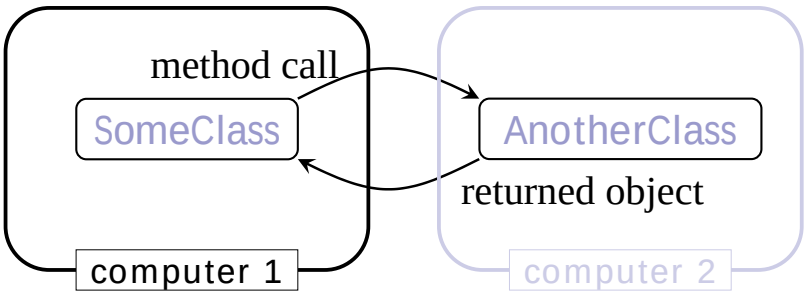
# Remote Method Invocation

## "The network is the computer"

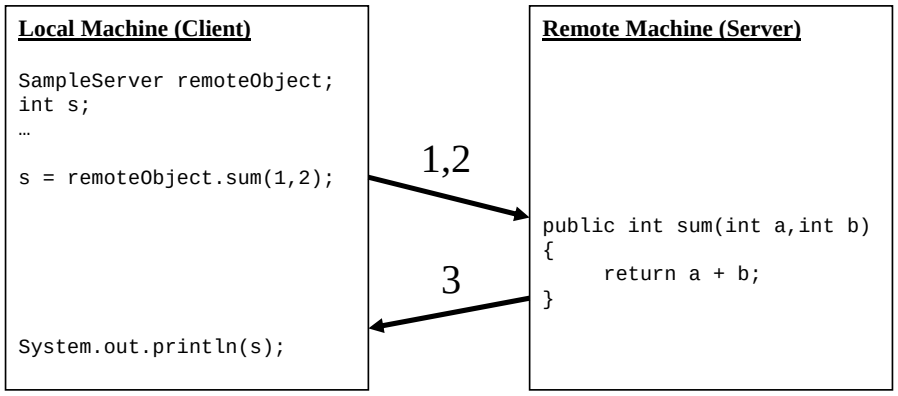
- Consider the following program organization:




- If the network *is* the computer, we ought to be able to put the two classes on different computers
  - RMI is one technology that makes this possible

## Java Remote Object Invocation (RMI)

- RMI allows programmers to execute remote function class using the same semantics as local functions calls.




# What is needed for RMI

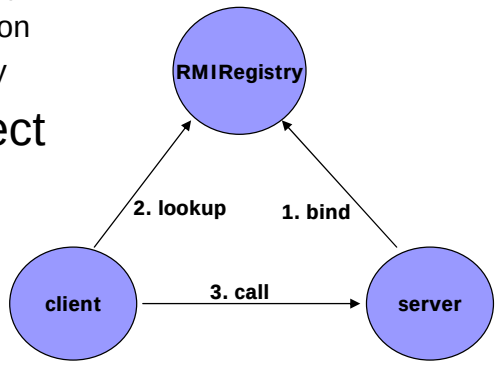
- Java makes RMI (Remote Method Invocation) *fairly* easy, but there are some extra steps
- To send a message to a remote "server object,"
  - The "client object" has to *find* the object
    - Do this by looking it up in a registry
  - The client object then has to marshal the parameters (prepare them for transmission)
    - Java requires Serializable parameters
    - The server object has to unmarshal its parameters, do its computation, and marshal its response
  - The client object has to unmarshal the response
- Much of this is done for you by special software

# Terminology

- A remote object is an object on another computer
- The client object is the object making the request (sending a message to the other object)
- The server object is the object receiving the request
- As usual, “client” and “server” can easily trade roles (each can make requests of the other)
- The rmiregistry is a special server that looks up objects by name
  - Hopefully, the name is unique!
- rmic is a special compiler for creating stub (client) and skeleton (server) classes

# RMI Components

- RMI registry
  - Each remote object needs to register their location
  - RMI clients find remote objects via the lookup service
- Server hosting a remote object
  - Construct an implementation of the object
  - Provide access to methods via skeleton
  - Register the object to the RMI registry
- Client using a remote object
  - Ask registry for location of the object
  - Construct stub
  - Call methods via the object’s stub



# Interfaces

- Interfaces define behavior
- Classes define implementation

- Therefore,
  - In order to use a remote object, the client must know its behavior (interface), but does not need to know its implementation (class)
  - In order to provide an object, the server must know both its interface (behavior) and its class (implementation)
- In short,
  - The interface must be available to both client and server
  - The class should only be on the server

# Classes

- A Remote class is one whose instances can be accessed remotely
  - On the computer where it is defined, instances of this class can be accessed just like any other object
  - On other computers, the remote object can be accessed via object handles
- A Serializable class is one whose instances can be marshaled (turned into a linear sequence of bits)
  - Serializable objects can be transmitted from one computer to another

# Conditions for serializability

- If an object is to be serialized:
  - The class must be declared as public
  - The class must implement Serializable
  - The class must have a no-argument constructor
  - All fields of the class must be serializable: either primitive types or serializable objects

# Remote interfaces and class

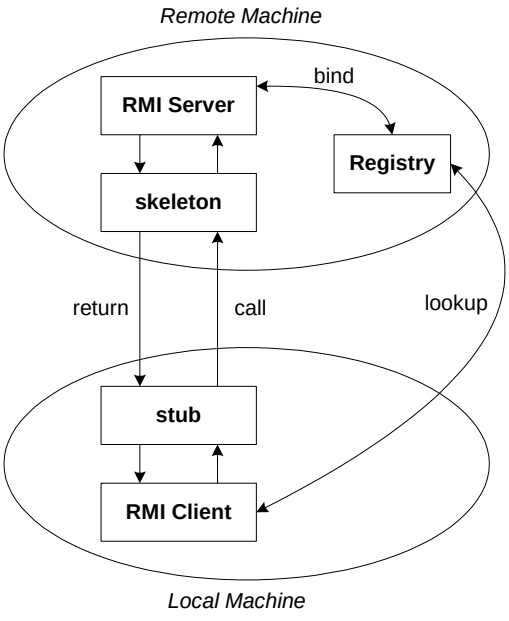
- A Remote class has two parts:
  - The interface (used by both client and server):
    - Must be public
    - Must extend the interface java.rmi.Remote
    - Every method in the interface must declare that it throws java.rmi.RemoteException (other exceptions may also be thrown)
  - The class itself (used only by the server):
    - Must implement a Remote interface
    - Should extend java.rmi.server.UnicastRemoteObject
    - May have locally accessible methods that are not in its Remote interface

# Remote vs. Serializable

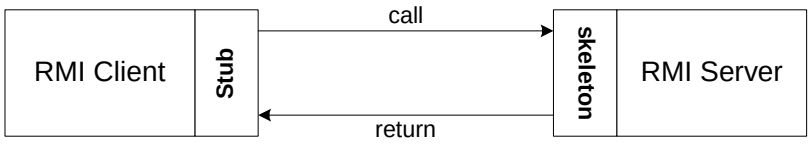
- A Remote object lives on another computer (such as the Server)
  - You can send messages to a Remote object and get responses back from the object
  - All you need to know about the Remote object is its interface
  - Remote objects don't pose much of a security issue
- You can transmit a *copy* of a Serializable object between computers
  - The receiving object needs to know how the object is implemented; it needs the class as well as the interface
  - There is a way to transmit the class definition
  - Accepting classes *does* pose a security issue

# RMI Architecture

- The server must first bind its name to the registry
- The client lookup the server name in the registry to establish remote references.
- The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.



# The Stub and Skeleton




- Each remote object has two interfaces
  - Client interface – a stub/proxy of the object
  - Server interface – a skeleton of the object
- When a client invokes a remote method, the call is first forwarded to stub.
- The stub is responsible for sending the remote call over to the server-side skeleton
- The stub opening a socket to the remote server, marshaling the object parameters and forwarding the data stream to the skeleton.
- A skeleton contains a method that receives the remote calls, unmarshals the parameters, and invokes the actual remote object implementation.

# Steps of Using RMI

1. Create Service Interface
2. Implement Service Interface
3. Create Stub and Skeleton Classes
4. Create RMI Server
5. Create RMI Client

# 1. Defining RMI Service Interface

- Declare an Interface that extends java.rmi.Remote
  - Stub, skeleton, and implementation will implement this interface
  - Client will access methods declared in the interface
- Example

```
public interface RMILightBulb extends java.rmi.Remote {
  public void on ()     throws java.rmi.RemoteException;
  public void off()     throws java.rmi.RemoteException;
  public boolean isOn() throws java.rmi.RemoteException;
}
```

# 2. Implementing RMI Service Interface

- Provide concrete implementation for each methods defined in the interface
- The class that implement the service should extend UnicastRemoteObject
- Every remotely available method must throw a RemoteException (because connections can fail)

```
public class RMILightBulbImpl extends java.rmi.server.UnicastRemoteObject
    implements RMILightBulb
{
    public RMILightBulbImpl() throws java.rmi.RemoteException
    {setBulb(false);}
    private boolean lightOn;
    public void on() throws java.rmi.RemoteException { setBulb (true); }
    public void off() throws java.rmi.RemoteException {setBulb (false);}
    public boolean isOn() throws java.rmi.RemoteException
    { return getBulb(); }
    public void setBulb (boolean value) { lightOn = value; }
    public boolean getBulb () { return lightOn; }
}
```

## 3. Generating Stub & Skeleton Classes

- The class that implements the remote object should be compiled as usual
- Then, it should be compiled with rmic:
- Example:
  - `rmic RMILightBulbImpl`
  - creates the classes:
    - RMILightBulbImpl_Stub.class
      - Client stub
    - RMILightBulbImpl_Skeleton.class
      - Server skeleton
- These classes do the actual communication

## 4. Creating RMI Server

- Create an instance of the service implementation
- Register with the RMI registry (binding)

```
import java.rmi.*;
import java.rmi.server.*;
public class LightBulbServer {
 public static void main(String args[]) {
  try {
   RMILightBulbImpl bulbService = new RMILightBulbImpl();
   RemoteRef location = bulbService.getRef();
   System.out.println (location.remoteToString());
   String registry = "localhost";
   if (args.length >=1) {
    registry = args[0];
   }
   String registration = "rmi://" + registry + "/RMILightBulb";
   Naming.rebind( registration, bulbService );
 } catch (Exception e) { System.err.println ("Error - " + e); } } }
```

# 5. Creating RMI Client

- Obtain a reference to the remote interface
- Invoke desired methods on the reference

```
import java.rmi.*;
public class LightBulbClient {
 public static void main(String args[]) {
  try { String registry = "localhost";
   if (args.length >=1) { registry = args[0]; }
   String registration = "rmi://" + registry + "/RMILightBulb";
   Remote remoteService = Naming.lookup ( registration );
   RMILightBulb bulbService = (RMILightBulb) remoteService;
   bulbService.on();
   System.out.println ("Bulb state : " + bulbService.isOn()  );
   System.out.println ("Invoking bulbservice.off()");
   bulbService.off();
   System.out.println ("Bulb state : " + bulbService.isOn() );
  } catch (NotBoundException nbe) {
   System.out.println ("No light bulb service available in registry!");
  } catch (RemoteException re) { System.out.println ("RMI - " + re);
  } catch (Exception e) { System.out.println ("Error - " + e); }
 }
}
```

# Steps of Running RMI

- Make the classes available in the server host's, registry host's, and client host's classpath
  - Copy, if necessary
- Start the registry
  - **`rmiregistry`**
- Start the server
  - **`java LightBulbServer reg-hostname`**
- Start the client
  - **`java LightBulbClient reg-hostname`**

# References


- Java Network Programming and Distributed Computing, David Reilly & Michael Reilly, Addison Wesley.
- CS587x – Remote Method Invocation, Ying Cai, Department of Computer Science, Iowa State University.