Introduction to Java Distributed Objects - Using RMI and CORBA

Welcome to the Course

Description:

This course introduces how to program distributed objects using Java Remote Method Invocation (RMI) and using the Common Object Request Broker Architecture (CORBA)

Prerequisites:

To get the most out of this course, you need to be an experienced Java programmer

Objectives:

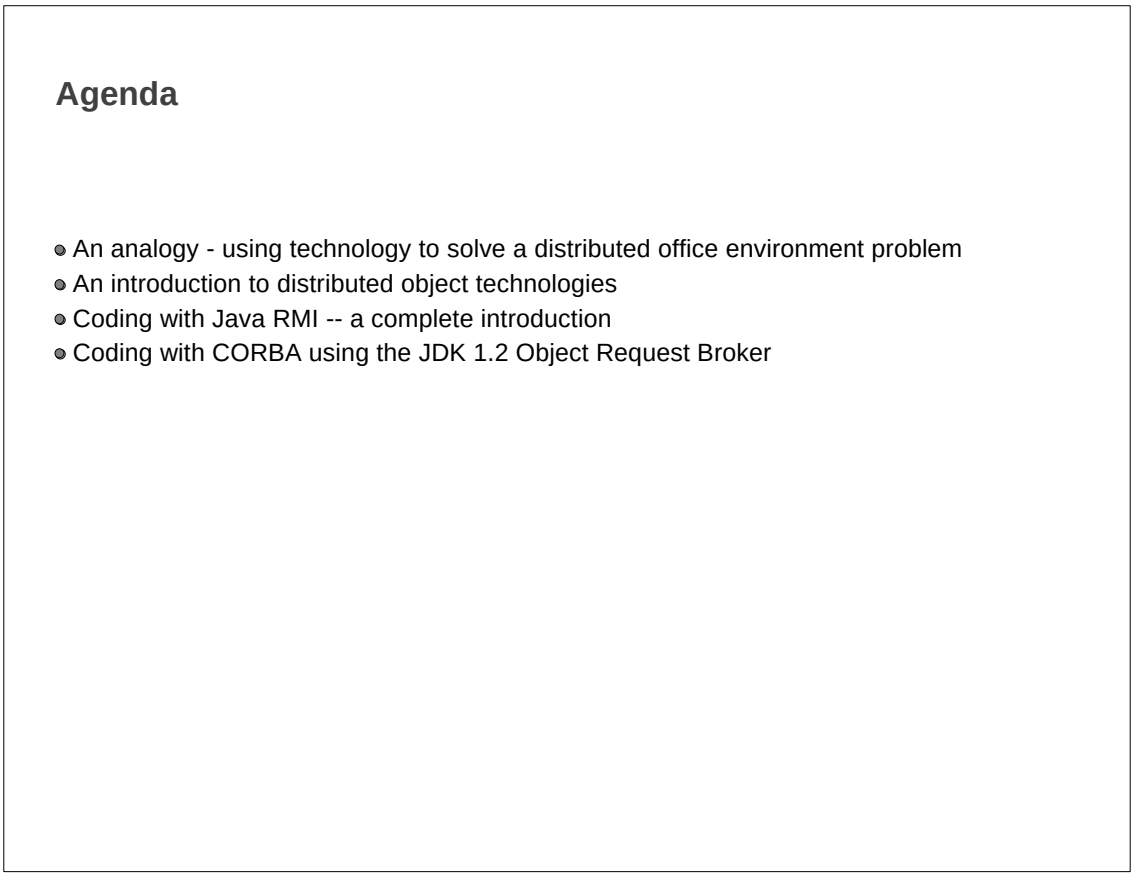
When you have completed the course, you should be able to:

- Contrast RMI and CORBA both at the conceptual and technical levels
- Write simple RMI and CORBA programs using the Java Development Kit (JDK) version 1.2

Welcome to the Java distributed objects course! Before we start, let's cover a few details. First of all, what should your background be? To get the most out the course, you already be a pretty good Java programmer. We will not cover any Java syntax. In particular, you should be comfortable with the notions of classes, interfaces, applications and applets. You don't need any background in distributed objects, however.

When you finish the course, you should have a pretty good idea about how distributed objects work in Java. You should also be able to code simple programs using either RMI or CORBA.

Now on to our agenda!



Here's what we will cover in this course: We'll start by using an analogy to introduce the notion of distributed processing - we'll look at a fictitious company with growing pains and how they used technology to make their distributed office environment work. Then we'll take these notions and use Java RMI and CORBA to solve a similar programming problem, that is, how to access objects that reside on different computers.

While doing so, you will see actual coding samples that use RMI and CORBA. We will place special emphasis on the deployment of these coding samples, since it turns out that deployment is the often the hardest part of using these technologies.

But let's get started by looking at the Twisted Transistor company, which needs to grow while remaining efficient.

Twisted Transistors, Inc. : the Beginning

Once upon a time, Jill and Bill started a new company named Twisted Transistor, Inc. to manufacture LCD panels for laptop computers. At the beginning, with only a few people in the company, and only one building, life was easy. When they needed to meet, they just walked over to someone's office and hashed things out. All of their work was done at one location, and communication was simple.

Twisted Transistors, Inc.: Growing Pains

But then a funny thing happened: the company grew, and soon found that it needed more space. The problem was, office space was expensive in their town, so they rented a building in Silver City, 10 miles away. This gave them more space to work and to manufacture product, but now things that were easy before became difficult.

For example, running a meeting was a nightmare compared to before. How could they get all of the meeting participants together? First they tried shuttling employees from one building to another, but that wasted too much time in transit. Then they tried conference calling, but found that to be awkward and it was difficult to show technical diagrams and presentations to each other.

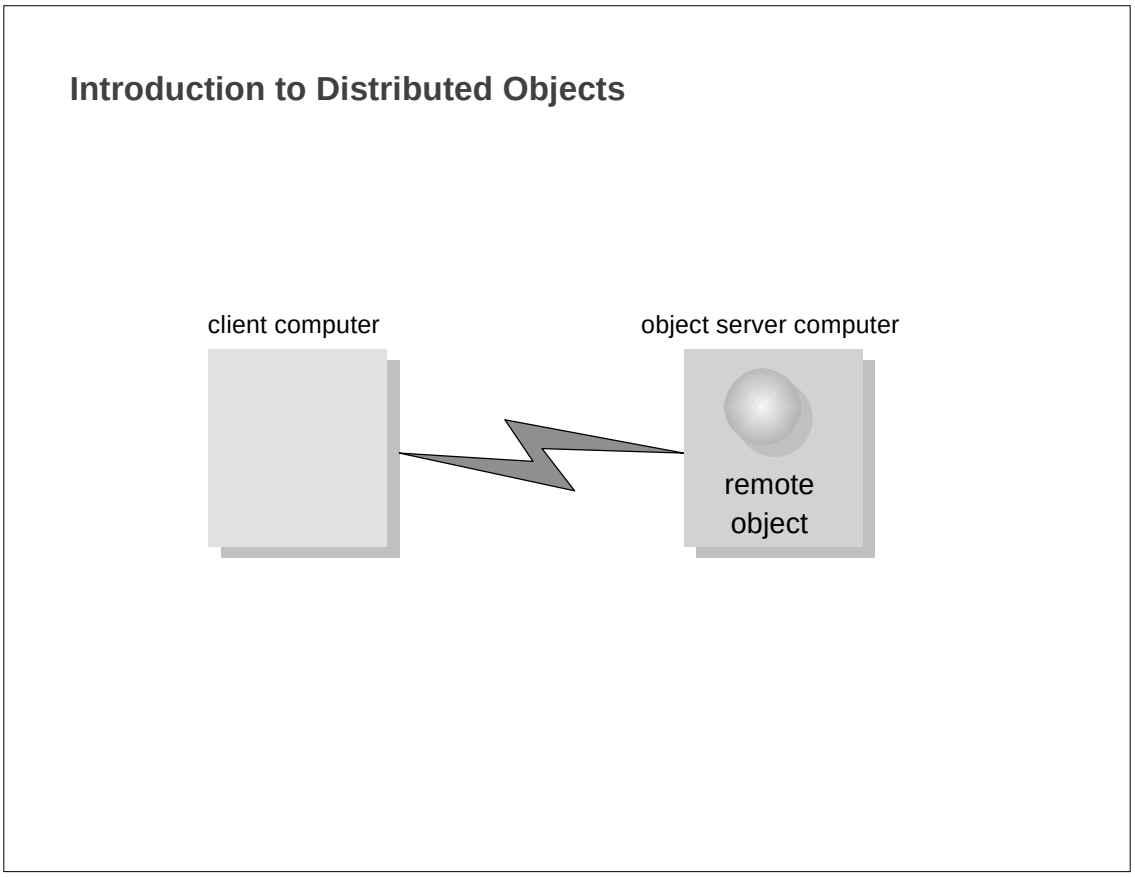
What Twisted Transistors needed was a way to make it seem like employees were together, when in fact they were spread out geographically.

Twisted Transistors, Inc.: Communication Nirvana!

To solve their meeting problem, Twisted Transistors purchased a communication system that included a virtual whiteboard and video teleconferencing. With the new system, even though the employees were located in another city, they could interact almost as if they were physically together. That way, the company could take advantage of the lower overhead costs in Silver City, but still all work as a group.

This solution required an infrastructure -- the two buildings needed to be connected with a leased-line network system and required special setup instructions to start the whiteboard and teleconferencing.

The system wasn't perfect, but it was the best available until Star-Trek-like transporters become available.

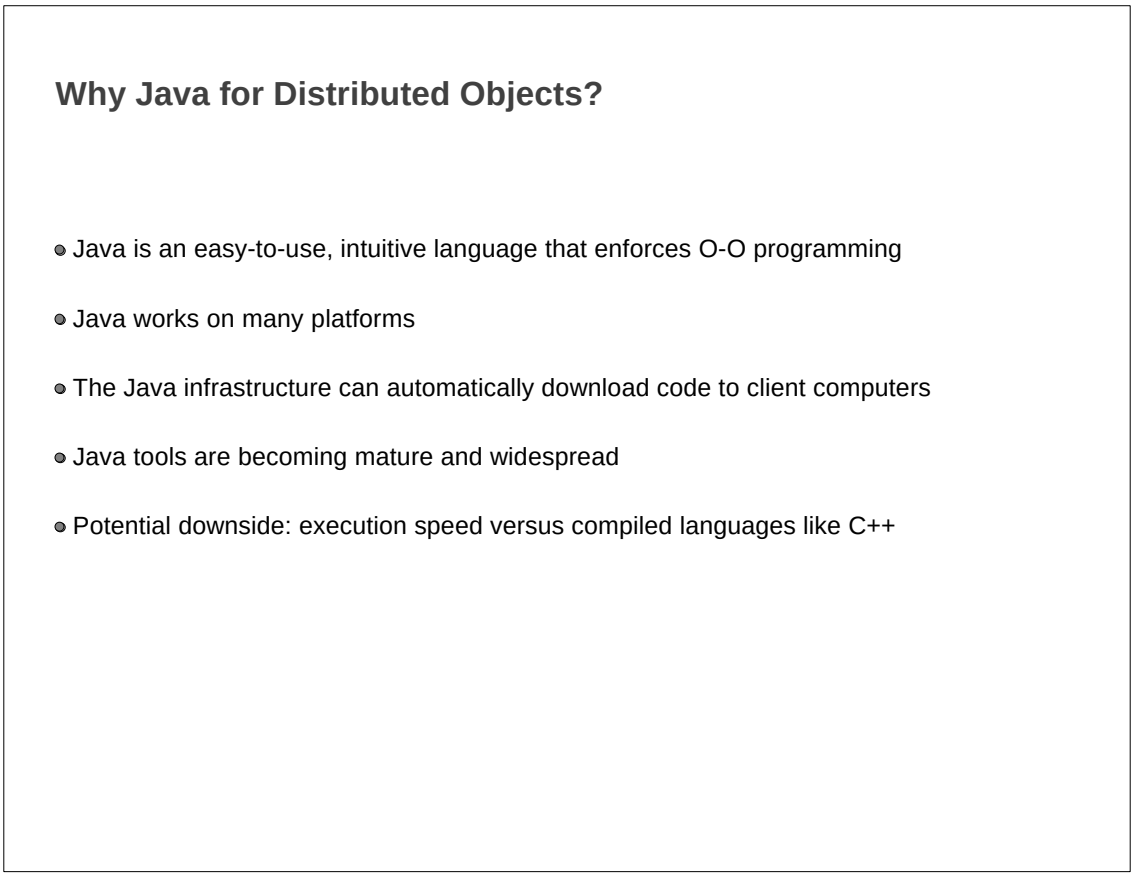


The big problem Twisted Transistor had was making people in a remote location easily accessible. We can have the same issue with applications that we write. Often, applications grow, just like Twisted Transistor did - the program might start out as a single-user program, but may expand to allow multiple people to access the program at the same time.

And remember, Java is an object-oriented language, so our real problem is: how do we work it so that programs running one computer can call methods on objects that reside on another? If we can pull this off, then we can distribute the computing load across networks - instead of writing standalone, single user programs, we can write programs that fully utilize our network.

Such programming is often referred to as client/server programming - the program issuing the method calls is the client, and the computer that supplies the remote object is the server.

Twisted Transistor solved their meeting problem by allowing people in separate offices to interact naturally - our goal here is to find technology so that remote and local objects can interact naturally.



If you've decided that distributed objects are a good thing, your next job is to choose a distributed object infrastructure and your programming language. We'll come back on the infrastructure part in just a moment, but now let's talk about the programming language.

This tutorial covers distributed objects using the Java programming language. Why Java? Because Java is ideally suited to the task. Most distributed object programs run on a mixture of operating systems, for example Windows for the client and a UNIX variant such as IBM's AIX for the server. Since Java itself is platform neutral, it lets you develop, test and deploy across different platforms without having to change languages.

In addition, the tool market for Java is exploding. Vendors such as IBM are developing powerful application servers that serve up Java distributed objects using technologies like Enterprise Java Beans. While we will not cover EJBs here, their use of Java helps to validate our choice of Java for this tutorial.

Finally, we would be remiss if we didn't at least mention that Java is not always the solution. During this program, we will discuss one infrastructure technology, CORBA, that lets you use compiled languages such as C++ whenever performance is the ultimate criteria.

Distributed Object Infrastructures

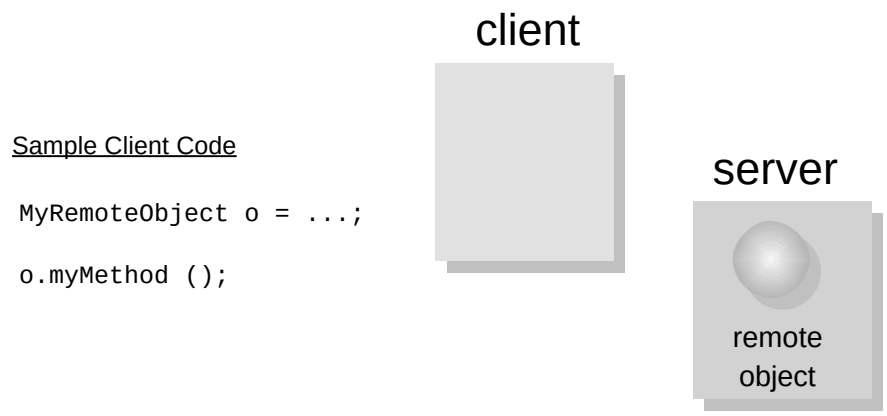
- Java Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)
- Microsoft's Distributed Component Object Model (DCOM)
- Enterprise Java Beans (EJB)

Here we show a list of some of the important infrastructure technologies for developing distributed objects. In this tutorial, we will concentrate on the first two and compare and contrast them.

That's not to say that other technologies have no merit; just that we will not cover them here.

Finally, many experts in the distributed object field believe that eventually, all of these technologies will interoperate, so that clients and servers of each type will be able to communicate with objects on servers of other types.

Introduction to RMI



Sample Client Code

```
MyRemoteObject o = ...;

o.myMethod ();
```

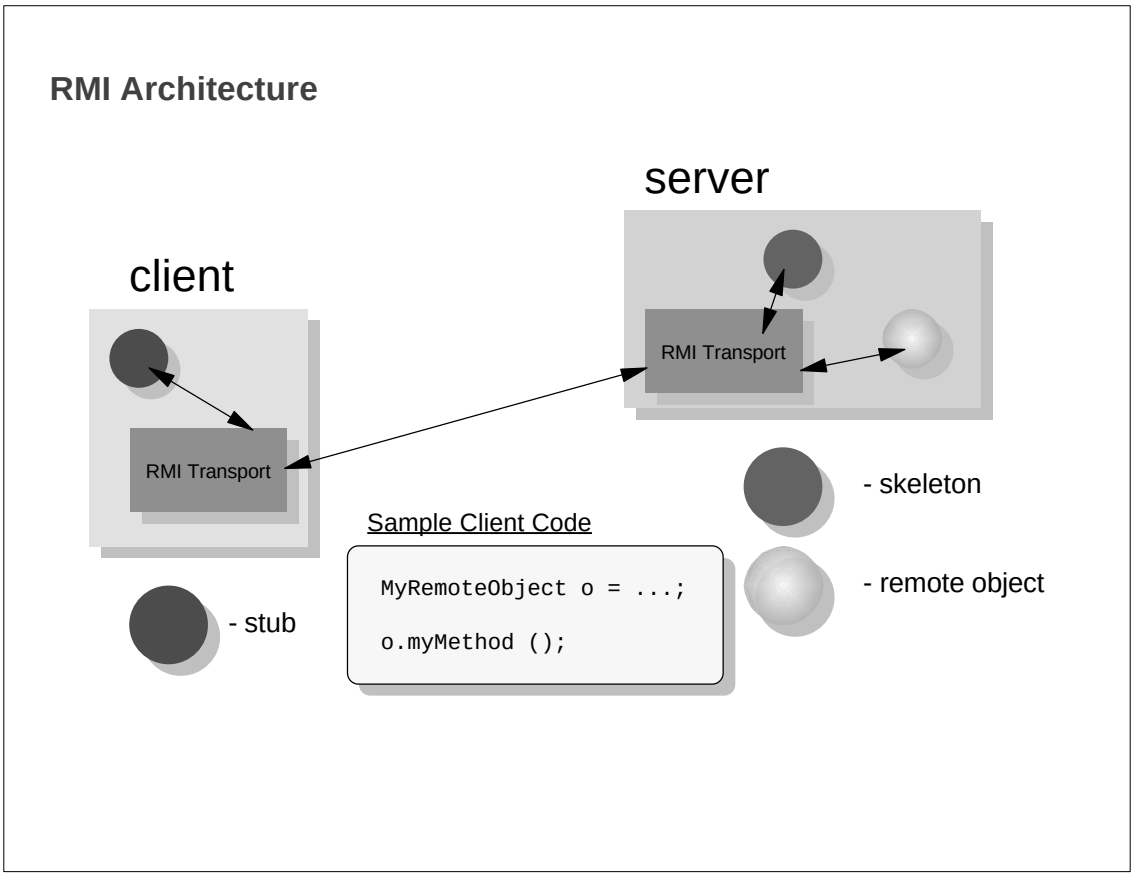
Note: All of the code in this course has been compiled and tested with JDK 1.2, Release Candidate 1. Please consult the documentation for the version of the JDK that you are using for details on changes for that version.

We will start our discussion of distributed object technologies with Java's Remote Method Invocation, which was introduced in Java 1.1.

RMI's purpose is to make objects in separate virtual machines look and act like local objects. The virtual machine that calls the remote object is sometimes referred to as a client. Similarly, we refer to the VM that contains the remote as a server.

Obtaining a reference for a remote object is a bit different than for local objects, but once you have the reference, you call the remote object just as if it was local as shown in the code snippet. The RMI infrastucture will automatically intercept the request, find the remote object, and dispatch the request remotely.

This location transparency even includes garbage collection. In other words, the client doesn't have to do anything special to release the remote object -- the RMI infrastructure and the remote VM handle the garbage collection for you.

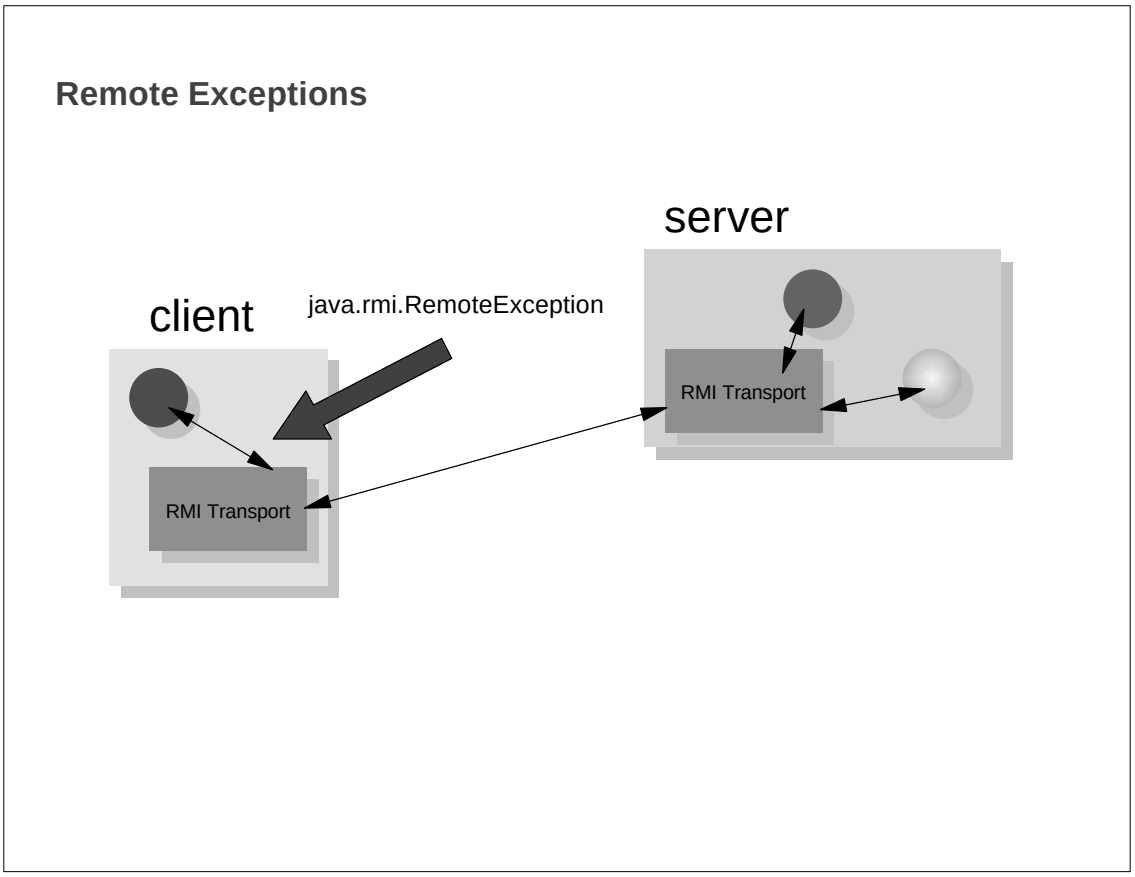


To achieve location transparency, RMI introduces two special kinds of objects known as stubs and skeletons.

The stub is a client-side object that represents the remote object. The stub has the same interface, or list of methods, as the remote object, but when the client calls a stub method, the stub forwards the request via the RMI infrastructure to the remote object, which actually executes it.

On the server side, the skeleton object takes care of all of the details of "remoteness" so that the actual remote object doesn't need to worry about them. In other words, you can pretty much code a remote object the same way as if it were local -- the skeleton insulates the remote object from the RMI infrastructure. During remote method requests, the RMI infrastructure automatically invokes the skeleton object so it can do its magic.

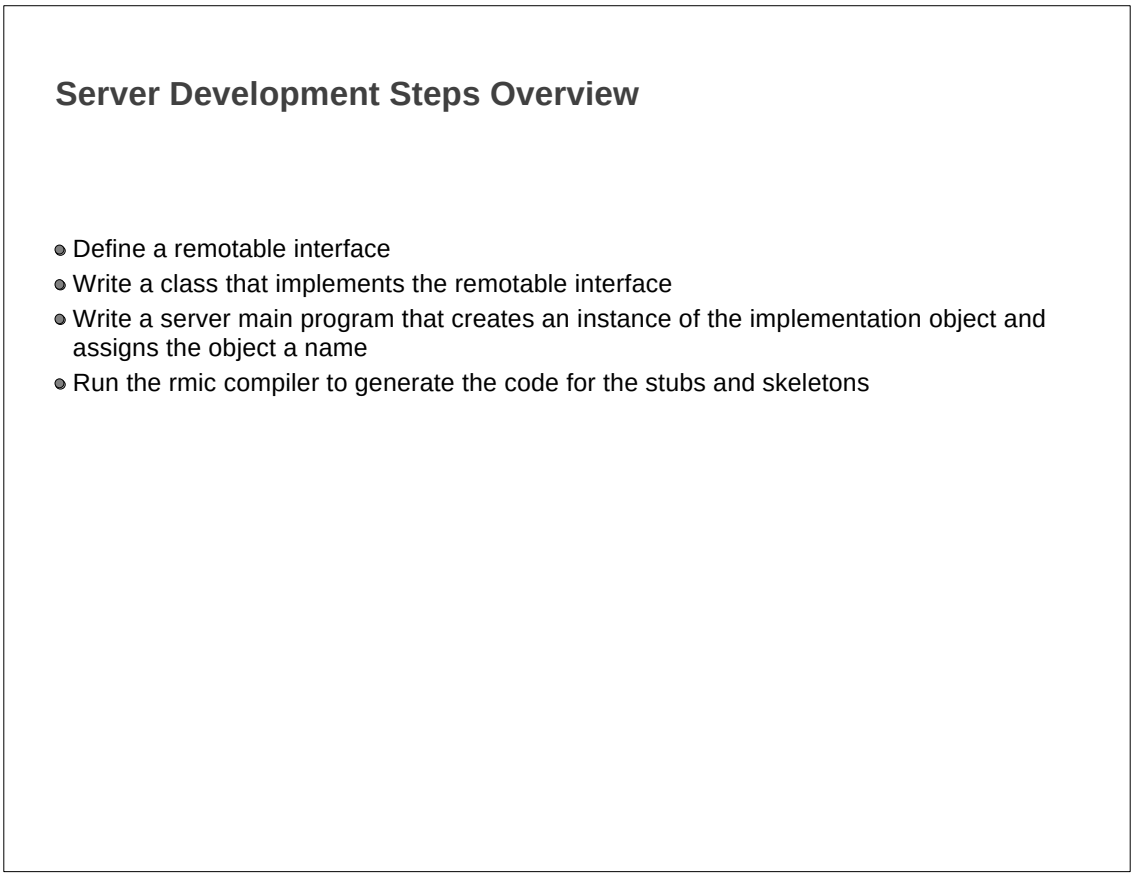
The best news about this setup is that you don't have to write the code for the stubs and skeletons yourself! The JDK contains a tool, rmic, that creates the class files for the stubs and skeletons for you.



Under the covers, RMI uses TCP/IP sockets to communicate remote method requests. While sockets are a fairly reliable transport, there are many things that could go wrong. For example, suppose the server computer crashes during a method request. Or, suppose that the client and server computers are connected via the Internet, and the client's 'net connection drops.

The point is that there are more things that can go wrong with remote objects than there is for local objects. And it's important that the client program be able to gracefully recover from errors.

So that the client knows about such errors, every method that will be called remotely must throw RemoteException, which is defined in the java.rmi package.



Now let's take a look at the steps involved in writing the object server. We will look at the client side in a few moments.

The first thing you need to do is define an interface for the remote object. This interface defines the methods that the client can call remotely. The big difference between a remoteable interface and local interface is that remote methods must throw the exception described on the last page.

Next, you must write a class that implements the interface.

Next, you need to write a main program that runs on the server. This program must instantiate one or more of the server objects and then typically registers the remote objects into the RMI name registry so that clients can find the objects.

Finally, you need to generate the code for the stubs and skeletons. The JDK provides the rmic tool, which reads the remote object's class file and creates class files for the stubs and skeletons.

Writing a Remote Interface

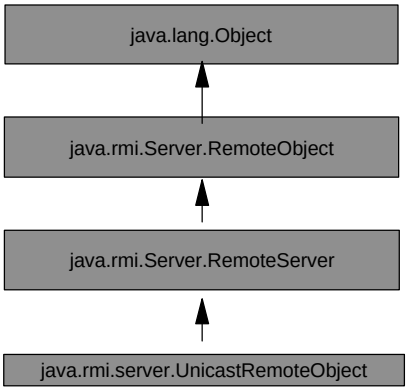
```
import java.rmi.*;

public interface Meeting extends Remote
{
    public String getDate ()
                  throws RemoteException;
    public void setDate ( String date )
                  throws RemoteException;
    public void scheduleIt()
                   throws RemoteException;
}
```

Here's the interface definition for a simple remote interface. Objects that implement this interface provide a three methods: one that returns a string, one that accepts a string as an argument, and one that accepts no arguments and returns nothing. As mentioned earlier, these methods must throw the RemoteException, which clients will catch if something goes wrong with the communication between the client and server.

Note that the interface itself extends the Remote interface that's defined in the java.rmi package. The Remote interface itself defines no methods, but by extending it, we indicate that the interface can be called remotely.

Implementing the Remote Interface

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

public class MeetingServer
    extends UnicastRemoteObject
    implements Meeting
{

    private String ivDate =
         new String ( "1/1/2000" );

    public MeetingServer()
         throws RemoteException
    {
    }

    public String getDate() throws RemoteException
    {
        return ivDate;
    }

  . . .
}
```



Now let's look at a class that implements the remote Meeting interface. As is typical, MeetingServer extends the UnicastRemoteObject class, which provides the basic behavior required of remote objects. The phrase "Unicast" refers to the fact that each client stub references a single remote object. In the future, RMI may allow for "Multicasting", where a stub could refer to a number of equivalent remote objects. With multicasting, the RMI infrastructure could balance the load between the set of remote objects.

Here we show the implementation of two methods: the "getDate" method defined in the Meeting interface, and a no-argument constructor. Notice that both throw the RemoteException; this is required for constructors and all methods that the client calls remotely. In this case, the constructor has no useful work to do, but we still need to define it so it can throw the remote exception.

The "getDate" method is the interesting one, though. It returns a string's reference back to the caller. While this may look simple, the RMI infrastructure and the skeletons and stubs actually have a lot of work to do here. They all must conspire so that a copy of the string is passed back to the client and then recreated as an object in the client's virtual machine.

Writing an RMI Server Overview

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

public class MeetingServer extends UnicastRemoteObject
      implements Meeting
{
    . . .

    public static void main ( String [] args ) throws
        RemoteException, java.net.MalformedURLException,
        RMISecurityException
    {
        // 1. Set Security Manager
        // 2. Create an object instance
        // 3. Register object into the name space
    }
}
```

In addition to implementing the interface, we also need to write the server's main program. RMI currently does not support server programs as applets, so the main program needs to be a standalone Java application. You can either code a separate Java class for main, or you can just code a main method in the implementation class as we did here.

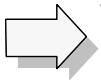
Note also that we coded the main function to throw any RMI-related exceptions to the command line. That's OK for a small sample program like this, but in a real program you will probably instead bracket the steps that follow in separate try-catch blocks so you can perform better error handling.

The steps that the server main typically does are:

1. Install a security manager class that lets the server program accept stub classes from other machines
2. Create an instance of the server object
3. Register the server object into the RMI naming registry so that client programs can find it

Let's now take a closer look at each of these steps.

Setting the Security Manager



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

public static void main ( String [] args ) throws
      RemoteException, java.net.MalformedURLException,
      RMISecurityException
{
      System.setSecurityManager (
                  new RMISecurityManager() );

      MeetingServer ms = new MeetingServer();

      Naming.rebind (
             "rmi://myhost.com/Meeting", ms );
}
```

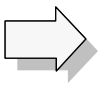
The first step is to install the RMI security manager. While this is not strictly required, it does allow the server virtual machine to download class files. For example, suppose the client calls a method in this server that accepts a reference to an application-defined object type, for example a BankAccount. By setting the security manager, we allow the RMI runtime to copy the BankAccount class file to the server dynamically - that eases the configuration on the server.

The downside to letting RMI dynamically download such classes is that it's a security risk. In other words, we are essentially letting the server execute code from another machine. While we hope that these class files are not going to harm the server, if you want to avoid the risk, your RMI server should not install a security manager. You must then ensure that all class files are installed locally in the server's classpath.

And now just an aside before we move on: passing object argument types is actually a pretty involved topic, since there are two ways to do it. One way is to pass only a reference across the communication wire; the other is to serialize the object and create a new object remotely. We will not discuss these any further since we don't have enough time, so you should read the RMI documentation in the JDK for more details.

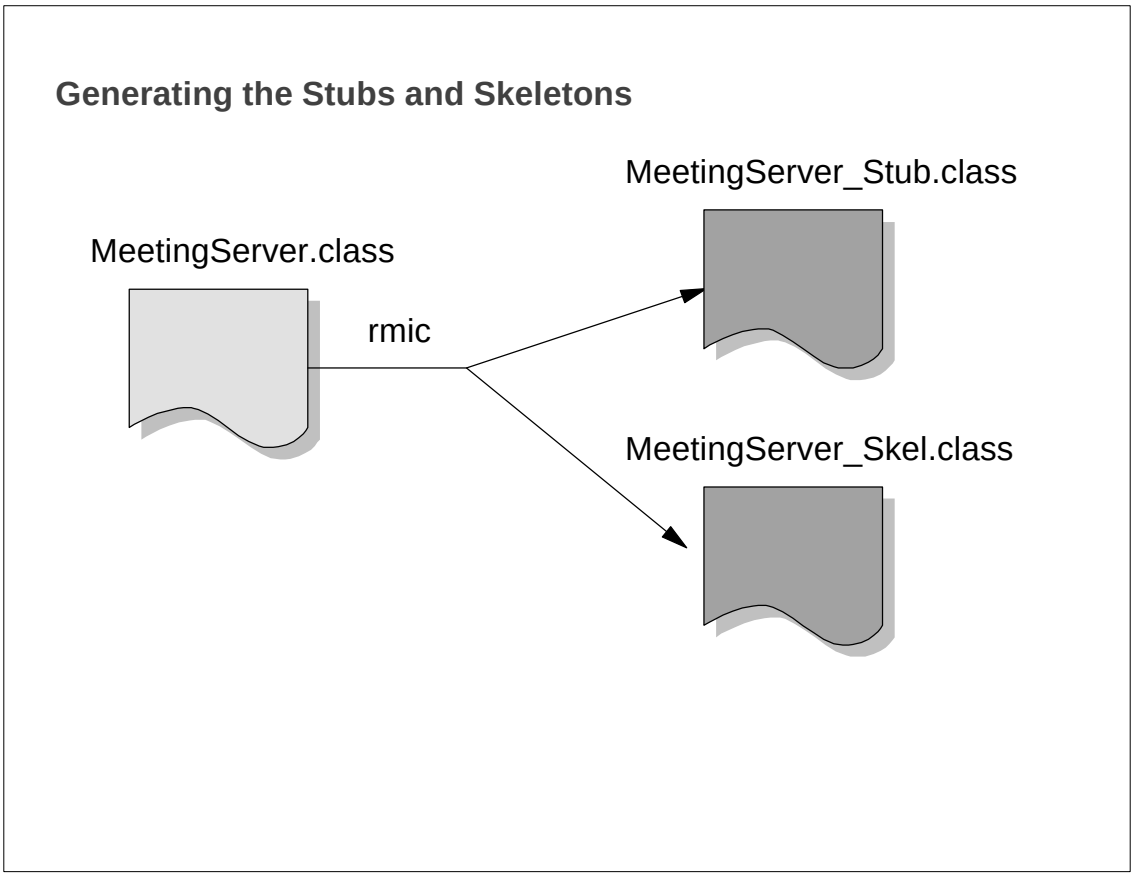
Naming the Remote Object

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

public static void main ( String [] args ) throws
      RemoteException, java.net.MalformedURLException,
      RMISecurityException
{
      System.setSecurityManager (
                  new RMISecurityManager() );

      MeetingServer ms = new MeetingServer();

      Naming.rebind (
            "rmi://myhost.com/Meeting", ms );
}
```



The next server job is to create an initial instance of the server object and then write a name for the object into the RMI name registry. The RMI name registry lets you assign URL names to objects so that clients can look them up To register the name, call the static rebind method, which is defined on the Naming class. This method accepts the URL name for the object and the object reference.

The name string is the interesting part. It contains the rmi:// prefix, the hostname of the computer where the RMI object's server runs, and the object's name itself, which is pretty much whatever you want. Note that instead of hard-coding the hostname as we did here, you can call the getLocalHost method defined by the java.net.InetAddress class.



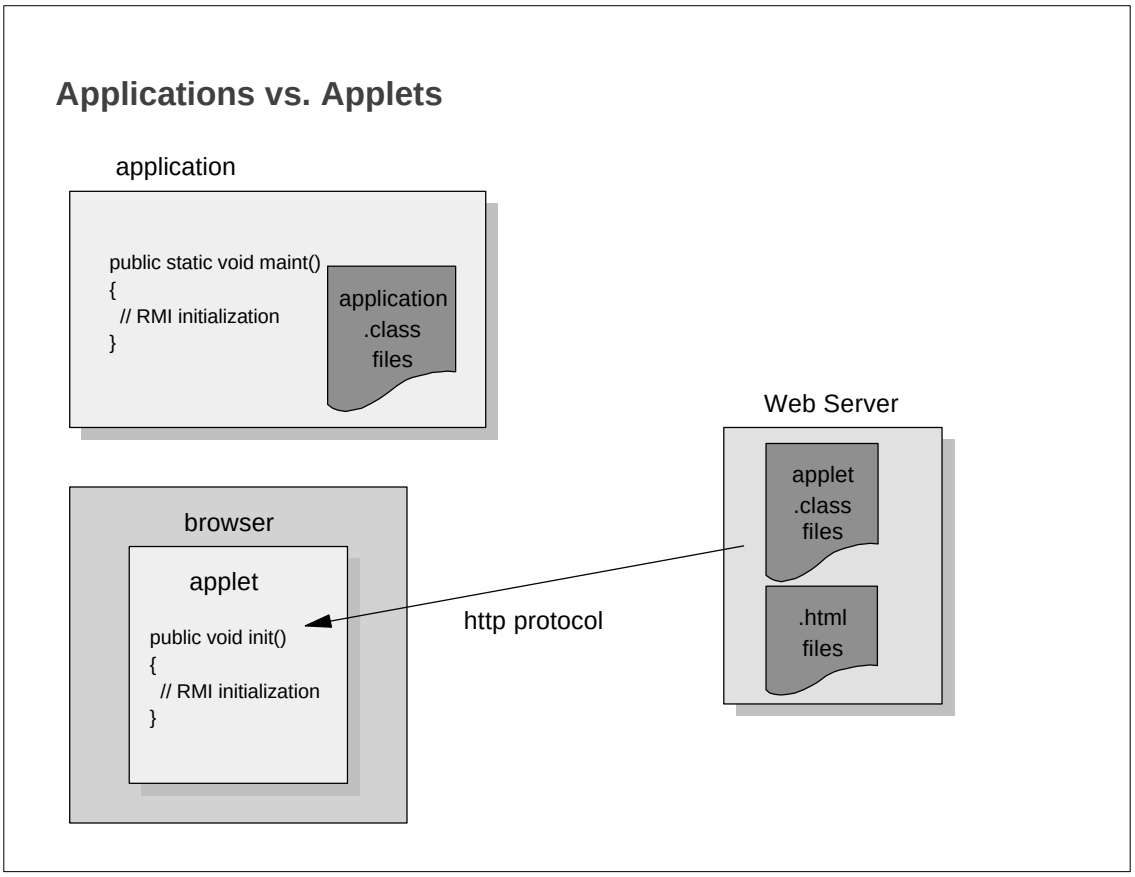
After writing and compiling your server implementation, you're ready to create the stubs and skeleton classes. And that's easy to do: just run the JDK's rmic command, specifying the name of your implementation class file (without the extension). RMIC will create a class file each for the stub and skeleton. You will then need to deploy these files properly, which we will cover after we cover writing the client-side code.

Client Development Overview

- Determine whether you want to write a client application or applet
- Write the client to look up the object in the naming registry
- Call remote methods

Now let's take a look at the client side. First, you must determine whether you want to write a client standalone application or a client applet. Applications are a bit simpler to setup, but applets are easier to deploy since the Java and RMI infrastructure can download them to the client machine. We will cover how to do both kinds.

In your client, the code needs to first look up the remote object using the RMI registry. Once you have done so, the client can then call the methods defined by the remote interface.



Let's take a quick look at the differences between Java applications and applets. If you write an application, you must define a main entry point in which you can execute the RMI startup code. You must then install the application's class files on the client machine. If you have multiple client computers, you must manually install the application's class files on each. And, as we will see in a while, you will also need to install some RMI-related server files on the client application computer.

If you decide to write an applet instead, you won't have a main entry point. Instead, the applet overrides the "init" method, which is called by the browser. You can do the same sort of code in init that you would in an application's main. The main advantage of applets is that you don't need to pre-install anything on the client computer except for the browser - the RMI and Java infrastructure will download all of the required class files automatically. Note that you do need to write an HTML file that the browser can load, though - we will cover all of this coming up.

Writing an RMI Client Application

```
import java.rmi.*;

public class MeetingClient
{
    public static void main ( String [] args )
        throws RemoteException,
               java.net.MalformedURLException,
               java.rmi.NotBoundException
    {
        // 1. Set Security Manager
        // 2. Look up remote object from name space
        // 3. Call remote methods
    }
}
```

Now let's examine the steps involved in writing an RMI standalone application. As usual, we code the "main" method as the entry point, and in it, we perform the RMI initialization steps. We first set the security manager so that the RMI runtime can download class files, then obtain a reference to the remote object by using the RMI naming registry. Then we can call remote methods.

While this code shows calling remote methods only in the "main" method, once your application has retrieved the object reference, the application can call methods during other methods, too. In addition, note that there's no clean-up required; the Java and RMI infrastructure work together to make sure that garbage collection work, even on remote objects.

We will now examine these steps in more detail.

Setting the Security Manager

```
import java.rmi.*;

  public static void main ( String [] args )
      throws RemoteException,
             java.net.MalformedURLException,
             java.rmi.NotBoundException
  {
  System.setSecurityManager(new RMISecurityManager());

  . . .

  }
```

This code looks similar to code in the object server main - like the object server, a client application also can optionally set a security manager. And the reason is similar too: the RMI runtime will automatically download the remote object's stub class file to the client, but can only do so if the application installs a security manager. If the application uses the default security manager, you will need to pre-install the stub class files in the client computer's classpath or else the application will catch a security exception.

You should also note that in this code, the main method simply throws RMI-related exceptions back to the command line. A more robust application would contain try-catch blocks to localize error processing.

Looking up an Object

```
import java.rmi.*;

  public static void main ( String [] args )
      throws RemoteException,
             java.net.MalformedURLException,
             java.rmi.NotBoundException
  {
      System.setSecurityManager(new RMISecurityManager());

      Remote r = Naming.lookup (
                 "rmi://myhost.com/Meeting" );

  . . .

  }
```

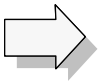
Once the application has installed a usable security manager, it can then retrieve the remote object's reference from the RMI naming registry. To do so, call the static "lookup" method, passing the same name under which the object server registered the remote object. The "lookup" method will retrieve a reference to the remote object and create a stub object, here stored in the variable named "r".

If the name supplied by the client doesn't match a name in the registry, the "lookup" method throws the NotBoundException. If the supplied URL is not valid, "lookup" throws MalformedURLException. In this simple code snippet, we don't explicitly catch these exceptions, but you would in a real-world program.

Note the type of the returned reference: it's of type Remote, which is the superclass of all remote objects. But we really want a reference to a Meeting interface as defined in the remote interface. Therefore, we will need to down-cast the reference as described on the next page.

Calling Remote Methods

```
import java.rmi.*;

  public static void main ( String [] args )
      throws RemoteException,
             java.net.MalformedURLException,
             java.rmi.NotBoundException
  {

  . . .
      Remote r = Naming.lookup (
               "rmi://myhost.com/Meeting" );

      String s = null;

      if ( r instanceof Meeting )
      {
          ms = (Meeting)r;
          s = ms.getDate();
      }
  }
```



Before the application can call remote methods, it must convert the remote reference's type to match the interface definition, in this case, HelloInterface. While you can do this with a simple cast, the code shown here first checks to see if the cast is valid by calling the instanceof operator. This technique lets you avoid an InvalidCast runtime exception if the type of the remote object isn't what you expect.

Once we have successfully casted the reference, we can then call remote methods; here the "getDate" method which returns a string.

Writing an RMI Client Applet

```
import java.rmi.*;

public class MeetingClientApplet extends java.applet.Applet
{
    public void init ()
    {
        try
        {
          // 1. Look up remote object in the RMI registry
          // 2. Call remote methods (can also call from
          //    other methods if you save the reference)
        }
        catch ( Exception e )
        {
        }
    }
}
```

Now let's look at the differences when writing an RMI client as an applet rather than as a standalone application. There are three basic differences:

1. You typically code the RMI initialization in the applet's "init" method rather than in "main"
2. Since you cannot change the method signature of init (you are overriding it from Applet), you cannot code "init" as throwing exceptions. Thus you must handle exceptions with try-catch blocks
3. You don't have to install the RMI security manager, since applets automatically use the AppletSecurityManager, which allows downloading of remote classes

Despite these changes, the RMI-related code in an applet client is basically the same as in an application. You still use the RMI naming registry to find remote references, cast the returned reference to the correct type, and so forth.

One other difference between applets and applications -- to use an applet from within a browser, you need to write an HTML page that references the applet. Let's look at that next.

Writing the HTML Page for the Applet

```
<HTML>
<title>Meeting Client Applet</title>

. . .



<applet code="MeetingClientApplet"  width=500 height=120>
</applet>

</HTML>
```

Here's a simple HTML file that loads the applet shown on the last page. The key line is the "applet" tag, in which we specify the applet's class file (no file extension). We will then need to install the HTML file and the applet class file on a web server. When the user displays this page in their Java-enabled browser, the browser will download the applet class file to the client computer and call the "init" method, which in this case, commences RMI communication.

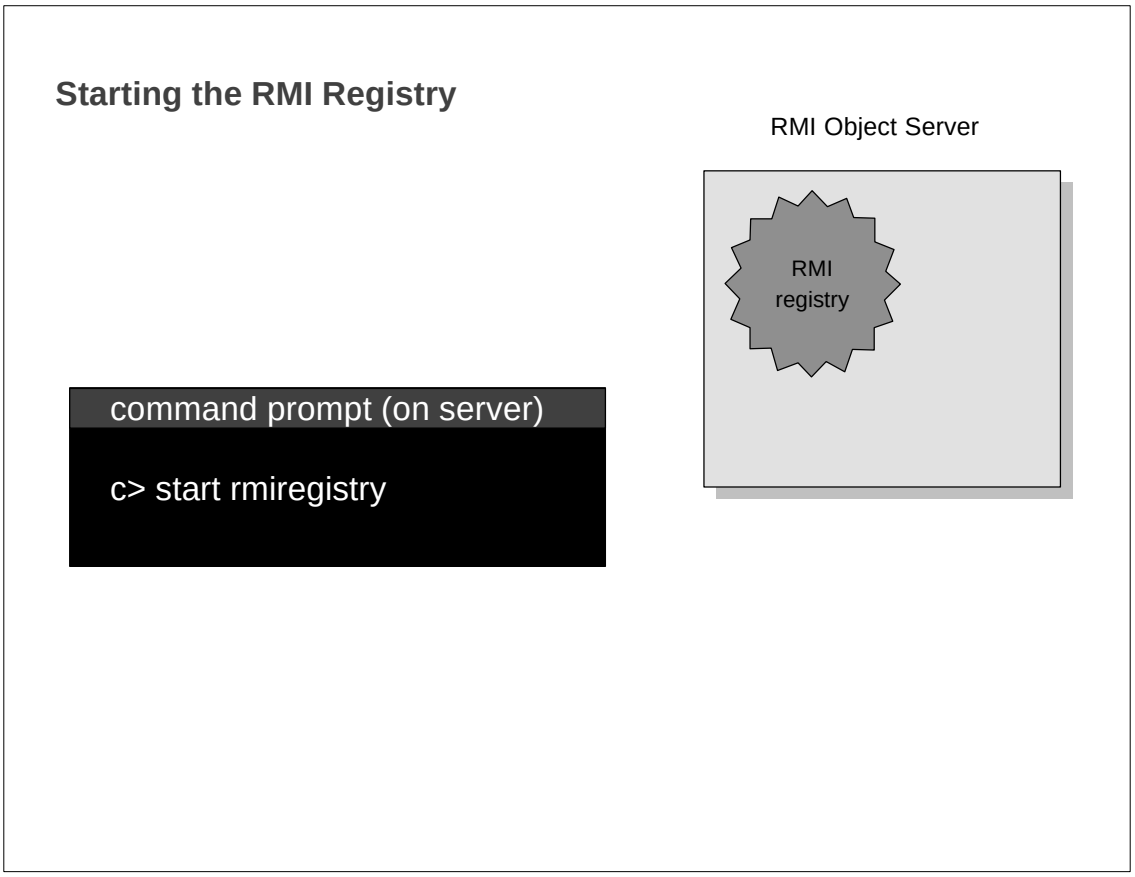
Our next task is to examine how to deploy the RMI code. We will look at both cases: deploying client standalone applications, and deploying client applets.

Deployment Overview

- Start the RMI Registry running
- Start the object server running
- Set up the web server
- Run the client application or applet

Let's overview the steps involved in deploying an RMI program. We must first start the RMI registry program running on the RMI object server, then we can start the object server main program. We also need to set up a web or ftp server so that the RMI runtime can download class files as described earlier. Finally, we can run the client application or display the applet's HTML page in a browser.

Now let's examine each step in more detail.

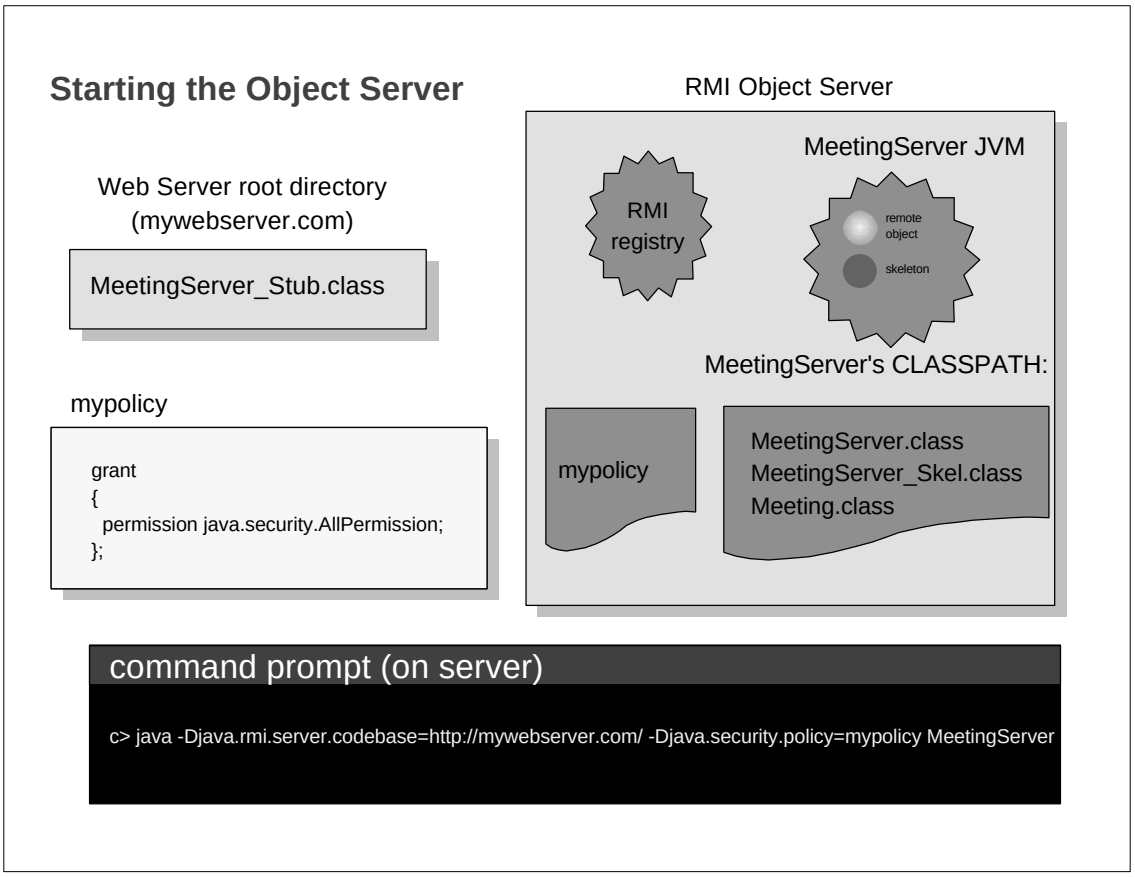


The first deployment step is to start the RMI naming registry on the object server computer. The rmiregistry command is included in the Java Development Kit.

Here we show the command in Microsoft Windows to start the registry in a separate window - see your operating system's documentation for specific instructions.

The registry program manages the mapping of names to object references so that your programs can call Naming.bind to register names and Naming.lookup to retrieve references.

One caution: make sure that the classpath in effect when you start the registry does not reference the stub class files on the server. If it does, then the RMI infrastructure will not download the stub automatically to the client computer.



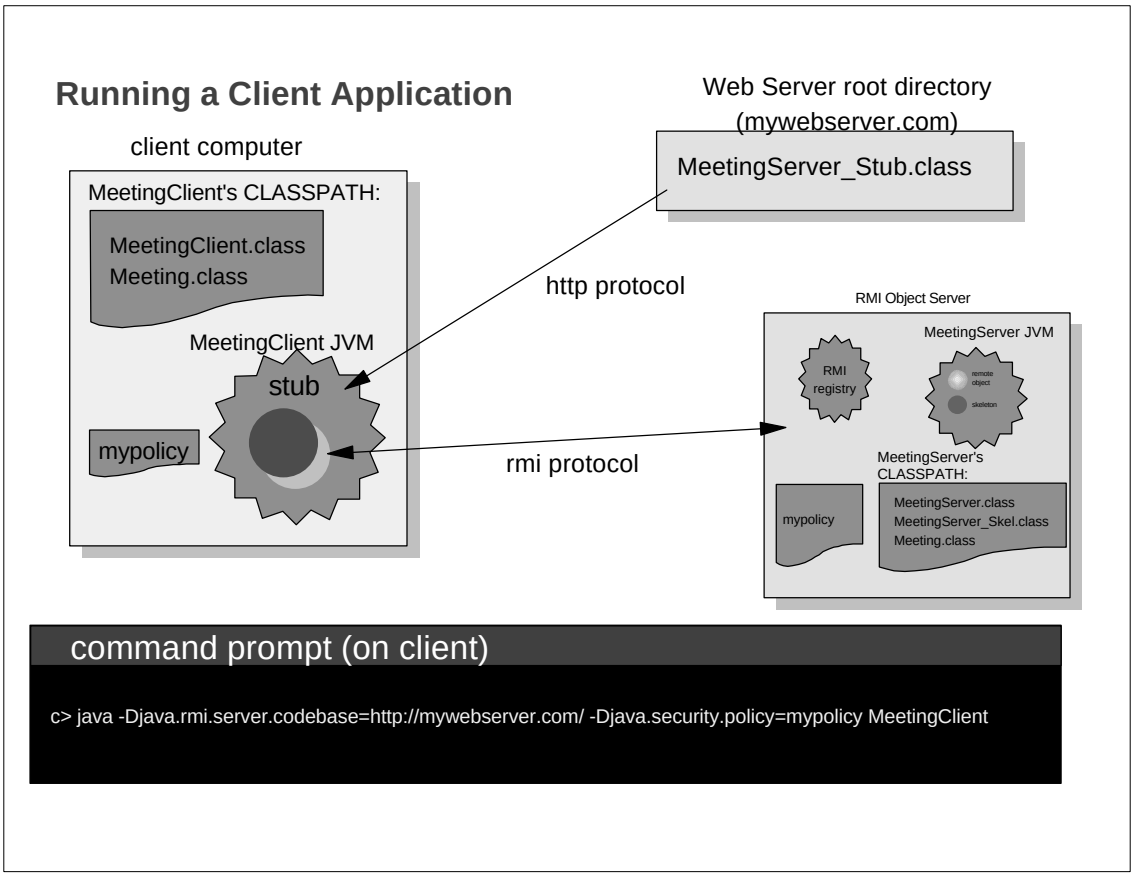
After starting the registry, we can next start the RMI object server main program on the server computer. Like starting the registry, this step is independent of whether the client is an applet or an application.

Remember that we coded the main method to create an object and register a name for the object in the RMI registry. The diagram shows the MeetingServer virtual machine containing the MeetingServer remote object as well as its skeleton.

The object server virtual machine needs access to all of the class files shown in the diagram: the MeetingServer main program, the skeleton class file, and the interface class file. Therefore, before starting the server program, you must ensure that all of these files are in the classpath. The stub class file must be installed on a web server; note that we reference the web server in the command to start to the object server main program. In this case, the codebase property specifies only a hostname, not a directory, so the web server will serve from its default directory. We will direct you to your web server's documentation for more details on how to setup directories.

Starting in JDK 1.2, RMI servers also need to reference a policy file that grants or denies permissions for the remote object. In this case, we created a policy file that grants all permissions; that's OK for a sample program like this one, but it's probably too dangerous for a production environment. Note that we must specify the policy file when we start the object server.

As before, this page shows the Microsoft Windows command to run the object server main program - see your operating system's documentation for specific details.



Now let's look at running the client program. We'll start by looking at the client as a standalone application, followed by a discussion of client applets.

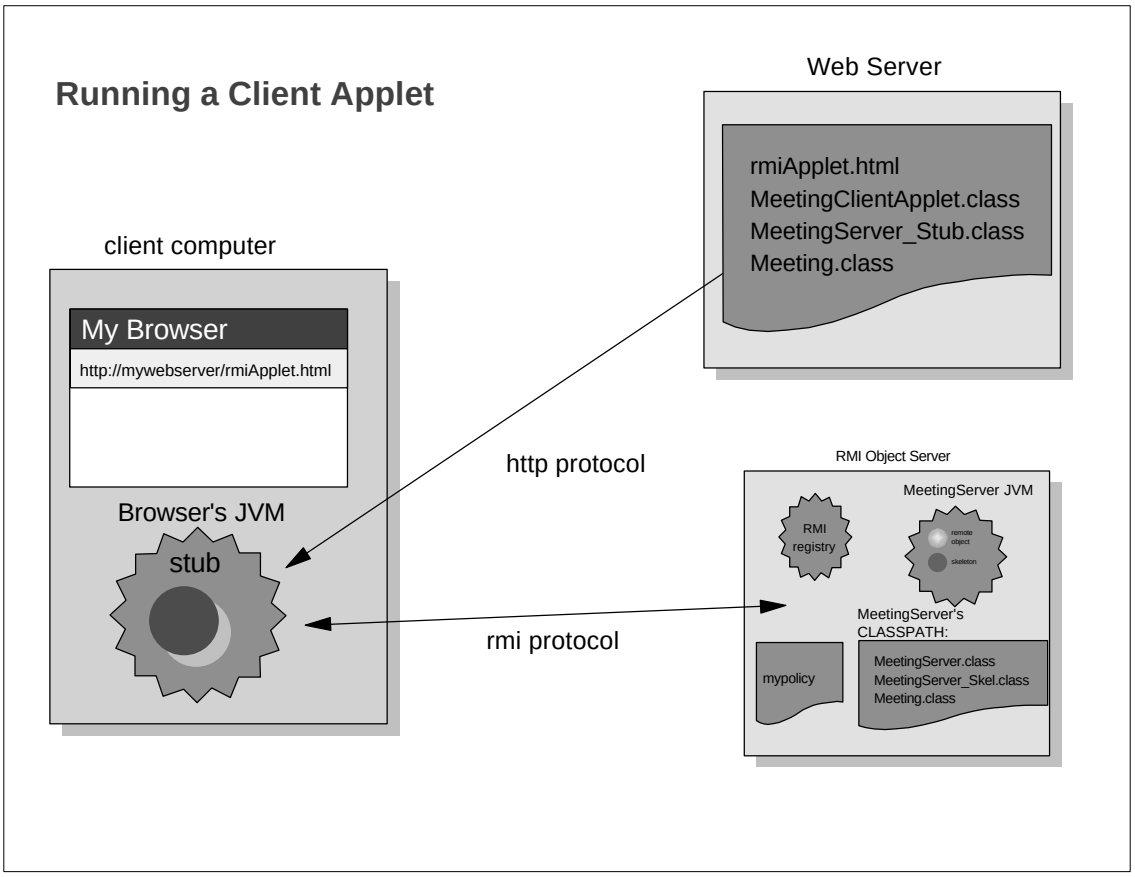
The virtual machine on the client computer needs access to the class files for the client program itself and for the remote interface - you must install these files in the client's classpath. The client also needs access to the stub class, but you don't need to install this file on the client - the RMI runtime will automatically download stub classes on demand. Remember to take advantage of this feature, we needed to install a security manager in the client application. So instead of installing the stub class file on the client, we can place it on a computer that's running a web server.

When we run the client application (see the command prompt), we specify the codebase property to indicate where to find the stub class files. If you don't want to have the stubs downloaded dynamically, or if you don't have a web, you can instead manually copy the stub class files to the classpath on the client computer.

Note that once the stubs have been downloaded, all further interaction with the RMI server uses the RMI protocol. This includes lookups in the naming registry and remote method calls and returns.

And as we saw with the server, in JDK 1.2 and later, the client needs to use a policy file. Here we have copied the same policy file as used on the server to the client so we can easily reference it on the command line.

Note: it's also possible to have the stubs downloaded to the client using the ftp protocol instead of http. See the JDK documentation for details.



Running a client applet is quite a bit different from running a standalone application. One difference is that you typically run applets from within a browser running on the client computer. Of course, the browser needs to support Java!

Another difference is that since applets already support downloading of class files, you don't need to pre-install anything on the client computer except for the browser! Instead, you can place all of the required HTML and class files on a web server, and the applet and RMI infrastructure will download them as needed. We also don't need to worry about the policy file on the client.

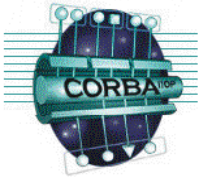
There is one other complication with regards to applets: if you want the object server and the web server to reside on different hosts, you will need to learn about digitally signing the applet. That's because, by default, the browser security manager only allows network access, including RMI methods, back to the web server. In other words, unless you sign your applet, you will see security violations if you distribute your objects to a different server than the web server. Digitally signing is a topic beyond the scope of this short course.

Finally, we should note that its actually possible to write standalone applications so that just about everything can be downloaded upon demand, similar to as shown here for applets. Consult the JDK documentation regarding "bootstrapping applications" for more details.

Well, we have now covered enough about RMI to get you started writing and deploying your own distrubuted programs. Let's now look an alternative infrastructure for distributed objects -- the CORBA approach.

Introduction to CORBA

Object
Managment
Group



Common
Object
Request
Broker
Architecture

Now that we've covered Java's RMI, let's look at another way to distribute objects. Like RMI, CORBA lets you write client code that accesses remote objects. However, there are some significant technical differences as we will detail on the next page.

The biggest difference though is that, unlike RMI, CORBA is not a part of Java itself. Instead, an industry consortium known as the Object Management Group invented CORBA. The OMG itself does not write code; instead, the consortium debates and publishes specifications. Then, vendors implement the specifications to produce products known as Object Request Brokers (ORBs). Some of the major ORB vendors include IBM, Iona, and Inprise (formerly known as Borland, which purchased the Visigenics ORB company). But the most exciting ORB news for Java developers is that the JDK version 1.2 includes an ORB, so you can write Java CORBA code without having to license any other software. We will use the JDK 1.2 ORB as our example ORB in this course.

CORBA vs RMI

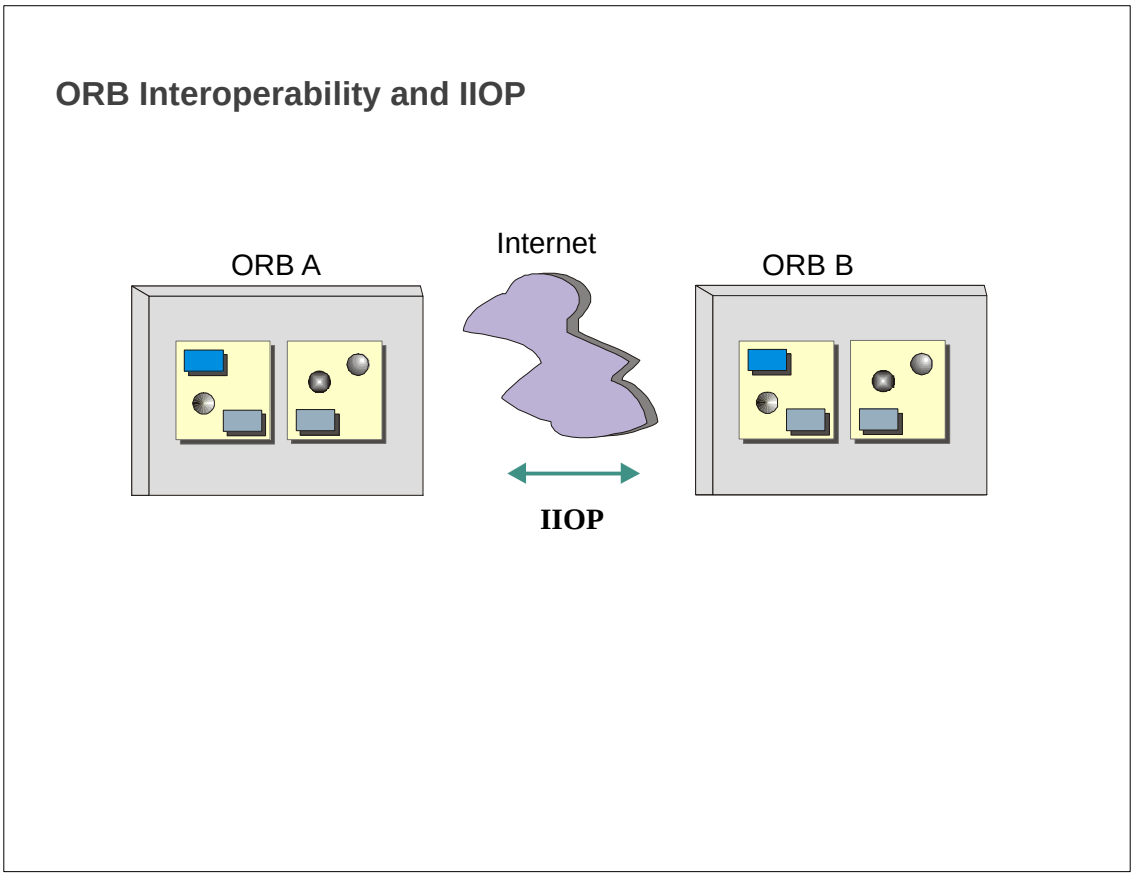
	RMI	*Java*
Languages supported	Java	Java, C++, C, Smalltalk, etc.
Runtime services	Naming	Naming, Lifecycle, Persistence, Transactions, etc.
Ease of programming and setup	Excellent	Good
Scalability	Good	Excellent (depending on ORB vendor)
Performance	Good	Excellent (depending on ORB vendor)

Before we cover CORBA in detail, let's compare it technically with RMI. The basic trade-offs between the technologies are:

- CORBA lets you mix programming languages. In other words, you can write your client programs and the distributed objects in any CORBA supported programming language, including Java, C++, C and so forth. RMI, of course, only supports Java, both on the client and the server.

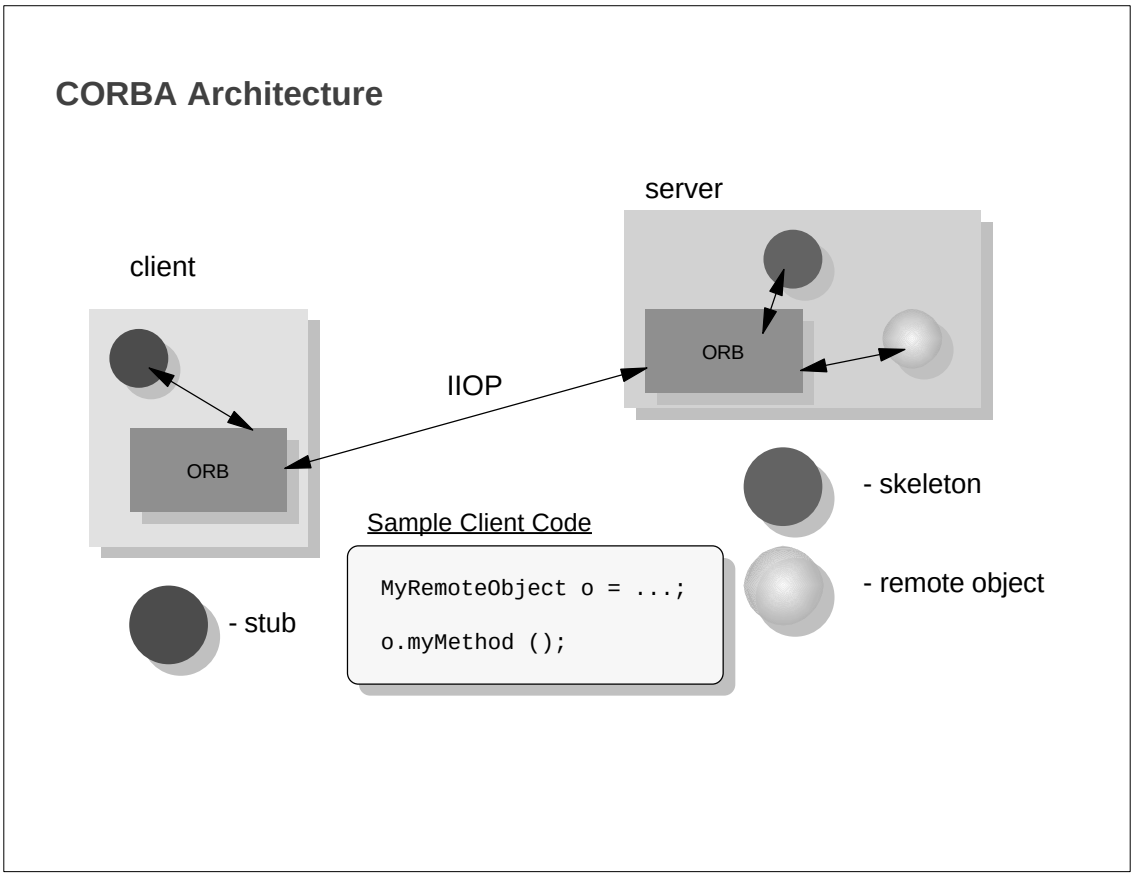
- CORBA is geared toward larger, more scalable systems, where you might have thousands of objects. CORBA is more complex to program and deploy than RMI, but lets you develop enterprise-level systems, where you need support for transactions, security and so forth. The CORBA Naming Service is also more powerful and flexible than the RMI Naming Registry.

So which is better? It really depends on your intended application. Hopefully, you will be able to make a more informed choice after taking this course.



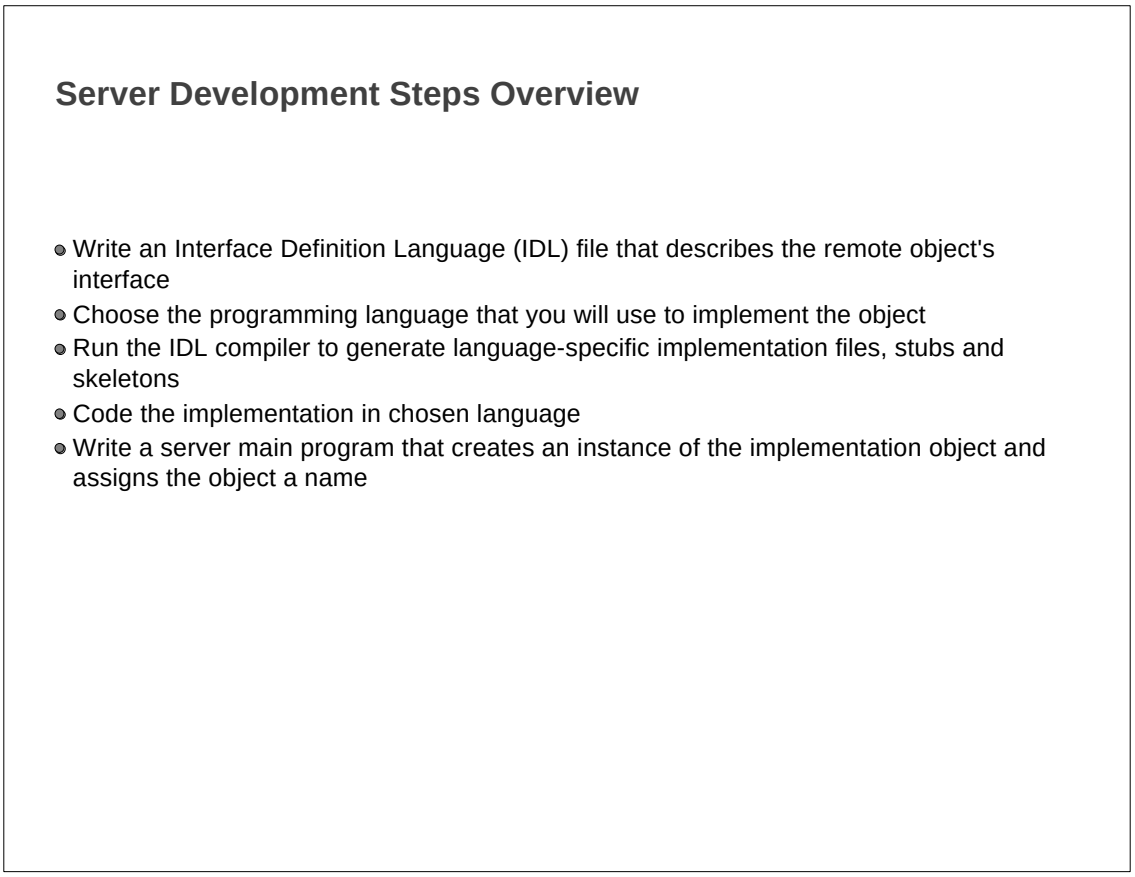
One of the advantages of CORBA is that it is an open industry standard, not a proprietary technology from a single company. And one of the features of CORBA is that ORBs that are networked together should be able to interoperate. In other words, client programs should be able to access objects spread across multiple ORBs, even if the ORBs are from different vendors.

To make this work, the OMG has defined a "wire format" for remote-method-call messages. There are a few implementations of the wire-format standard, but the best known is IIOP, which stands for Internet Interoperability Protocol. IIOP lets you connect ORBs together via the TCP/IP networking protocol and allow them to work together. Since the Internet uses TCP/IP, you can write clients that access remote CORBA objects across the Internet.



Now let's take a look at the CORBA runtime architecture. I think you'll see that it's remarkably similar to RMI: both use the notions of stubs and skeletons to insulate clients and remote objects from networking details. One difference is that CORBA uses IIOP to transmit method requests while RMI has it's own private protocol. However, Sun has promised that a future release of RMI will also support IIOP. That will make it easier for network administrators to configure firewalls and the like, especially for installations that are using both CORBA and RMI.

However, the development steps are quite a bit different for CORBA and RMI. We will next spend time looking at the differences.



Here are the steps for developing a CORBA object.

First, you must describe the object's interface using an OMG-defined language referred to as the Interface Definition Language (IDL). IDL is not a coding language - it simply lets you define the object's interface in a language-neutral fashion.

Next, you run an IDL compiler that converts the language-neutral interface definition into a particular programming language. In most CORBA environments, the IDL compiler also generates the stub and skeletons at this point, too.

Next you must write the remote object's implementation in your chosen language. Most IDL compilers generate a "starter" file that contains some of the code required to be a CORBA remote object.

Finally, you need to write a main program that creates an instance of the remote object. You can then optionally assign the new object a name in the CORBA Naming Service.

Now that you seen the big picture of CORBA server development, let's take a closer look at each step.

Defining the IDL for a Remotable Object

```
interface Meeting
{
    attribute string date;

    void scheduleIt();
};
```

IDL is an interface-definition language invented by the OMG. It's purpose is to let you describe a remote object's interface in a programming-language-neutral fashion. IDL syntax is similar to Java or C, however, it's not the same! We will not attempt to cover the entire IDL grammar in this course, but just focus in on the basics.

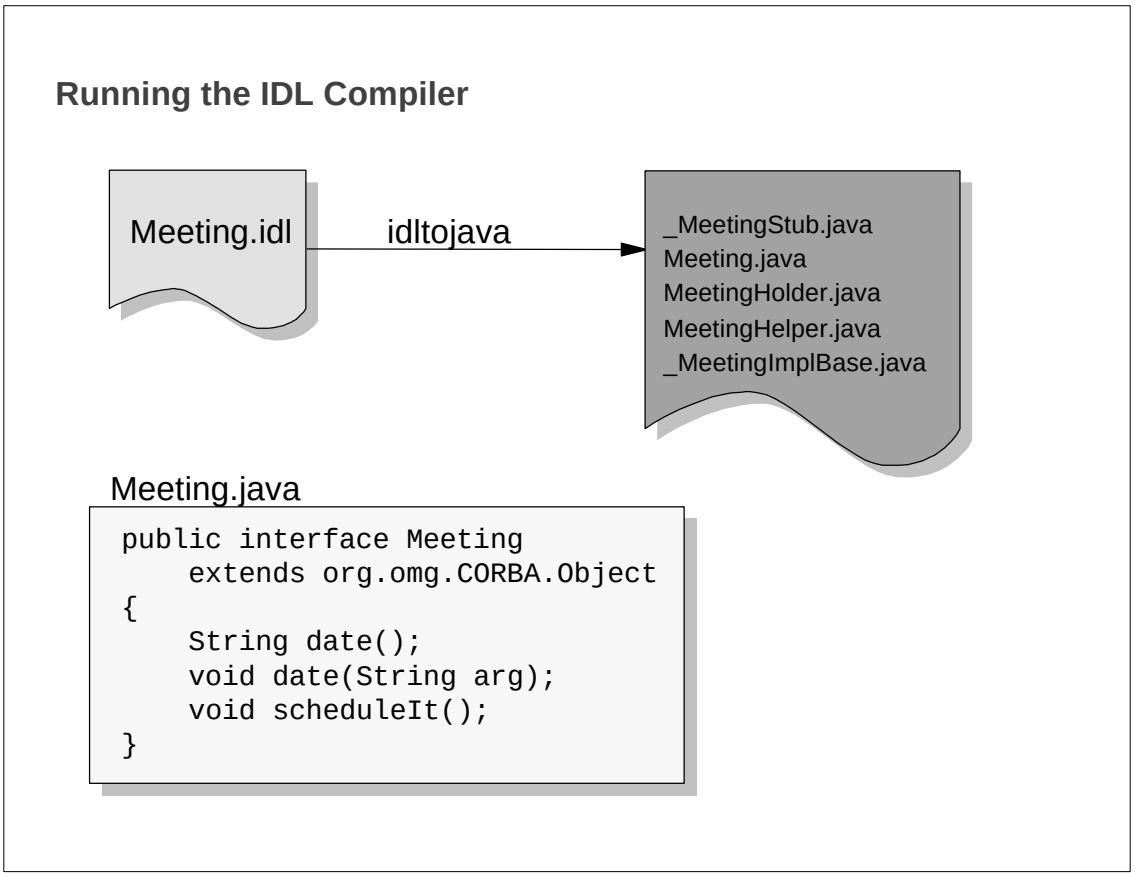
This page shows a simple interface definition in IDL. We use the keyword "interface" to start the definition, followed by an open brace. We must also supply a matching closing brace along with the terminating semicolon.

Inside of the interface, we typically define two sorts of things. The first thing shown here is a method named "scheduleIt" - this method accepts no arguments and returns nothing.

The second line in this interface defines a string attribute named "date". String is a CORBA data type - the IDL compiler must map each CORBA data type to an equivalent language type. In the case of Java, the IDL compiler maps CORBA strings to the Java language strings.

The keyword attribute seems to be a data definition, but it's not: instead, each attribute really corresponds to two methods: a method to retrieve an item of the given type, and a method to modify the item. These two methods are sometimes referred to as "getters" and "setters" or accessors and mutators. The actual implementation must provide the code for both of these methods for each attribute.

While we could spend much more time on IDL, we really can't in a short course like this one. You can find more information about IDL on the OMG web page, which is at www.omg.org.



The next step is to run the IDL compiler to generate language-specific source files that will implement the object, stubs and skeletons. Each ORB vendor supplies an IDL compiler; and each compiler works a little bit differently. Don't let that worry you, though: as long as there is IDL and IIOP, the objects created by different ORB development environments will be compatible with each other.

Anyway, this page shows running the IDL compiler that's part of JDK 1.2. It generates the following files: _MeetingStub.java, which implements the stub, Meeting.java which defines the remote interface as a Java interface, MeetingHolder.java and MeetingHelper.java, which implement some extra classes that help clients use the remote object and _MeetingImplBase.java. This last file has two jobs: first, it implements the skeleton for the remote object, and second, it provides a superclass which we can extend to write the remote object's implementation.

You don't really need to understand the code inmost of these source files; the only one that's really worth reading is Meeting.java, shown on this page, which defines a Java interface. Note that this Java interface extends a CORBA-defined interface, and includes three methods:a "getter" and a "setter" for the "date" attribute, and the "scheduleIt" method. Our next job will be to implement these methods.

One other note: as of this writing, the idltojava compiler is not included with the JDK. Instead, you must download it separately from the www.javasoft.com web site. See the JDK documentation for more details.

Writing the Remote Object Implementation

```
public class MeetingServant extends _MeetingImplBase
{
    private String ivDate = new String ( "1/1/2000" );

    public String date()
    {
        return ivDate;
    }

    public void date ( String arg )
    {
        ivDate = arg;
    }

    public void scheduleIt()
    {
      // do some scheduling stuff
    }
}
```

Now we must write the class for the remote object. By convention, we name such classes so that they have the word "Servant" in their names, here MeetingServant. Note that this class is NOT generated by the IDL compiler - you must write it from scratch. An aside: Other ORB development environments work differently than the JDK 1.2 ORB. Some IDL compilers actually generate part of the implementation class for you, requiring you more or less to fill in the blanks. Not so with JDK 1.2 though; it requires you to write the entire implementation class.

Here, we extend _MeetingImplBase, which is the stub, and implement the methods from the Meeting interface, which was shown on the last page. To implement the attribute methods, we define a private Java string variable, ivDate, and retrieve or modify it during the "getter" and "setter" methods. We didn't really write much of an implementation for the "scheduleIt" method - we'll leave that as an exercise for the student!

Writing an Object Server Overview

- Initialize the ORB
- Create an instance of the remote object
- Connect the remote object to the ORB
- Assign the remote object a name in the CORBA Name Service

Just like we did with RMI, we need to write a main program that acts as the object server. Here we overview the steps involved. First, we must initialize the server-side object request broker runtime; then we can create an instance of the remote object class we saw on the last page. Next, we must inform the ORB about the new object so it can be called remotely; we refer to this process as connecting the object to the ORB. Finally, we can assign the new object a name so that clients can find it - again this technique is quite similar to what we saw with RMI.

Let's now look at these steps in detail.

Object Server Initialization

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class MeetingServer
{
    public static void main ( String[] args )
    {
        try
        {
            ORB orb = ORB.init ( args, null );

            MeetingServant h = new MeetingServant();
            orb.connect ( h );

. . .

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

Here we show part of the main program for the object server. Note the import statements that let us reference the CORBA classes defined as a part of the JDK.

Our first job, initializing the ORB, is easy - we just need to call the static "init" method defined on the ORB class. This method accepts two arguments: any command line arguments that were passed to the server program itself, and a Java properties object which you can use to configure some ORB characteristics. We set the second argument to null to indicate that we didn't need to configure these properties. The command-line arguments also let you configure ORB properties, such as the port on which the ORB listens for method requests. We really won't use them, either - we coded like this so it will work if we do decide to pass arguments on the command line.

Next, this code creates an instance of the MeetingServant remote object. We must next connect to object to the ORB - that makes the object callable from outside of the server virtual machine.

Naming an Object

```
public class MeetingServer
{
    public static void main ( String[] args )
    {

  . . .

        org.omg.CORBA.Object o =
             orb.resolve_initial_references ( "NameService" );

        NamingContext ns = NamingContextHelper.narrow ( o );

        NameComponent nc = new NameComponent ( "Meeting", "" );
        NameComponent path[] = {nc};

        ns.rebind ( path, h );

  . . .
```

CORBA defines an elaborate and powerful naming service, which you can use to make it easy for clients to find objects. The CORBA name service has much more function than the RMI naming registry. For example, in CORBA, you can organize your names into "contexts", which are similar in concept to directories on a file system. The trade-off is that the CORBA name service methods are more cumbersome than we saw with the RMI registry.

The code shown here first accesses the naming service itself by calling resolve_initial_references. This method returns a reference to the so-called "root context", which is similar to the root directory on a hard disk. We then call a special method named "narrow" to cast the returned reference to the correct type. This notion of narrowing is a CORBA programming idiom - it works much like a Java cast. For any given interface, the "narrow" method is defined on the "helper" class generated by the IDL compiler.

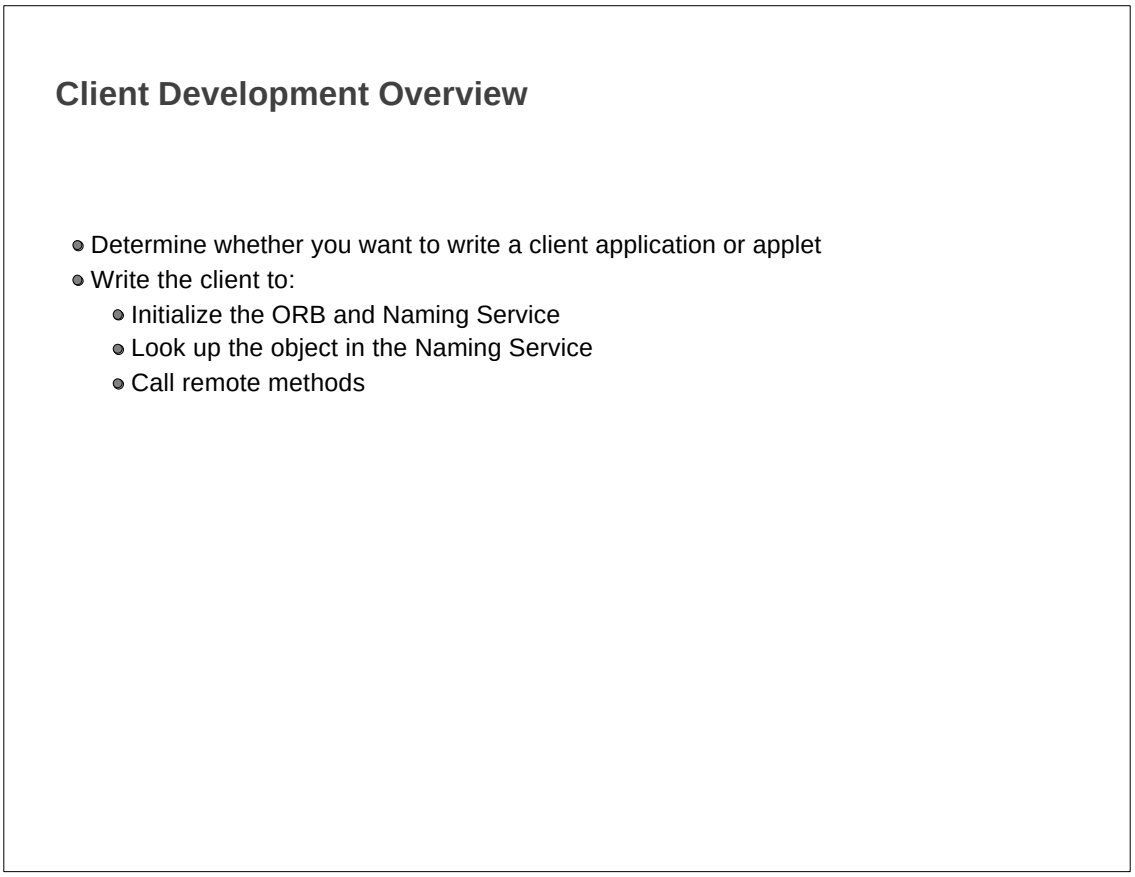
Once we have a correctly typed reference to the root context, we can then assign a name to the MeetingServant object that we showed creating on the last page. To do this, we need to define an array that contains the full name of the object, starting at the root. Since we will assign this name directly in the root context, the array contains a single NameComponent element. I think you can see that this is much more involved than the simple RMI registry!

Once we have built the name into the array, we can call the "rebind" method on the root naming context, passing the naming array and the MeetingServant object reference. Rebind stores the name and object reference in the name service, silently overwriting it if the name already exists.

Waiting for Method Requests

```
public class MeetingServer
{
    public static void main ( String[] args )
    {

  . . .

        java.lang.Object sync = new java.lang.Object();
        synchronized ( sync )
        {
            sync.wait();
        }

  . . .

}
```

Once it has finished initialization, the object server must then wait patiently for method requests, which the ORB automatically dispatches to the servant objects. Here we show executing a simple wait-loop that keeps the server alive, while allowing method requests to be dispatched.



Writing a CORBA client is quite similar to writing an RMI client. You first need to decide whether to write a standalone application or a client. At present, it's a bit more straightforward to write standalone applications, since browser support for CORBA is somewhat immature.

Whether you write an applet or application, the client needs to first initialize the client-side ORB and then create a stub for the remote object using the CORBA Name Service. The client can then call remote methods.

We will start by discussing how to write standalone Java clients, followed by a quick discussion on applets.

Writing a Client Application

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class MeetingClient
{
    public static void main ( String[] args )
    {
        try
        {
          // 1. Initialize the ORB
          // 2. Resolve to the name service
          // 3. Look up the object in the name service
          // 4. Call remote methods
        }
        catch ( Exception e )
        {
        }
    }
}
```

This page overviews the structure of a client standalone application. Note the import statements and the fact that we enclose the main code in a try-catch block for error processing. A more sophisticated error-handling scheme would be to issue each CORBA and remote method call in a separate try-catch block so that you can code more specific error-response code.

Initializing the ORB

```
public static void main ( String[] args )
{
    . . .

    ORB orb = ORB.init ( args, null );

    org.omg.CORBA.Object o =
        orb.resolve_initial_references ( "NameService" );

    NamingContext ns = NamingContextHelper.narrow ( o );

    . . .
}
```

The client code to initialize the ORB and the name service is pretty much identical to what we saw in the object server. We first call the static ORB.init method, which initializes the client-side ORB and returns a its reference. Then we can retrieve a reference to the naming service by calling "resolve_initial_references". We must then narrow, or cast the name service reference to its correct type.

Next we will see how to look up a remote object's reference from the name service.

Looking up an Object

```
public static void main ( String[] args )
{
    . . .

    NameComponent nc = new NameComponent ( "Meeting", "" );
    NameComponent path[] = {nc};

    o = ns.resolve ( path );
    Meeting m = MeetingHelper.narrow ( o );

    . . .

}
```

The name service reference we retrieved on the last page actually refers to the root naming context in the naming service. Recall that the CORBA naming service lets you create a hierarchy of naming contexts that are similar to directories on a hard disk; the root naming context is thus similar to the root directory on a disk.

You should also recall that in the object server, we registered a remote Meeting object's reference in the root context using the name "Meeting". So here, in the client, we retrieve the object reference given that name.

To compose the name, we create an array, with an element for each level that the name descends into the naming hierarchy. Since we registered the name in the root context, the array needs only a single element.
Once we have initialized the array, we then retrieve the object reference by calling the "resolve" method, passing the naming array.

After retrieving the reference, we follow normal CORBA programming practice of narrowing, or casting the reference to the correct type, "Meeting", in this case.

Calling Remote Methods

```
public static void main ( String[] args )
{
    . . .

    m.date ( "2/2/2000" );

    System.out.println ( "date string attribute: "
                              + m.date() );

    m.scheduleIt();

    . . .

}
```

Once we have a valid reference for the remote meeting object, we can call its methods. When we defined the Meeting interface, we specified an attribute named "Date", and a method named "scheduleIt". Here we call the "setter" and "getter" methods for the attribute and then call the scheduleIt method, which accepts no arguments and returns nothing.

Remember that each method call is actually dispatched remotely - the reference we hold in the variable "m" actually refers to a stub object on the client. When we call the stub, it forwards the method request, via the ORB, to the remote Meeting object. When the method returns, the process reverses itself: the server-side ORB notifies the client-side ORB of any results of the method, causing the stub to return back to the caller.

Writing a CORBA Client Applet

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class MeetingClientApplet extends java.applet.Applet
{
    public void init ()
    {
        try
        {
          // 1. Initialize ORB and name service
          // 2. Look up remote object in the name service
          // 3. Call remote methods (can also call from
          //    other methods if you save the reference)
        }
        catch ( Exception e )
        {
        }
    }
}
```

Now let's look at the differences when writing an CORBA client as an applet rather than as a standalone application. Here are the basic differences:

1. You typically code the CORBA initialization in the applet's "init" method rather than in "main"
2. The CORBA.init method is a bit different for applets - the first argument is the "this" pointer for the applet itself

Despite these changes, the CORBA-related code in an applet client is basically the same as in an application. You still use the CORBA name service to find remote references, narrowt the returned reference to the correct type, and so forth.

One other difference between applets and applications -- to use an applet from within a browser, you need to write an HTML page that references the applet. And it turns out that the HTML for CORBA applets is quite messy as we will see next.

Writing the HTML Page for the Applet

```
<HTML>
<title>Meeting Client Applet</title>
<center> <h1>Meeting Client</h1> </center>

The message from the MeetingServer is:
<p>

<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 500 HEIGHT = 120
codebase="http://java.sun.com/products/plugin/1.1.1/jinstall-111-win32.cab#Version=1,1,1,0">
<PARAM NAME = CODE VALUE = "MeetingClientApplet" >

<PARAM NAME="type" VALUE="application/x-java-applet;version=1.1">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.1"
java_CODE = "MeetingClientApplet" WIDTH = 500 HEIGHT = 120
pluginspage="http://java.sun.com/products/plugin/1.1.1/plugin-install.html">
<NOEMBED></COMMENT>

</NOEMBED></EMBED>
</OBJECT>

</HTML>
```

Look at this mess! The good news is that you don't have to write all of this (unless you really like that sort of thing). Instead, you can write a "normal" HTML page that references the CORBA client applet and then run a converter program, which you can download from the www.javasoft.com web site. Look for the HTML JRE 1.2 Plug-in converter.

Why all this complexity? As of this writing, no browser supports the JDK 1.2, which is required to use the Java IDL which we have covered. To solve this problem, Sun has created a browser plug-in application that substitutes a JDK 1.2 virtual machine for the browser's normal JVM. So, before running a CORBA Java applet, you need to download the plug-in from the javasoft web site and configure your browser to use it. Then you can run the converter mentioned previously to create the HTML such that the plug-in is activated.

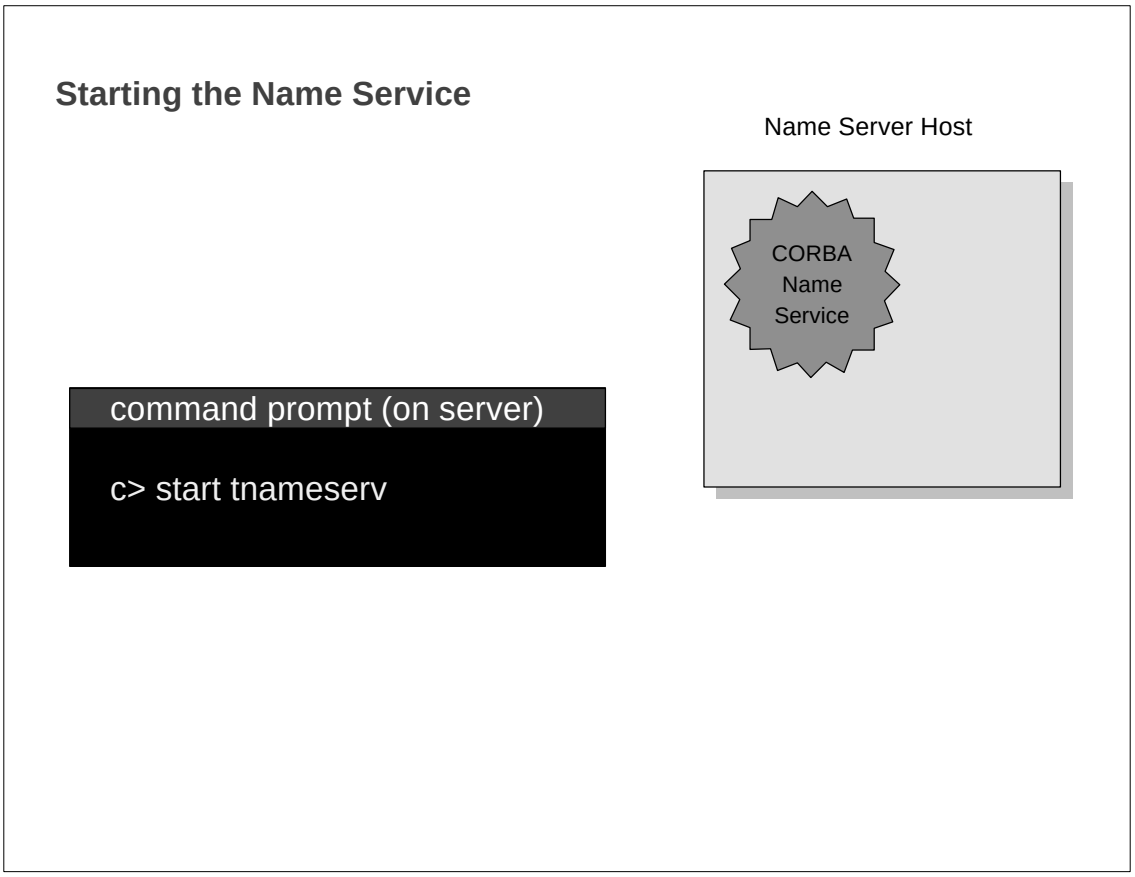
It sure was a lot easier in RMI! However, this complexity results from the fact that browsers typically lag the JDK. Once the browser vendors become compliant with JDK 1.2, writing CORBA client applets will be a lot easier.

Now on to deployment!

Deployment Overview

- Start the name service running
- Start the object server running
- Setup the web server (applets only)
- Run the client application or applet

This page shows the overall steps involved in setting up an CORBA system. We first need to start the name service, then start the main program that creates the object. If we are using an applet client, we also need to deploy and start a web server. Unlike in RMI, we don't need a web server if we are using standalone client applications. But in either case, we then need to run the client, either by bringing it up in a web browser in the case of an applet, or by running it from the command line.



The first step is to start the CORBA name service program, which typically runs on the same computer as the object server itself. The name server supplied with the JDK 1.2 is a transient name server - like the RMI registry, it does not save registered names in permanent storage. In other words, if you close the name server program, any registered names are lost.

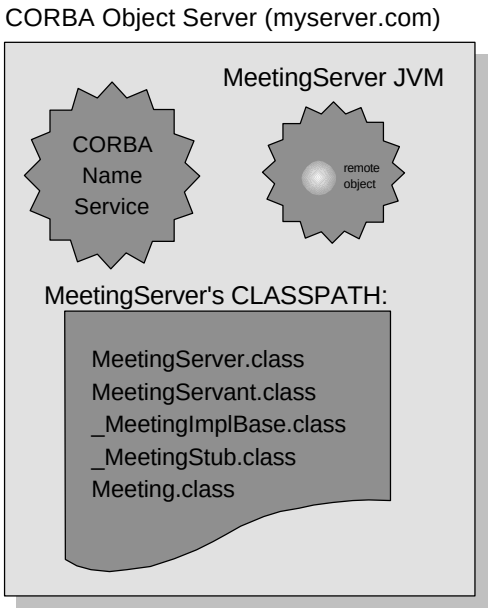
Unlike the RMI registry however, the JDK transient name server actually displays something in its window - the info it displays is not particularly useful, but it's interesting to look at!

As before, this diagram shows the Microsoft Windows command to start the name server as a background process; see your operating system's documentation for more details.

Starting the Object Server

command prompt (on server)

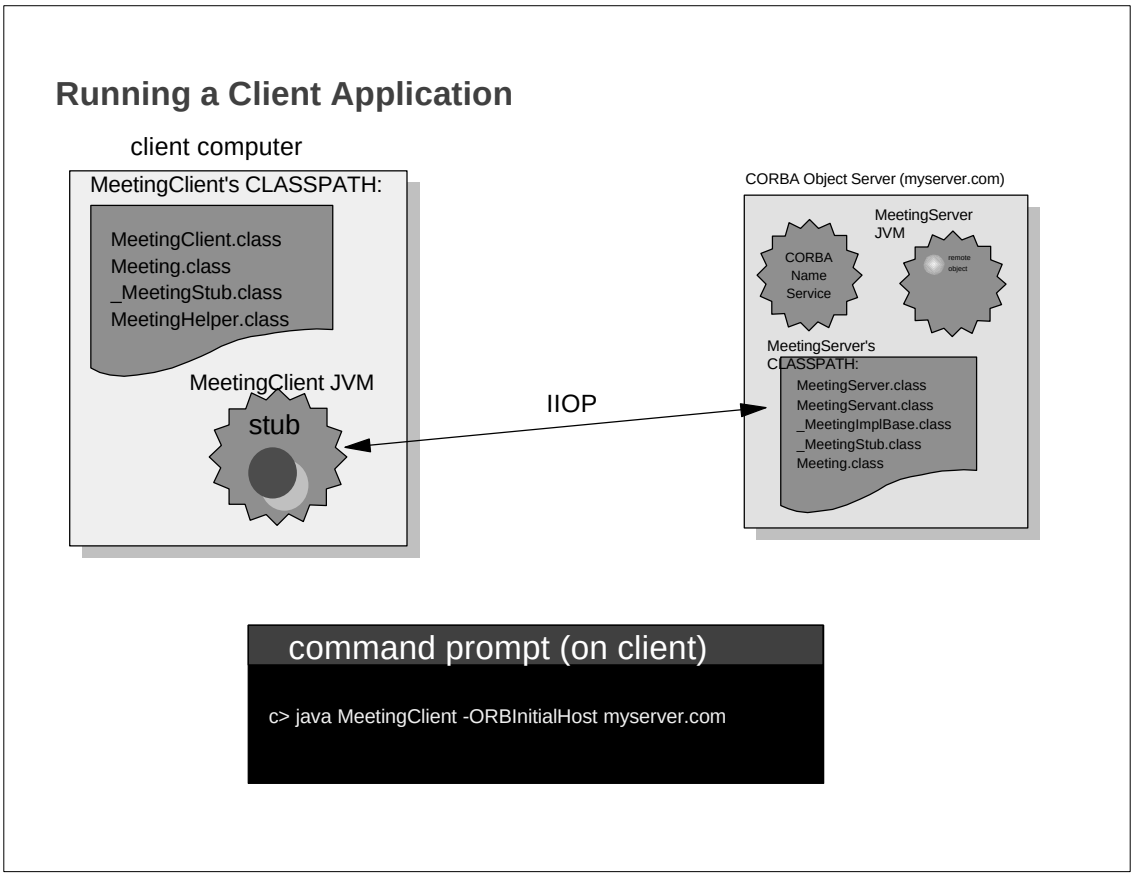
```
c> java MeetingServer
```



Once we have the name server running, we can start the object server. Remember that the object server creates a MeetingServant object and then registers it in the name service. The diagram shows the server virtual machine containing the MeetingServant object.

The MeetingServer program needs access to the class files shown in the diagram. The two most noteworthy are: MeetingServer, which contains the code for the main program itself and MeetingServant, which is the remote object itself.

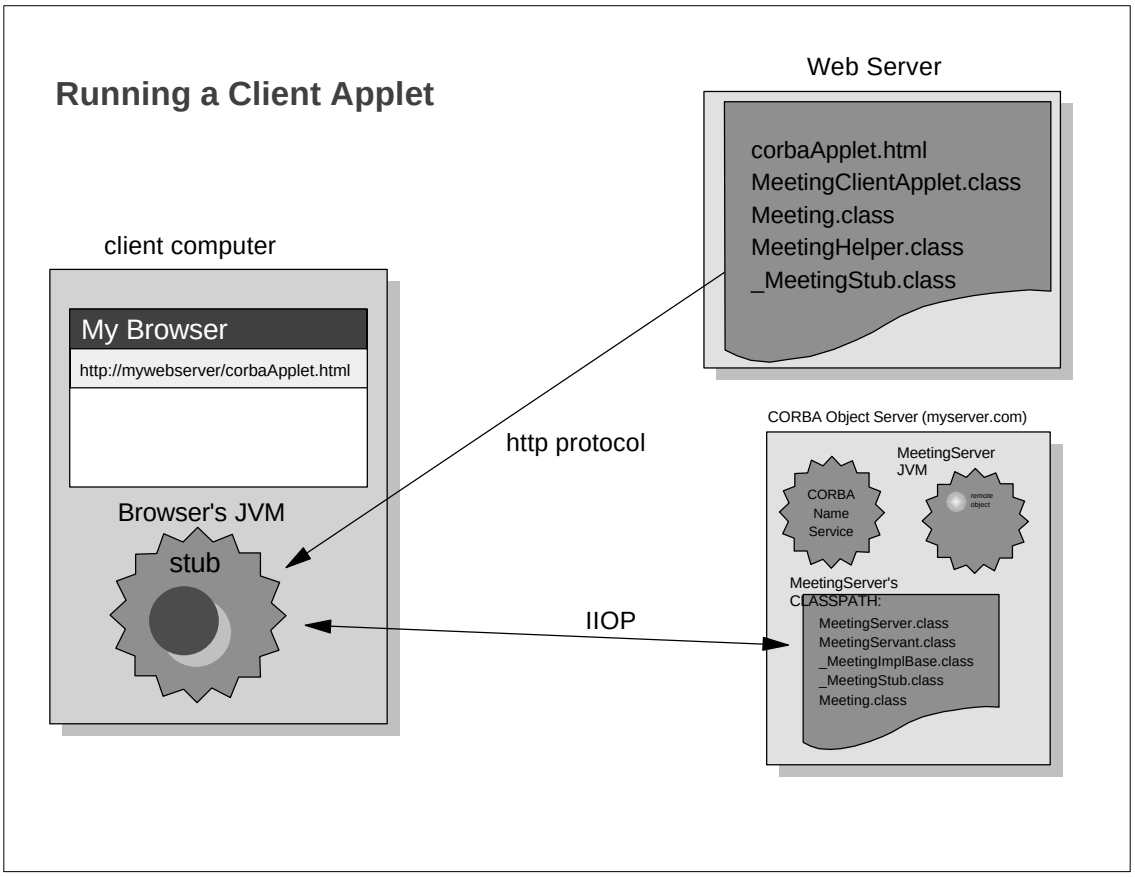
Now it's on to the client!



Let's first look at running a standalone client application -- we'll look at applets next. On the client computer, you need to install the files shown in the diagram, which include the client program itself, the remote interface and the stub and helper classes. To run the client, use the command line shown in the diagram. The ORBInitialHost flag indicates the host name where the name server is running, in this case, myserver.com.

Notice that this setup is a bit different than we saw with RMI - the CORBA infrastructure will not automatically download the stub class files, so we need to manually install them on the client. On the plus side, in CORBA, we don't have to worry about the security policy file!

Now let's look at a client applet.



In some ways, CORBA client applets are easier to deploy than standalone applications. That's mostly because the Java applet infrastructure itself knows how to download class files, so we don't have to pre-install our code on the client Instead, the infrastructure will download them from the web server.

The flip side is that since browsers are not up-to-date with the JDK 1.2, so we need to install the 1.2 browser plug-in on the client as discussed earlier. We also need to use the HTML converter program to generate the HTML that we install on the web browser.

We need to repeat the message we gave when we discussed applets in RMI with regards to digital signing: if you don't sign your applet, you must put the web server and the object server on the same computer to avoid security violations.

This concludes our discussion of CORBA and Java!

Summary

Well, I hope that you have found this introduction to distributed object with Java to be a good use of your time. In short order, we discussed distributed object theory, and then took a closer look at the programming models for Java RMI and CORBA.

And like the folks at Twisted Transistor, Inc., you should now be ready to start work on using these techniques to improve your business.