

# RMI PROGRAMMING

September 2018

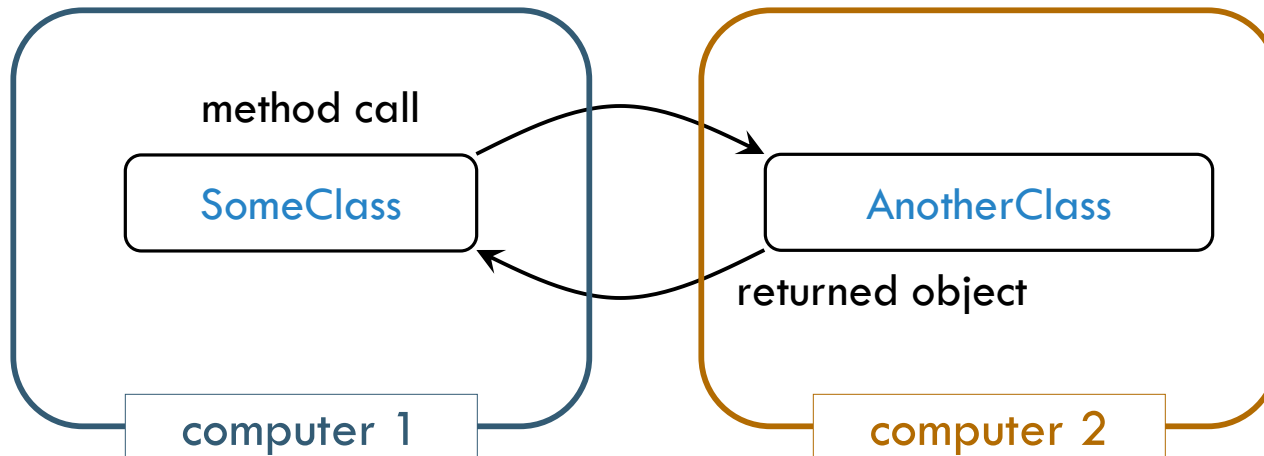
# WHAT IS RMI?

## Remote Method Invocation

- A **true distributed computing application interface** for Java, written to provide easy access to objects existing on remote virtual machines
- **Provide access to objects** existing on remote virtual machines

# “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

# WHAT IS RMI? CONT.

## Remote Method Invocation

- Remote objects can be treated similarly to local objects
- Handles marshalling, transportation, and garbage collection of the remote objects
- Became part of the JDK with version 1.1

# WHAT IS RMI NOT?

Not an all-purpose ORB architecture like CORBA and DCOM (but close ...)

Not language independent

- Limited only to Java and Perl
- Interfaces to remote objects are defined using ordinary Java **interfaces** (rather than having to use a special purpose interface definition language)
- Can provide more advanced features like serialization and security

# 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

# 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](#)

# WHAT IS NEEDED FOR RMI

- 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



# PROCESSES

For RMI, you need to be running *three* processes

- The Client
- The Server
- The **Object Registry**, **rmiregistry**, which is like a DNS service for objects

You also need TCP/IP active

# 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 of any transmitted object must be on both client and server
- The class whose method is being used 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**

# CLASSES

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

It is probably not a good idea for an object to be both remote and serializable

# CONDITIONS FOR SERIALIZABILITY

If an object is to be serialized:

- The class must be declared as **public**
- The class must implement **Serializable**
  - However, **Serializable** does not declare any methods
- The class must have a no-argument constructor
- All fields of the class must be **serializable**: either primitive types or Serializable objects
  - Exception: Fields marked **transient** will be ignored during serialization

# 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)

# REMOTE INTERFACES AND CLASS

- The class itself (used only by the server):
  - Must implement the `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**



# SECURITY

It isn't safe for the client to use somebody else's code on some random server

- `System.setSecurityManager(new RMISecurityManager());`
- The security policy of `RMISecurityManager` is the same as that of the default `SecurityManager`
- Your client program should use a more conservative security manager than the default

# SECURITY

Most discussions of RMI assume you should do this on both the client and the server

- Unless your server also acts as a client, it isn't really necessary on the server

See:

- <http://fragments.turtlemeat.com/rmi.php>
- <http://docs.oracle.com/javase/tutorial/rmi/running.html>

# THE SERVER CLASS

**The class that defines the server object should extend:**  
**UnicastRemoteObject**

- This makes a connection with exactly one other computer
- If you must extend some other class, you can use `exportObject()` instead
- Sun does *not* provide a **MulticastRemoteObject** class

**The server class needs to register its server object:**

- `String url = "rmi://" + host + ":" + port + "/" + objectName;`
  - The default port is 1099
- `Naming.rebind(url, object);`

Every remotely available method must throw a **RemoteException**  
(because connections can fail)

Every remotely available method should be **synchronized**

# HELLO WORLD SERVER: INTERFACE

```
import java.rmi.*;

public interface HelloInterface extends Remote
{
    public String say() throws RemoteException;
}
```

# HELLO WORLD SERVER: CLASS

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

public class Hello extends UnicastRemoteObject
    implements HelloInterface {
    private String message; // Strings are serializable

    public Hello (String msg) throws RemoteException {
        message = msg;
    }

    public String say() throws RemoteException {
        return message;
    }
}
```

# REGISTERING THE HELLO WORLD SERVER

```
class HelloServer {
    public static void main (String[] argv) {
        try {
            Naming.rebind("rmi://localhost/HelloServer",
                          new Hello("Hello, world!"));
            System.out.println("Hello Server is ready.");
        }
        catch (Exception e) {
            System.out.println("Hello Server failed: " + e);
        }
    }
}
```

# THE HELLO WORLD CLIENT PROGRAM

```
class HelloClient {
    public static void main (String[] args) {
        HelloInterface hello;
        String name = "rmi://localhost/HelloServer";
        try {
            hello =(HelloInterface) Naming.lookup(name);
            System.out.println(hello.say());
        }
        catch (Exception e) {
            System.out.println("HelloClient exception: " +
e);
        }
    }
}
```

# RMIC

The class that implements the remote object should be compiled as usual

Then, it should be compiled with `rmic`:

- `rmic Hello`

This will generate files `Hello_Stub.class` and `Hello_Skel.class`

These classes do the actual communication

- The “**Stub**” class must be *copied* to the client area
- The “**Skel**” was needed in SDK 1.1 but is no longer necessary



# TRYING RMI

In three different terminal windows:

1. Run the registry program:

- `rmiregistry`

2. Run the server program:

- `java HelloServer`

3. Run the client program:

- `java HelloClient`

- If all goes well, you should get the “Hello, World!” message

# SUMMARY

1. Start the registry server, `rmiregistry`
2. Start the object server
  1. The object server registers an object, with a name, with the registry server

# SUMMARY

## 1. Start the client

1. The client looks up the object in the registry server

## 2. The client makes a request

1. The request actually goes to the Stub class
2. The Stub classes on client and server talk to each other
3. The client's Stub class returns the result