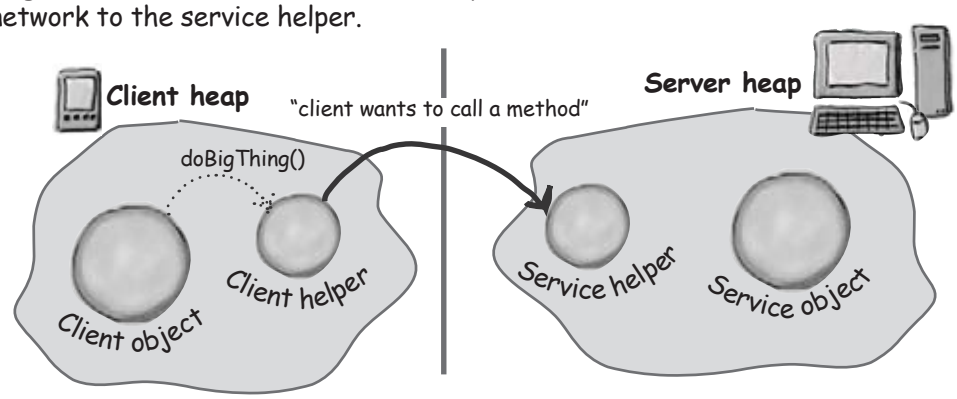


How the method call happens

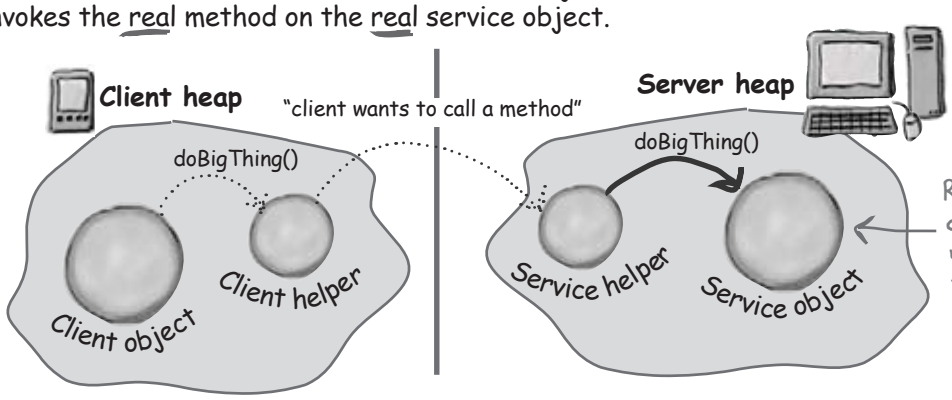
1. Client object calls doBigThing() on the client helper object



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!

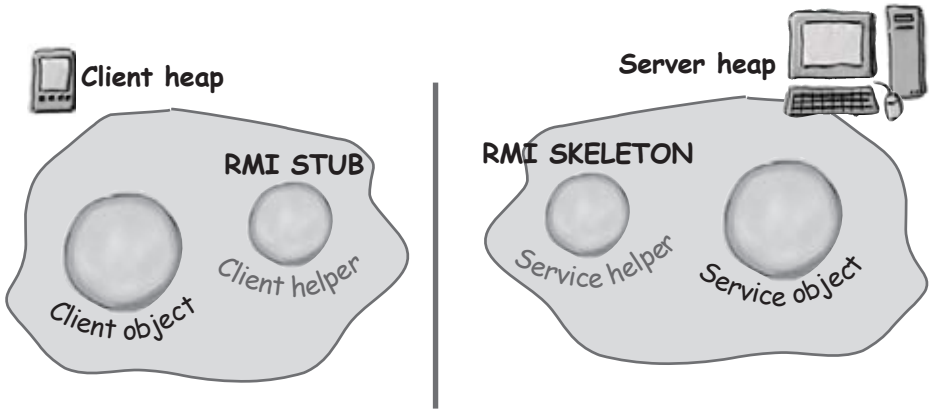
They throw exceptions all over the place.

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:

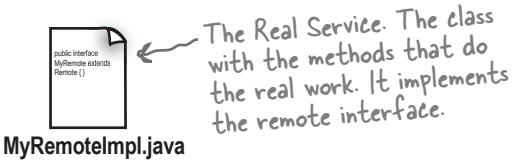
Make a Remote Interface

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:

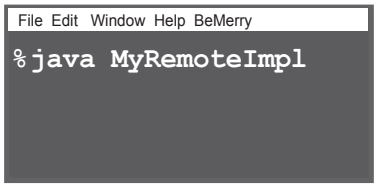
Start the RMI registry (rmiregistry)

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.

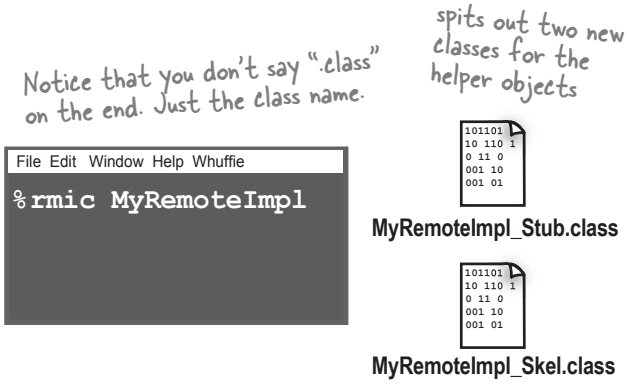
```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("Remote Hello", service);
} catch(Exception ex) {...}
```

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

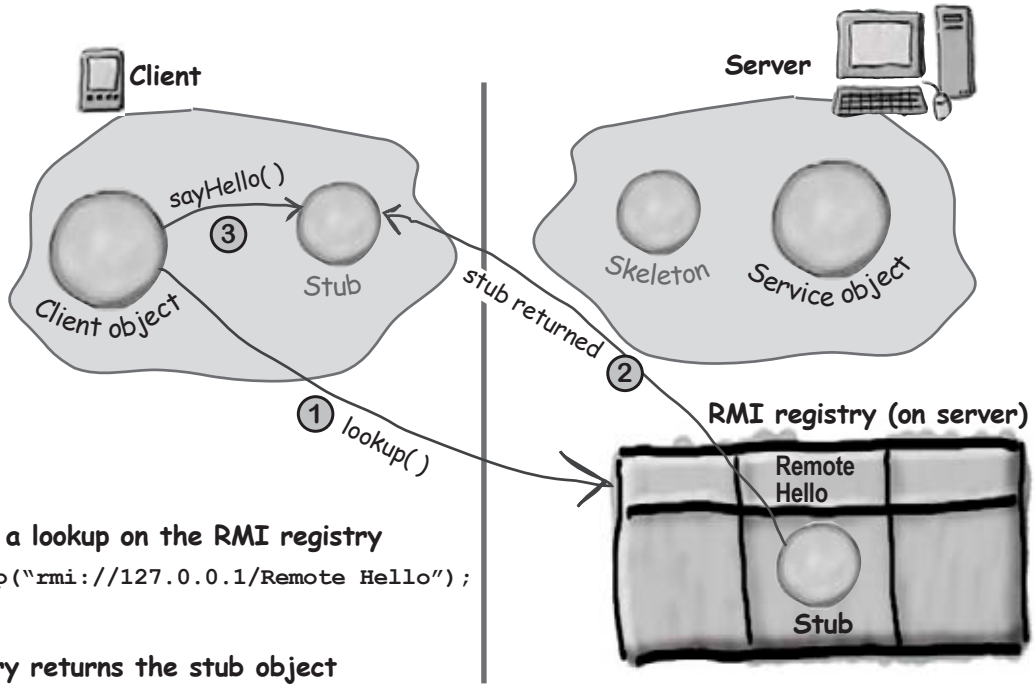
This must be the name that the service was registered under

```
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)

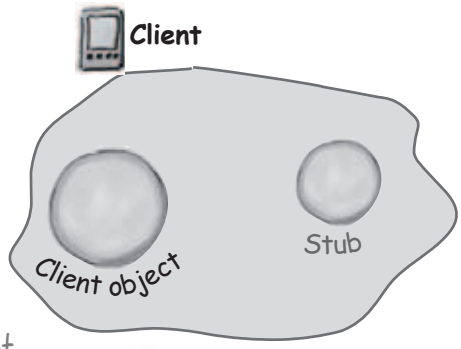
Be sure each machine has the class files it needs.

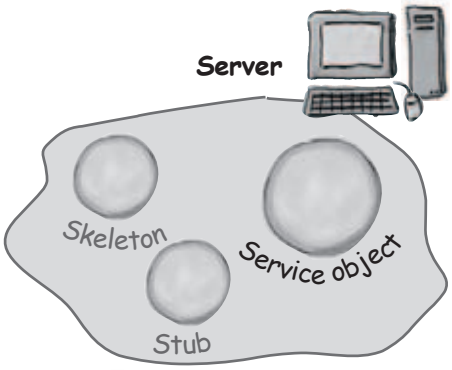
The top three things programmers do wrong with RMI are:

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

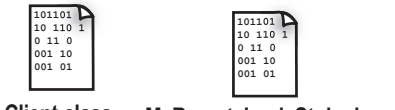
2) Forget to make arguments and return types serializable (you won't know until runtime; this is not something the compiler will detect.)

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

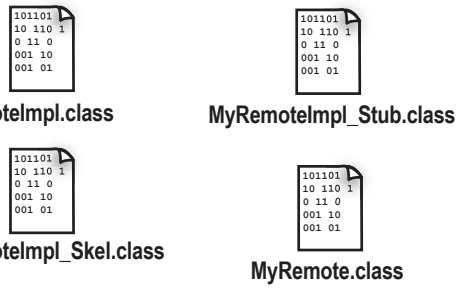




Don't forget, the client uses the interface to call methods on the stub. The client JVM needs the stub class, but the client never refers to the stub class in code. The client always uses the remote interface, as though the remote interface WERE the actual remote object.



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.