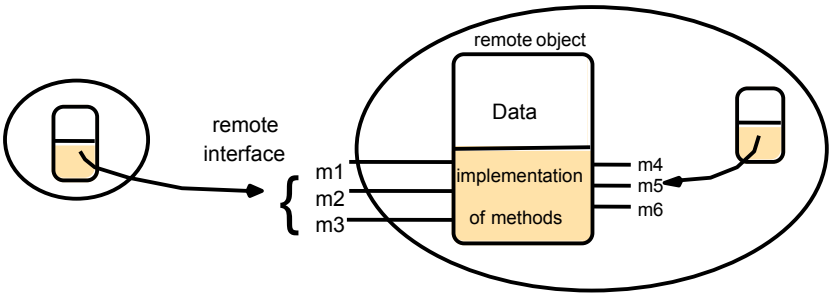
Java RMI 101

CS 475

Java RMI

- Features
  - Integrated with Java language + libraries
    - Security, write once run anywhere, multithreaded
    - Object orientation
  - Can pass "behavior"
    - Mobile code
    - Not possible in CORBA, traditional RPC systems
  - Distributed Garbage Collection
  - *Remoteness of objects **intentionally** not transparent*

A remote object and its remote interface



Remote Interfaces, Objects, and Methods

- Objects become remote by implementing a remote interface
  - A remote interface extends the interface java.rmi.Remote
  - Each method of the interface declares java.rmi.RemoteException in its `throws` clause in addition to any application-specific clauses

Creating distributed applications using RMI

1. Define the remote interfaces
2. Implement the remote objects and server
3. Implement the client
4. Compile the remote interface, server, and client (*javac*)
5. Generate the stub and skeleton using *rmic*
   - *Not necessary in Java 5 (and later)*
6. Start the RMI registry
7. Start the server
8. Run the client

An Example: Echo service

- We will build a remote server that echoes any text sent to it by a client after converting the text to uppercase
  - We've seen this example before
- We will need to create the following files:
  - Echo.java (interface)
  - EchoServer.java (server program)
  - EchoImpl.java (implementation of remote object providing echo service)
  - EchoClient.java (client program)

The Interface code

```
import java.rmi.*;
public interface Echo extends Remote {
    String EchoMessage(String strMsg)
   throws RemoteException;
}
```


The Remote Object

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

public class EchoImpl extends UnicastRemoteObject implements
Echo {
   public EchoImpl() throws RemoteException { super(); };

   public String EchoMessage(String Msg) throws RemoteException {
     String capitalizedMsg;

       System.out.println("Server: EchoMessage() invoked...");
       System.out.println("Server: Message > " + Msg);
     capitalizedMsg = Msg.toUpperCase();
       return(capitalizedMsg);
   }
}
```

Remote Object explained

- The class EchoImpl implements the remote interface Echo and provides a remote object
- It extends another class known as UnicastRemoteObject which implements a remote access protocol
- All the methods for EchoImpl must throw a remote exception

EchoServer code

```
import java.rmi.*;

public class EchoServer{
   public static void main(String argv[]) {
      try {
            System.setSecurityManager(new RMISecurityManager());

       System.out.println("Server: Registering Echo Service");
       EchoImpl remote = new EchoImpl();
       Naming.rebind("EchoService", remote);
       System.out.println("Server: Ready...");
      }
      catch (Exception e) {
     System.out.println("Server: Failed to register Echo Service: " + e);
       }
   }
}
```

EchoServer explained

- Installs a new security manager for the RMI service
- Creates an object of class EchoImpl (the remote object)
- Registers the object called "EchoService" with the RMI Naming Service

The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)
This method is used by a server to register the identifier of a remote object by name.

void bind (String name, Remote obj)
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)
This method removes a binding.

Remote lookup(String name)
This method is used by clients to look up a remote object by name. A remote object reference is returned.

String [] list()
This method returns an array of Strings containing the names bound in the registry.

EchoClient code

```
import java.rmi.*;
import java.rmi.server.*;
public class EchoClient
{
   public static void main(String argv[]) {
    //code for processing command line argument
          String strMsg = argv[0];

     System.setSecurityManager(new RMISecurityManager());

 // Get a remote reference to the RMIExampleImpl class
     String strName = "rmi://localhost/EchoService";
     System.out.println("Client: Looking up " + strName + "...");
    Echo RemEcho = null;
    try {
       RemEcho = (Echo)Naming.lookup(strName);
    } catch (Exception e) {
       System.out.println("Client: Exception thrown looking up " + strName);
       System.exit(1);
    }
```


EchoClient cont'd

```
 // Send a messge to the remote object

     try {
        String modifiedMsg = RemEcho.EchoMessage(strMsg);

        System.out.println("From Server: "+ modifiedMsg);
     }
     catch (Exception e) {

        System.out.println("Client: Exception thrown calling EchoMessage().");
        System.exit(1);
     }
   }

 }
```

EchoClient explained

- Create and install the security manager
- Use the Naming.lookup method to obtain a reference to the remote object
- Invoke the remote method on the remote object

RMI with Java 5

- J2SE 5.0 (and later) support *dynamic generation of stub classes at runtime, that is,* no need to use ***rmic***
- Compile the interface, Server, and Client
- Start the rmiregistry
  - > rmiregistry &
- Start the server
  - > java EchoServer
- Start the client
  - > java EchoClient "This is a test"

Advanced Techniques

- Security Manager
- Parameter Passing
- Passing behavior
  - See Java RMI tutorial track example
- Callbacks
- Activation

Parameter Passing

- Arguments to and return values from remote methods can be of any type including local objects, remote objects or primitive data types
  - Local objects must be serializable, i.e. must implement the interface java.io.serializable
    - All primitive objects and most Java core classes are serializable
    - Examples of objects that are not serializable: threads, file descriptors, etc., i.e. objects that encapsulate information that only makes sense within a single address space
- Remote objects are passed by reference
- Local objects are passed by value, using serialization

Security Manager

- The Java security model requires code to be granted specific permissions to be allowed to perform certain operations
- In Java 1.2 (and later), if you install a security manager, you need to specify a policy file (typically as a command line argument)
- For example:

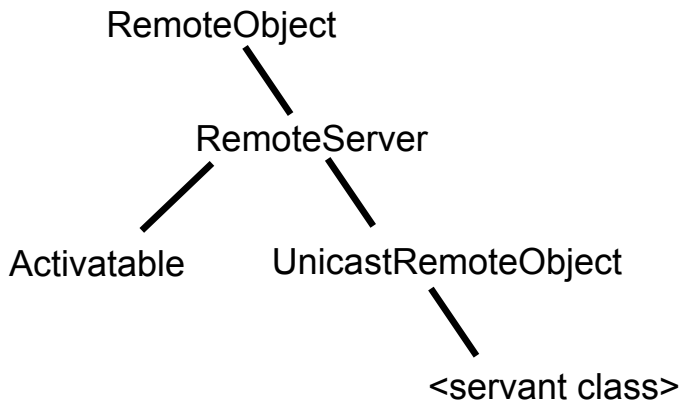
  java –Djava.security.policy=filename EchoServer

Sample Policy

- The following policy allows downloaded code, from any code base, to do two things:
  - Connect to or accept connections on unprivileged ports (ports greater than 1024) on any host
  - Connect to port 80 (the port for HTTP)

```
grant {
permission java.net.SocketPermission "*:1024-65535",
   "connect,accept";
Permission java.net.SocketPermission "*:80", "connect";
 };
```

Classes supporting Java RMI



Readings

- Coulouris – Chapter 5 or Liu -- Chapters 7, 8
- WWW (see links on class web page)
  - Java RMI tutorial on web