Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

KATEDRA INŻYNIERII OPROGRAMOWANIA http://kio.wi.zut.edu.pl/

INŻYNIERIA OPROGRAMOWANIA

Zaawansowane Techniki Programowania Java

#04 : RMI (*java.rmi*)

Prowadzący:

Krzysztof Kraska

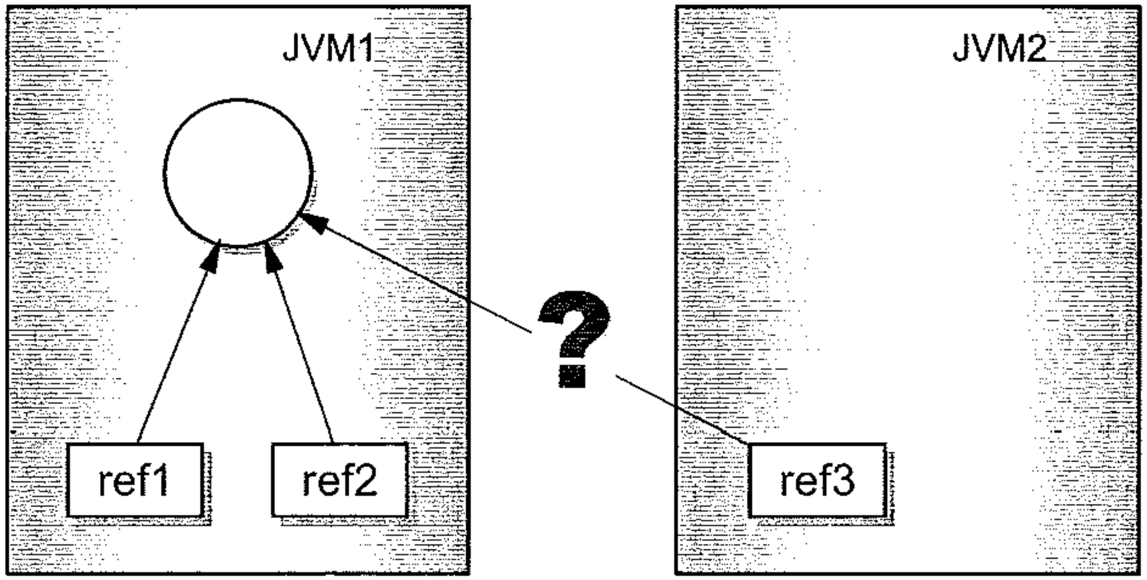
email: kkraska@wi.zut.edu.pl

Szczecin, 19 kwietnia 2018 r.

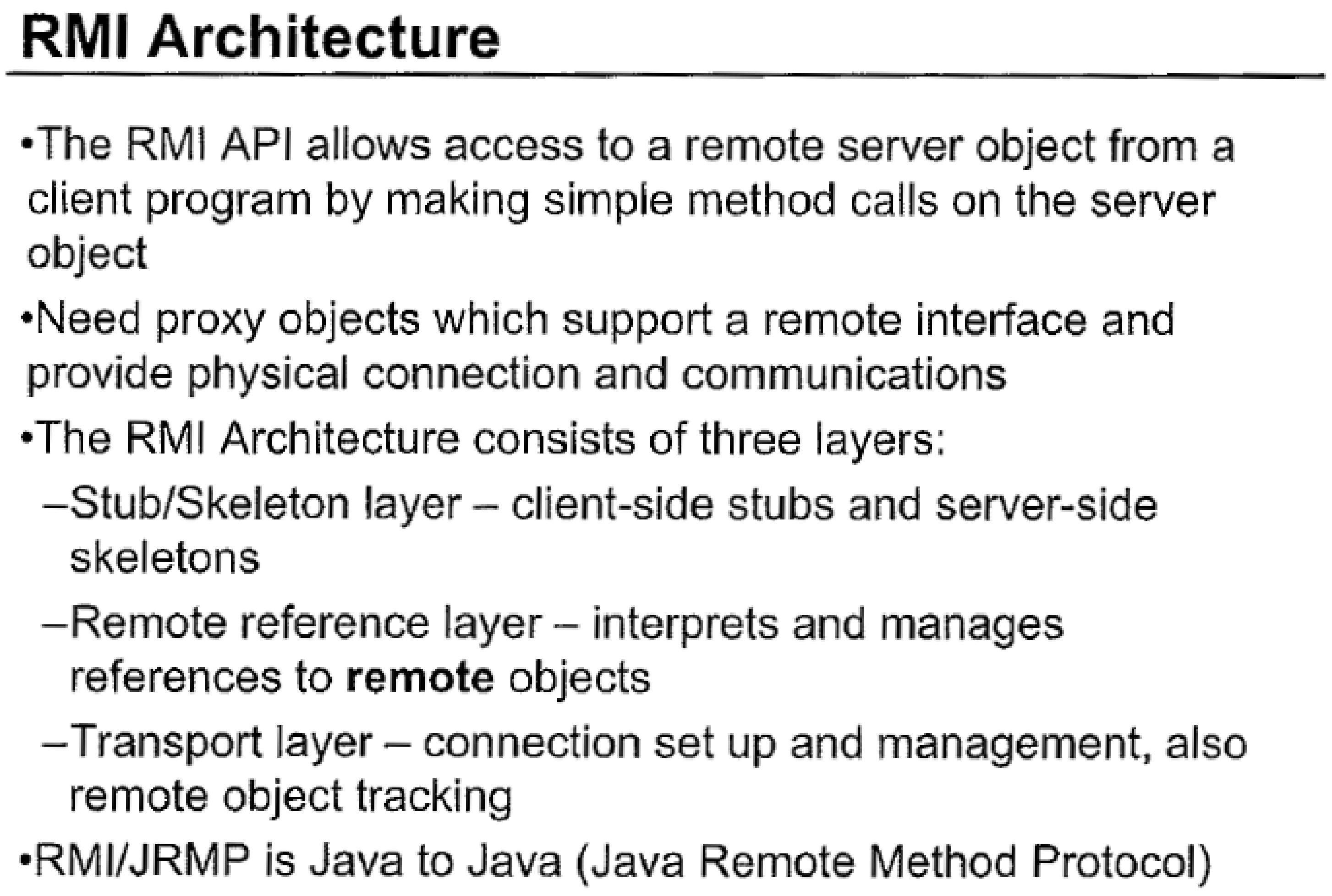
Remote Method Invocation

Paradigm Shift

Move from sharing resources to sharing objects.



RMI only works between Java applications.

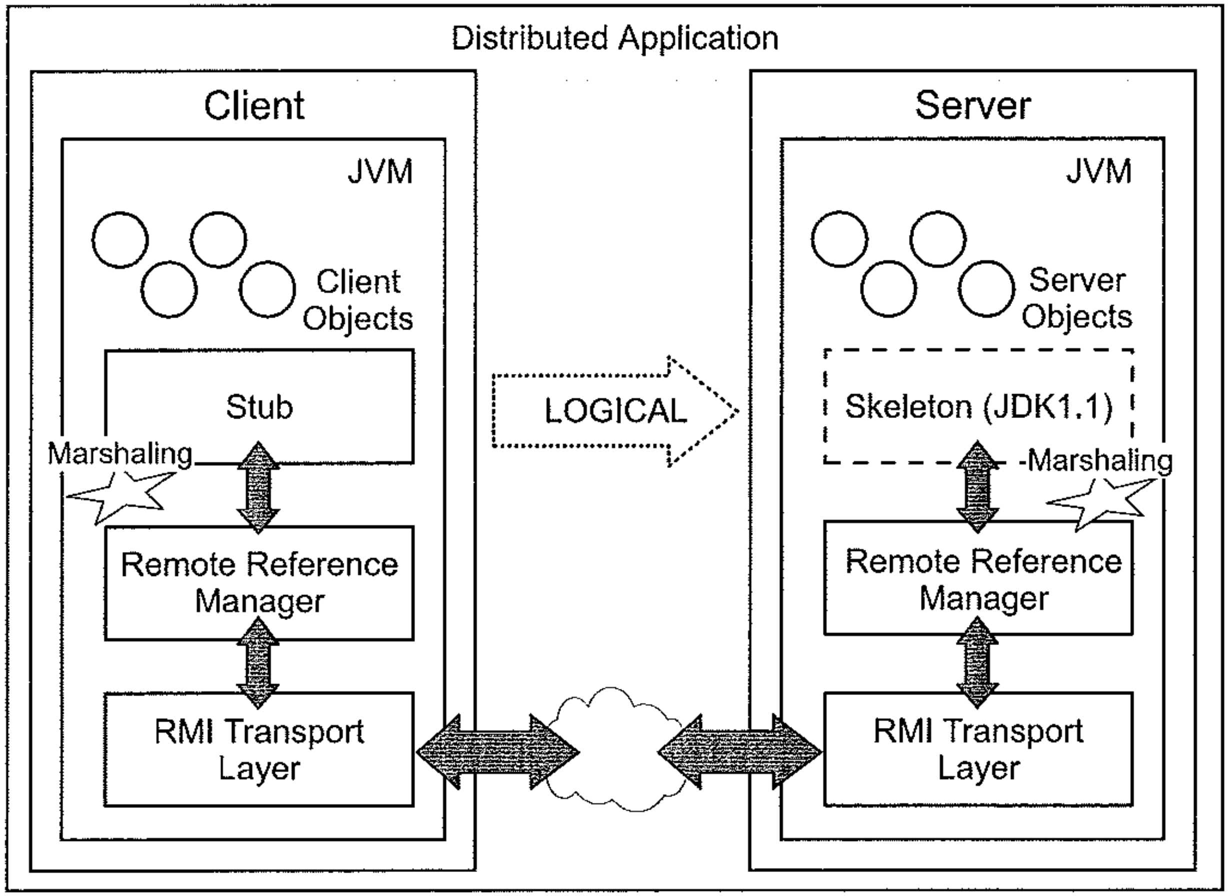


RMI Architecture

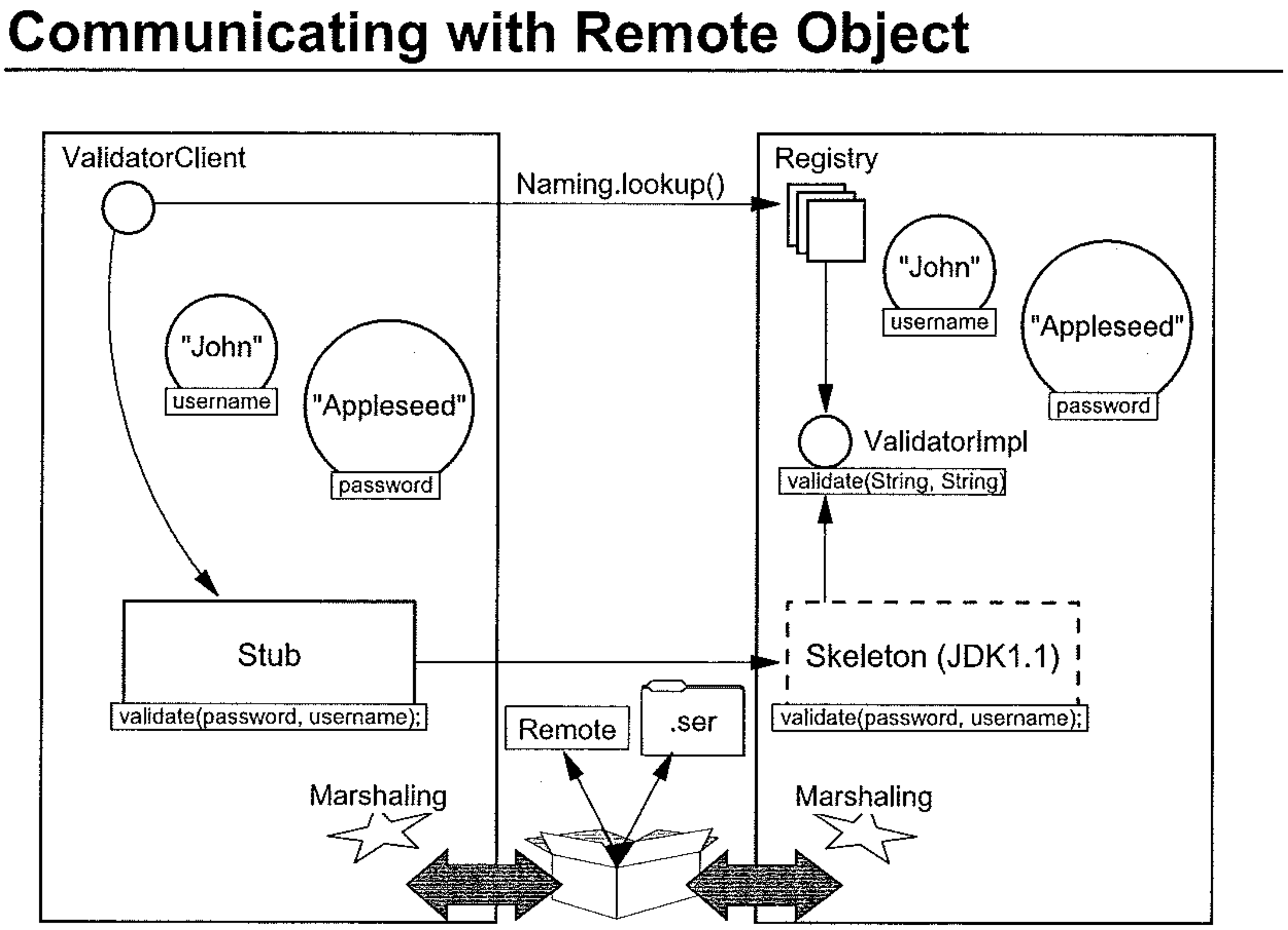
- The RMI API allows access to a remote server object from a client program by making simple method calls on the server object
- Need proxy objects which support a remote interface and provide physical connection and communications
- The RMI Architecture consists of three layers:
  - Stub/Skeleton layer – client-side stubs and server-side skeletons
  - Remote reference layer – interprets and manages references to **remote** objects
  - Transport layer – connection set up and management, also remote object tracking
- RMI/JRMP is Java to Java (Java Remote Method Protocol)

Remote Method Invocation

Remote Method Invocation



REMOTE METHOD INVOCATION

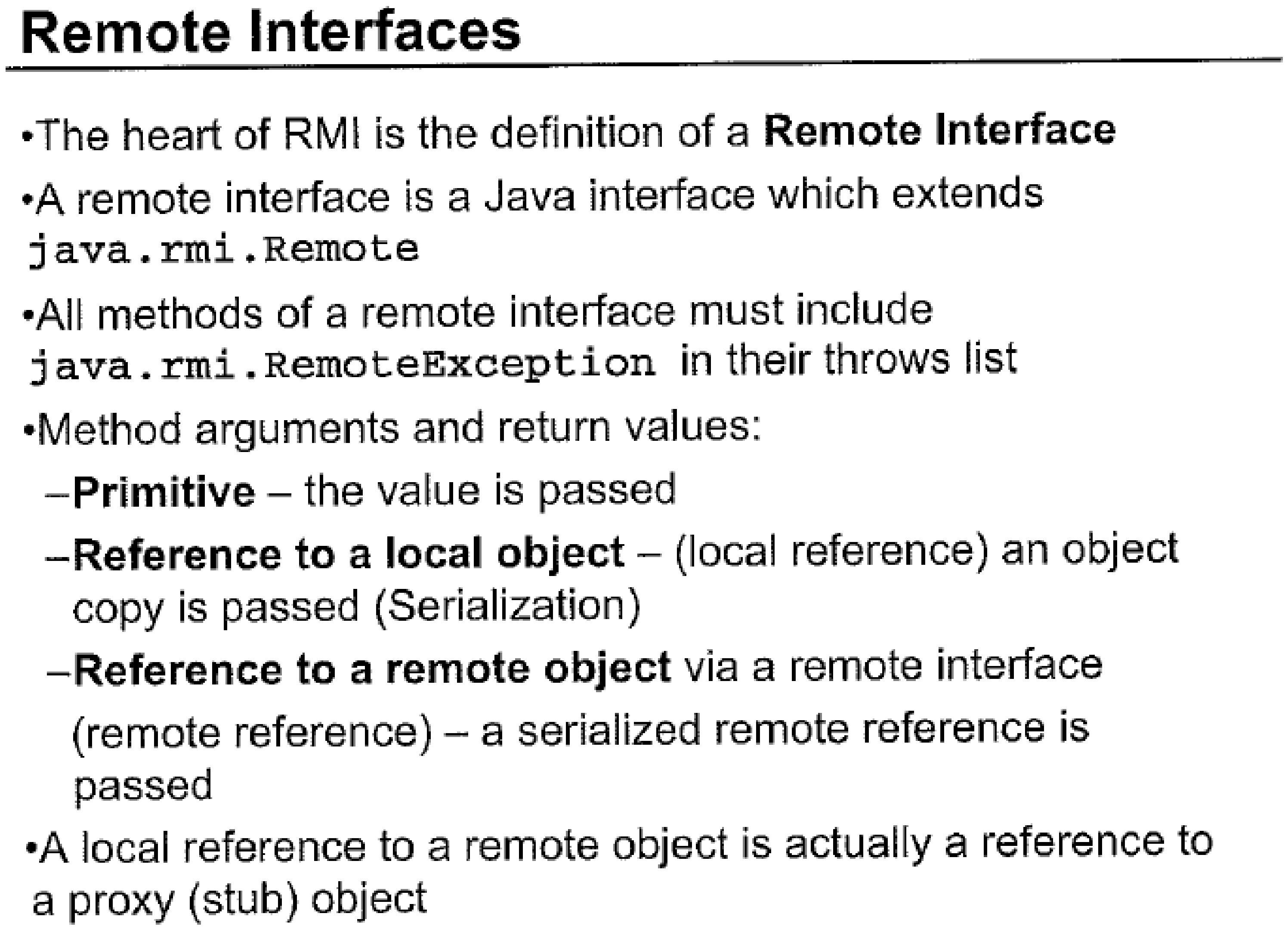


Szczecin, 19 kwietnia 2018 r.

REMOTE METHOD INVOCATION

If the object is a Remote object, a remote reference for the object is generated, and the reference is marshaled and sent to the remote process. If the object is serializable it is converted to bytes and sent to the remote process in byte form. If the method argument is neither remote nor serializable, the argument cannot be sent to the client and a java.rmi.MarshalException is thrown.

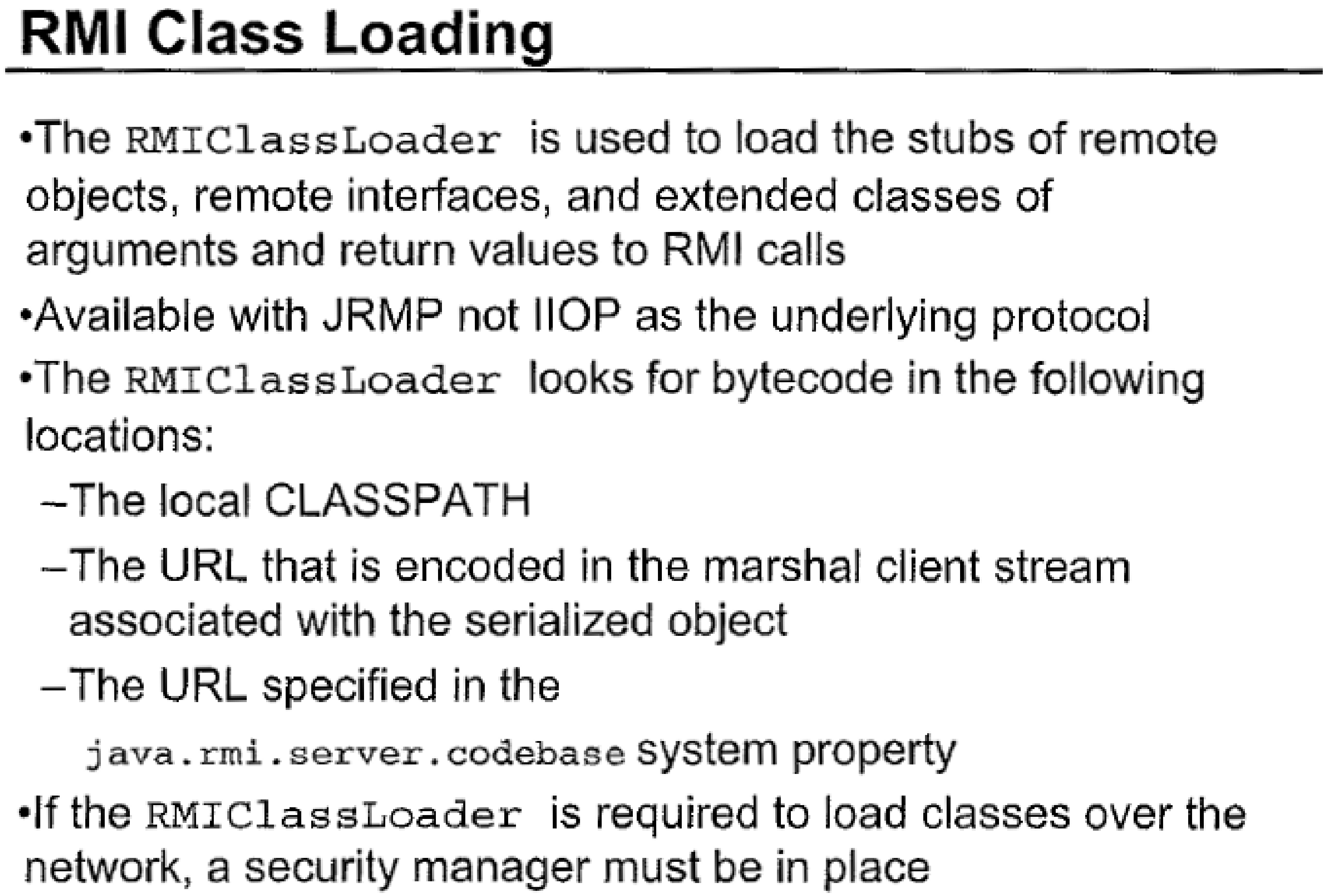
REMOTE METHOD INVOCATION



Remote Interfaces

- The heart of RMI is the definition of a **Remote Interface**
- A remote interface is a Java interface which extends `java.rmi.Remote`
- All methods of a remote interface must include `java.rmi.RemoteException` in their throws list
- Method arguments and return values:
  - **Primitive** – the value is passed
  - **Reference to a local object** – (local reference) an object copy is passed (Serialization)
  - **Reference to a remote object** via a remote interface (remote reference) – a serialized remote reference is passed
- A local reference to a remote object is actually a reference to a proxy (stub) object

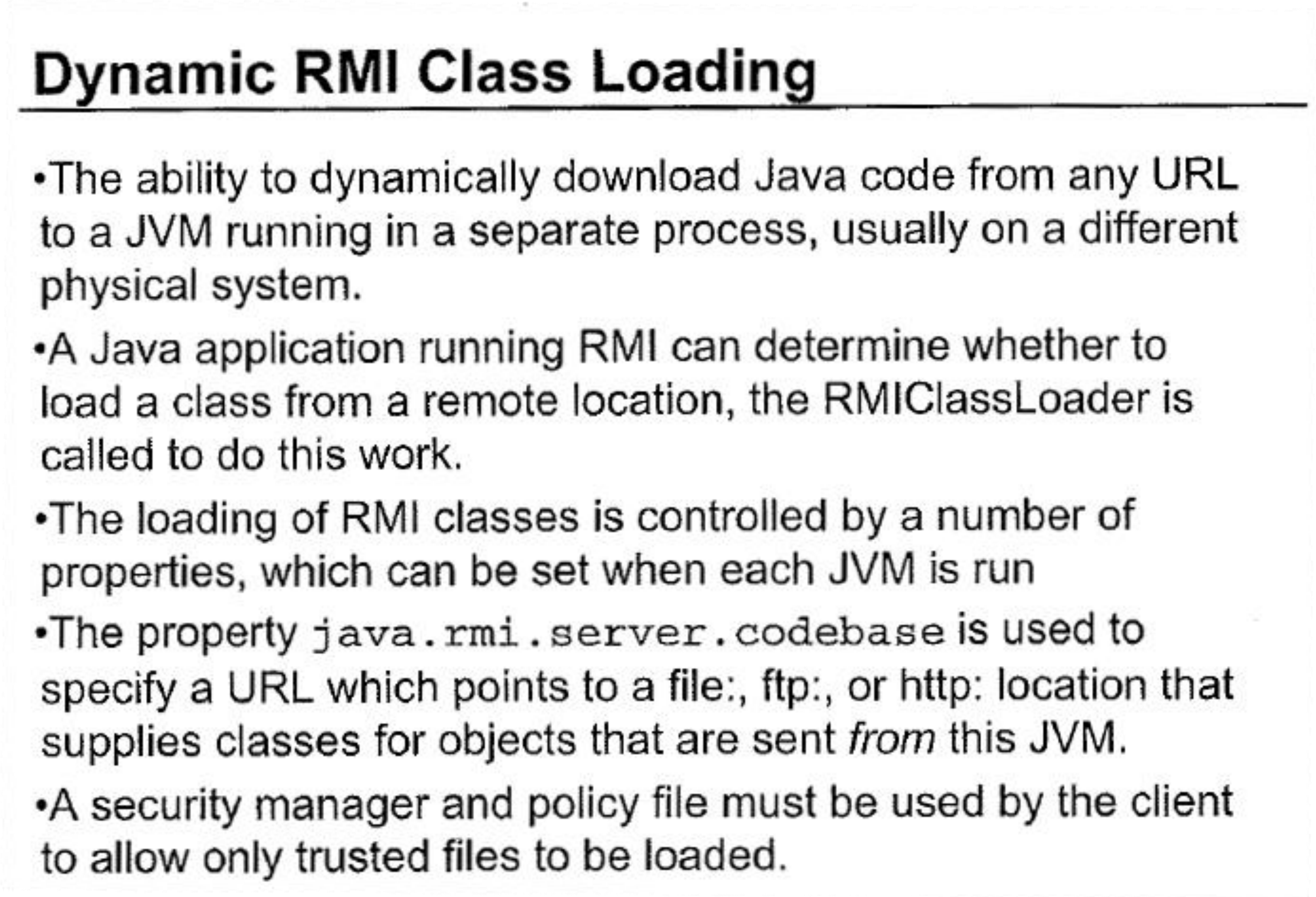
REMOTE METHOD INVOCATION



RMI Class Loading

- The `RMIClassLoader` is used to load the stubs of remote objects, remote interfaces, and extended classes of arguments and return values to RMI calls
- Available with JRMP not IIOP as the underlying protocol
- The `RMIClassLoader` looks for bytecode in the following locations:
  - The local CLASSPATH
  - The URL that is encoded in the marshal client stream associated with the serialized object
  - The URL specified in the `java.rmi.server.codebase` system property
- If the `RMIClassLoader` is required to load classes over the network, a security manager must be in place

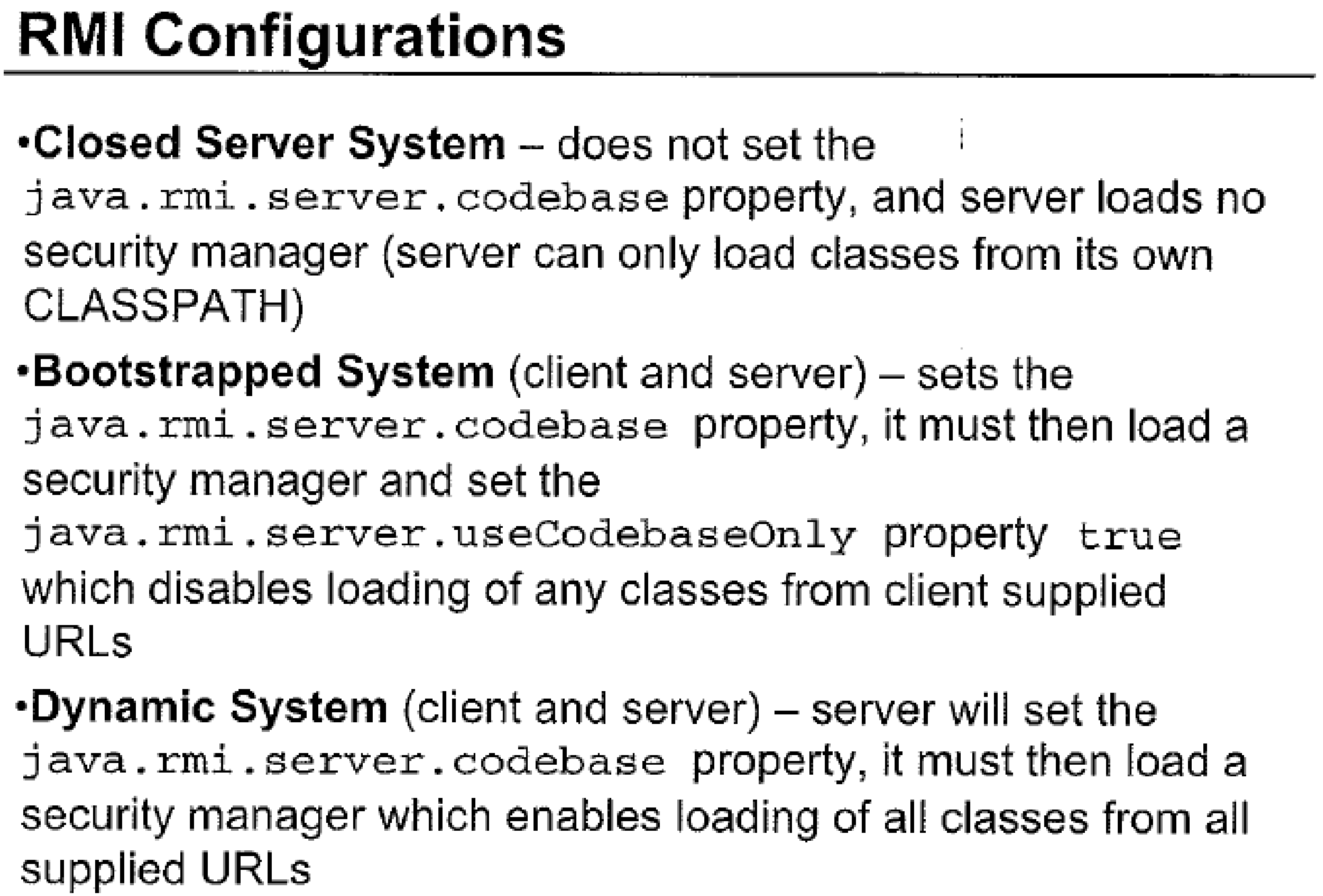
REMOTE METHOD INVOCATION



Dynamic RMI Class Loading

- The ability to dynamically download Java code from any URL to a JVM running in a separate process, usually on a different physical system.
- A Java application running RMI can determine whether to load a class from a remote location, the RMIClassLoader is called to do this work.
- The loading of RMI classes is controlled by a number of properties, which can be set when each JVM is run
- The property `java.rmi.server.codebase` is used to specify a URL which points to a file:, ftp:, or http: location that supplies classes for objects that are sent *from* this JVM.
- A security manager and policy file must be used by the client to allow only trusted files to be loaded.

REMOTE METHOD INVOCATION

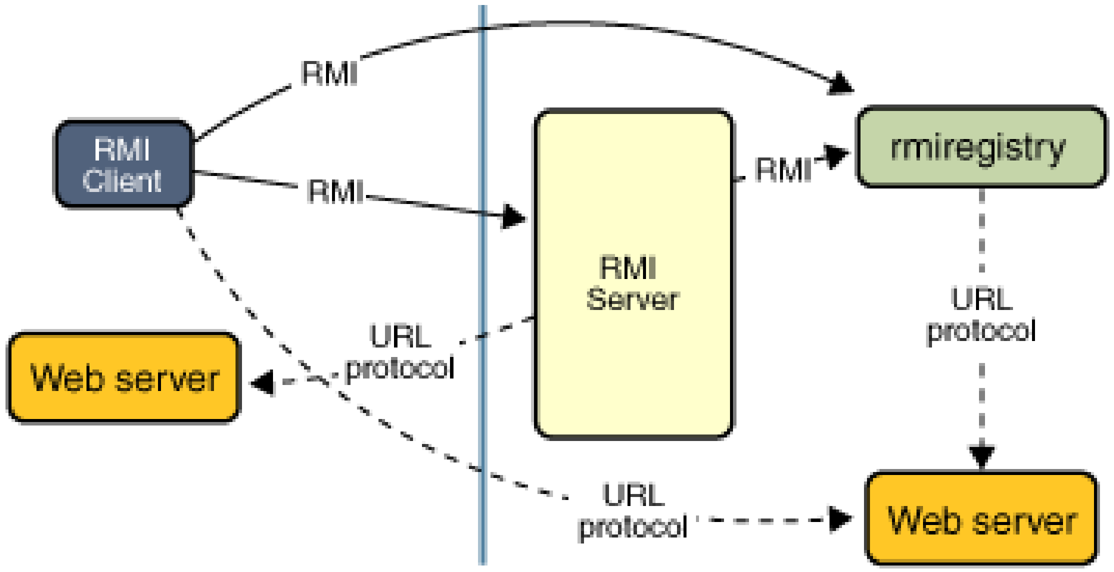


RMI Configurations

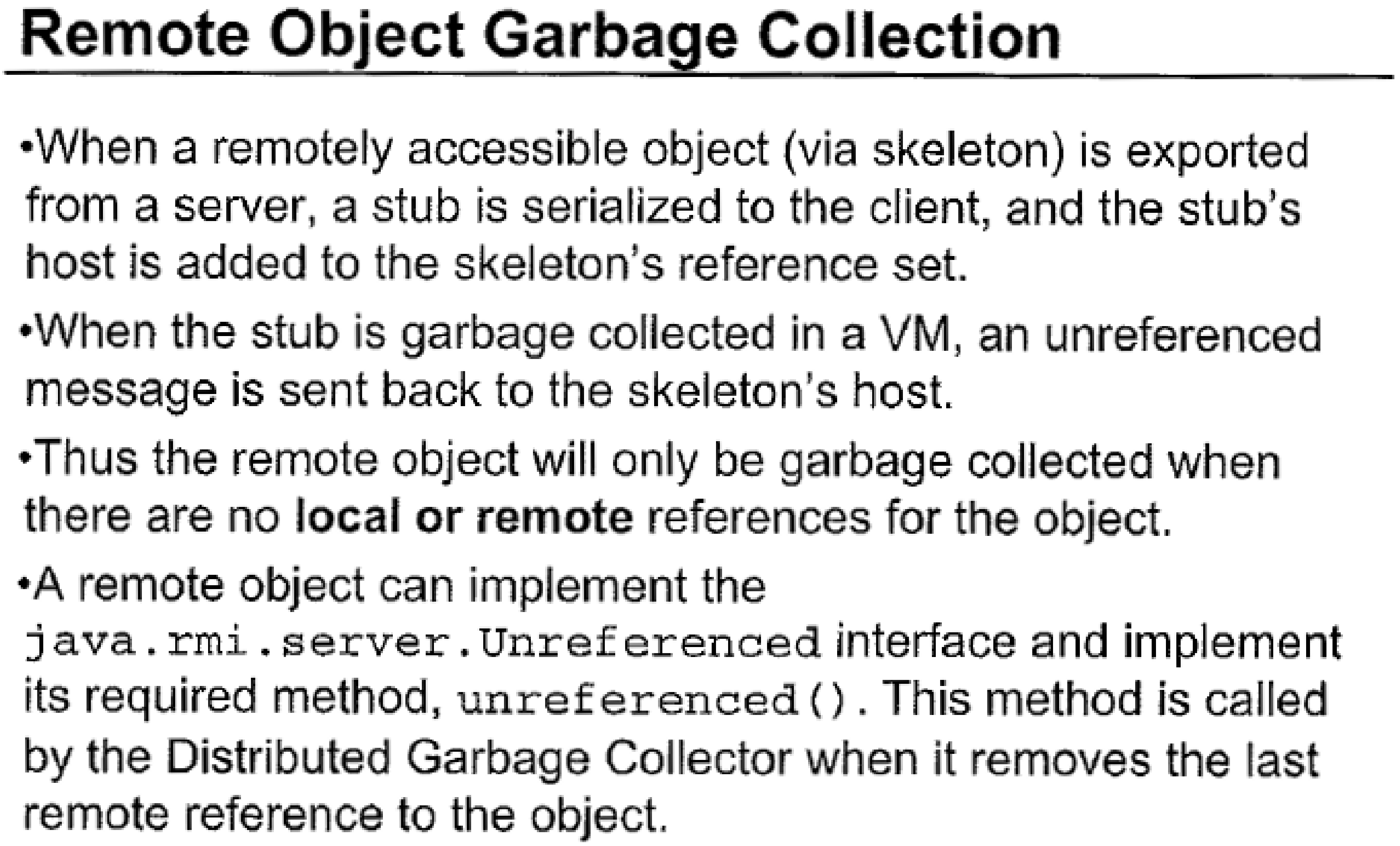
- **Closed Server System** – does not set the `java.rmi.server.codebase` property, and server loads no security manager (server can only load classes from its own CLASSPATH)
- **Bootstrapped System** (client and server) – sets the `java.rmi.server.codebase` property, it must then load a security manager and set the `java.rmi.server.useCodebaseOnly` property `true` which disables loading of any classes from client supplied URLs
- **Dynamic System** (client and server) – server will set the `java.rmi.server.codebase` property, it must then load a security manager which enables loading of all classes from all supplied URLs

RMI Configurations

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.



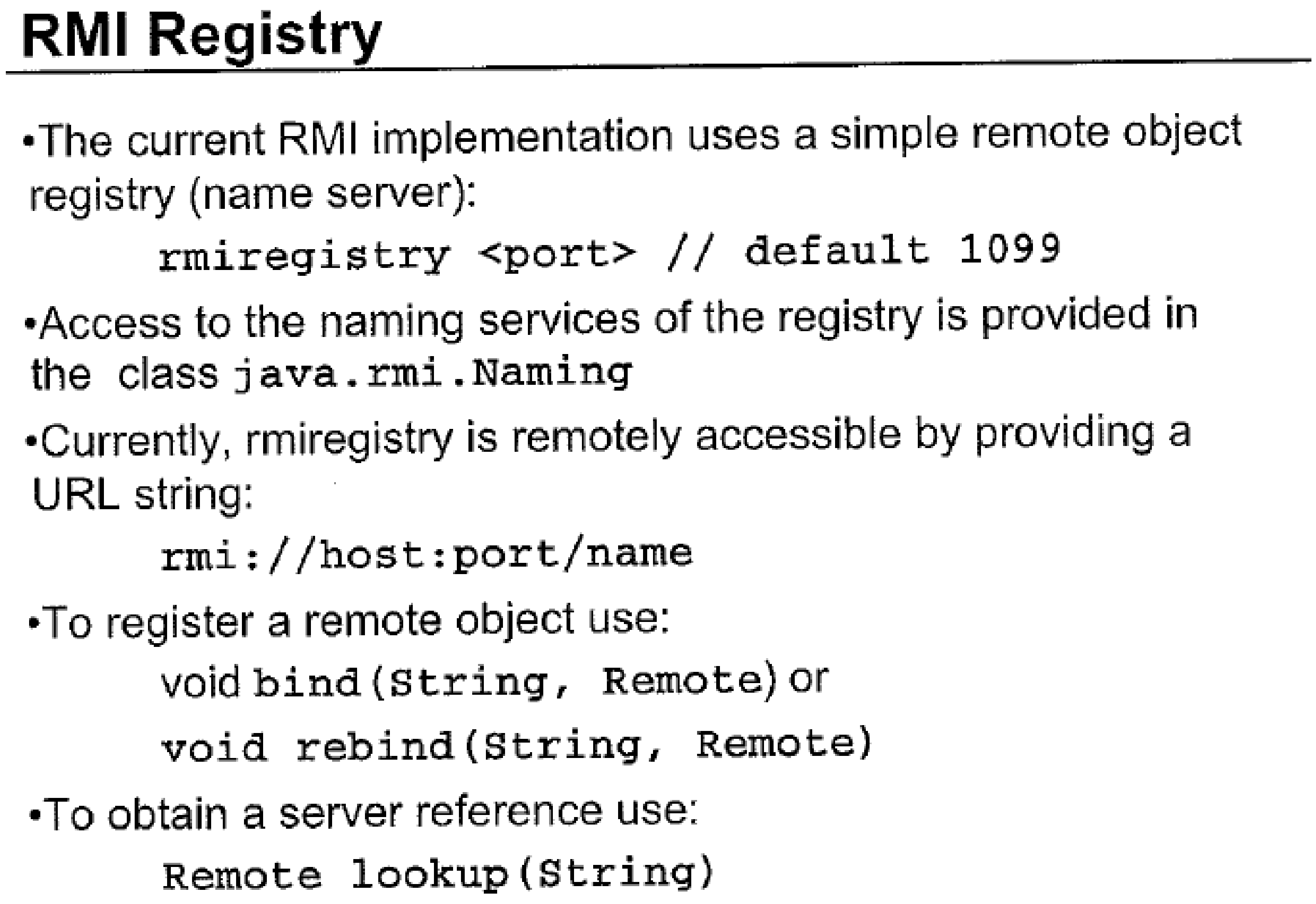
REMOTE METHOD INVOCATION



Remote Object Garbage Collection

- When a remotely accessible object (via skeleton) is exported from a server, a stub is serialized to the client, and the stub's host is added to the skeleton's reference set.
- When the stub is garbage collected in a VM, an unreferenced message is sent back to the skeleton's host.
- Thus the remote object will only be garbage collected when there are no **local or remote** references for the object.
- A remote object can implement the `java.rmi.server.Unreferenced` interface and implement its required method, `unreferenced()`. This method is called by the Distributed Garbage Collector when it removes the last remote reference to the object.

REMOTE METHOD INVOCATION



RMI Registry

- The current RMI implementation uses a simple remote object registry (name server):

```
rmiregistry <port> // default 1099
```

- Access to the naming services of the registry is provided in the class `java.rmi.Naming`
- Currently, rmiregistry is remotely accessible by providing a URL string:

```
rmi://host:port/name
```

- To register a remote object use:

  void `bind(String, Remote)` or

  `void rebind(String, Remote)`

- To obtain a server reference use:

```
Remote lookup(String)
```

RMI Registry

A client can enumerate all registered RMI objects by calling:

```
import javax.naming.NameClassPair;
...
Context context = new InitialContext();
Enumeration<NameClassPair> e = context.list("rmi://regserver.mycompany.com");
```

`NameClassPair` is a helper class that contains both the name of the bound object and the name of its class. For example, the following code displays the names of all registered objects:

```
while (e.hasMoreElements())
   System.out.println(e.nextElement().getName());
```

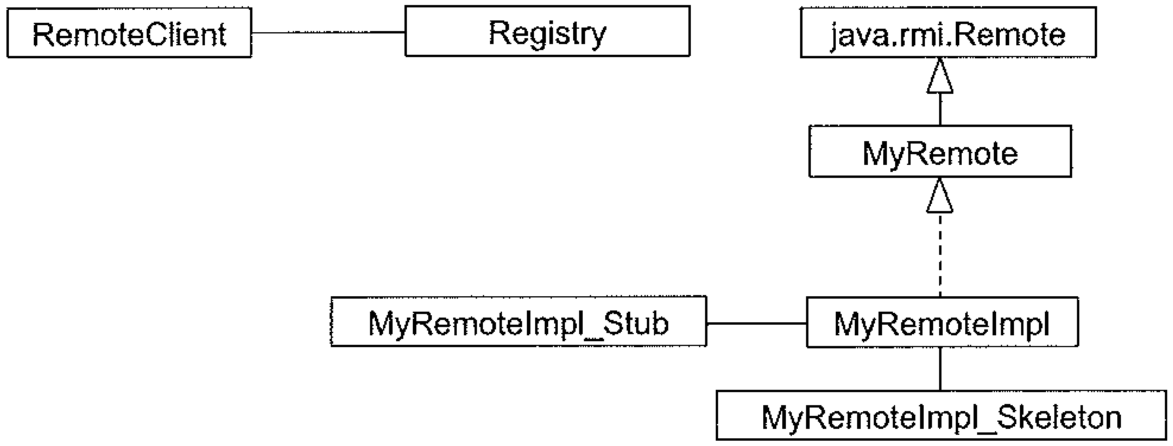
The following code is done to create and get the RMI registry running on the server :

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
...
LocateRegistry.createRegistry(1099);
...
Registry registry = LocateRegistry.getRegistry( );
```

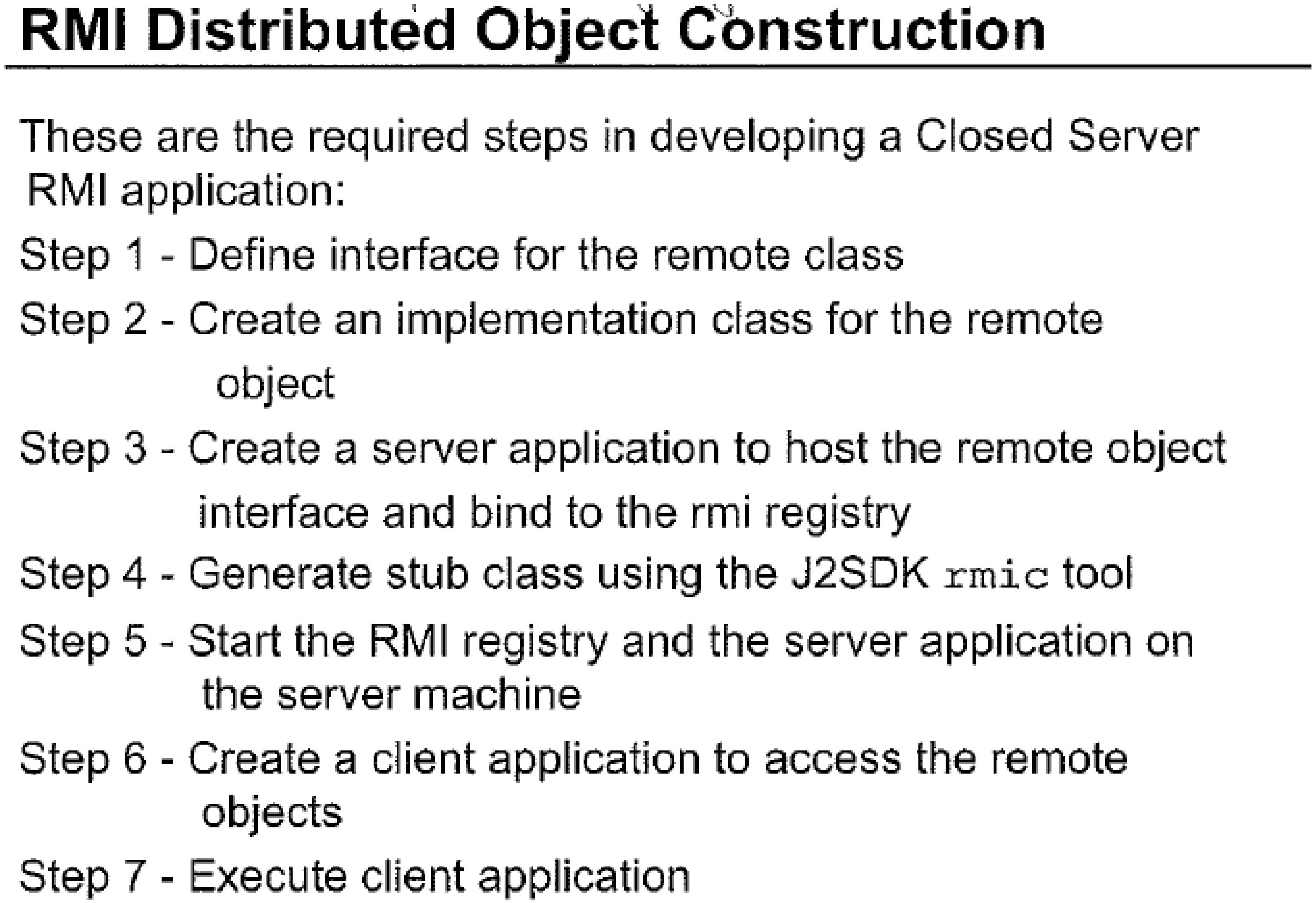
REMOTE METHOD INVOCATION

Creating a Remote Object

1. Extend the Remote Interface.
2. Implement the new interface and bind it to a naming service.
3. Generate stubs and skeletons that are needed for the remote implementations by using the rmic program.
4. Create a client program that will make RMI calls to the server.
5. Start the Registry and run your remote server and client.



REMOTE METHOD INVOCATION



RMI Distributed Object Construction

These are the required steps in developing a Closed Server RMI application:

Step 1 - Define interface for the remote class

Step 2 - Create an implementation class for the remote object

Step 3 - Create a server application to host the remote object interface and bind to the rmi registry

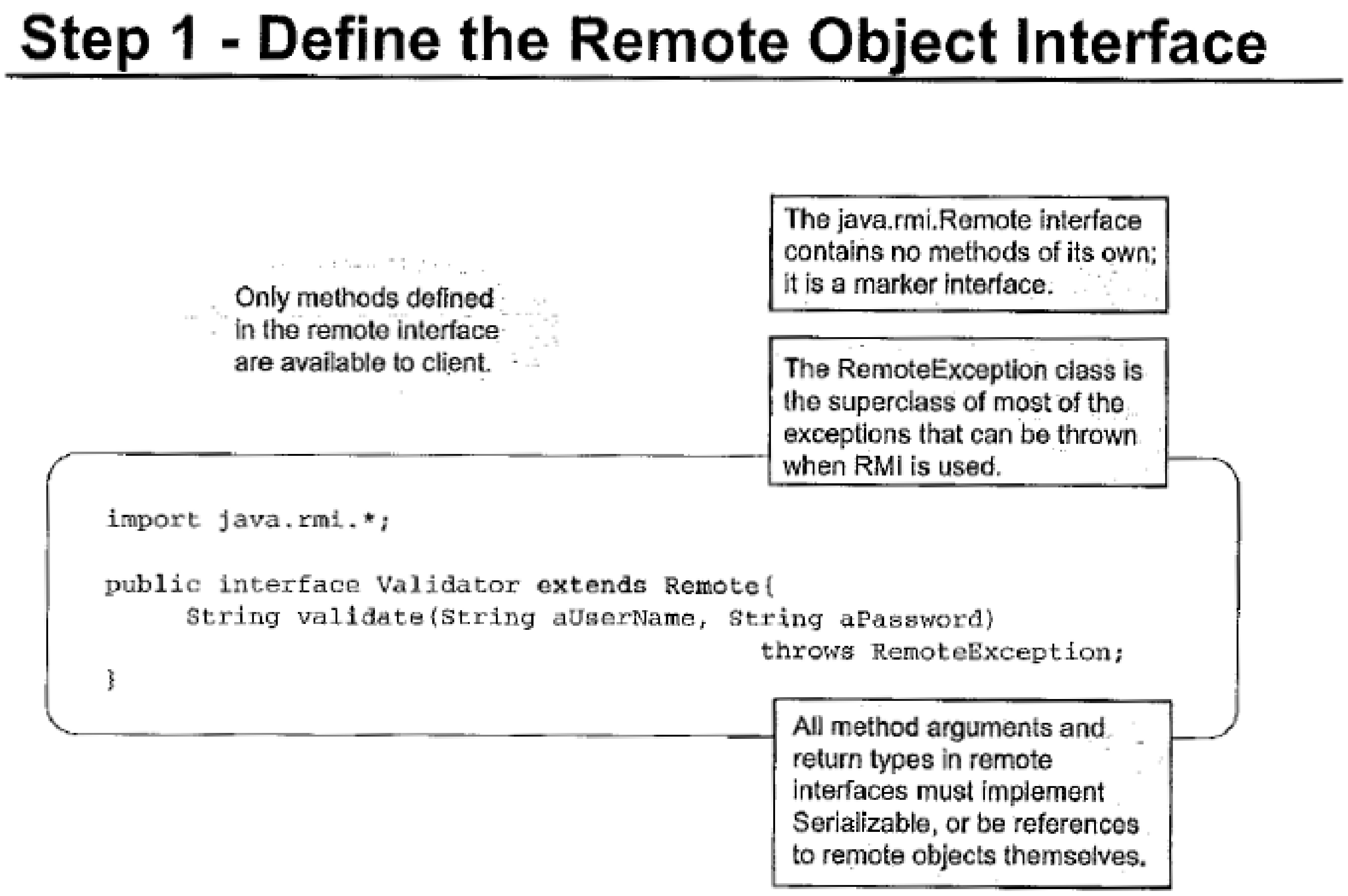
Step 4 - Generate stub class using the J2SDK `rmic` tool

Step 5 - Start the RMI registry and the server application on the server machine

Step 6 - Create a client application to access the remote objects

Step 7 - Execute client application

REMOTE METHOD INVOCATION



Step 1 - Define the Remote Object Interface

Only methods defined in the remote interface are available to client.

The java.rmi.Remote interface contains no methods of its own; it is a marker interface.

The RemoteException class is the superclass of most of the exceptions that can be thrown when RMI is used.

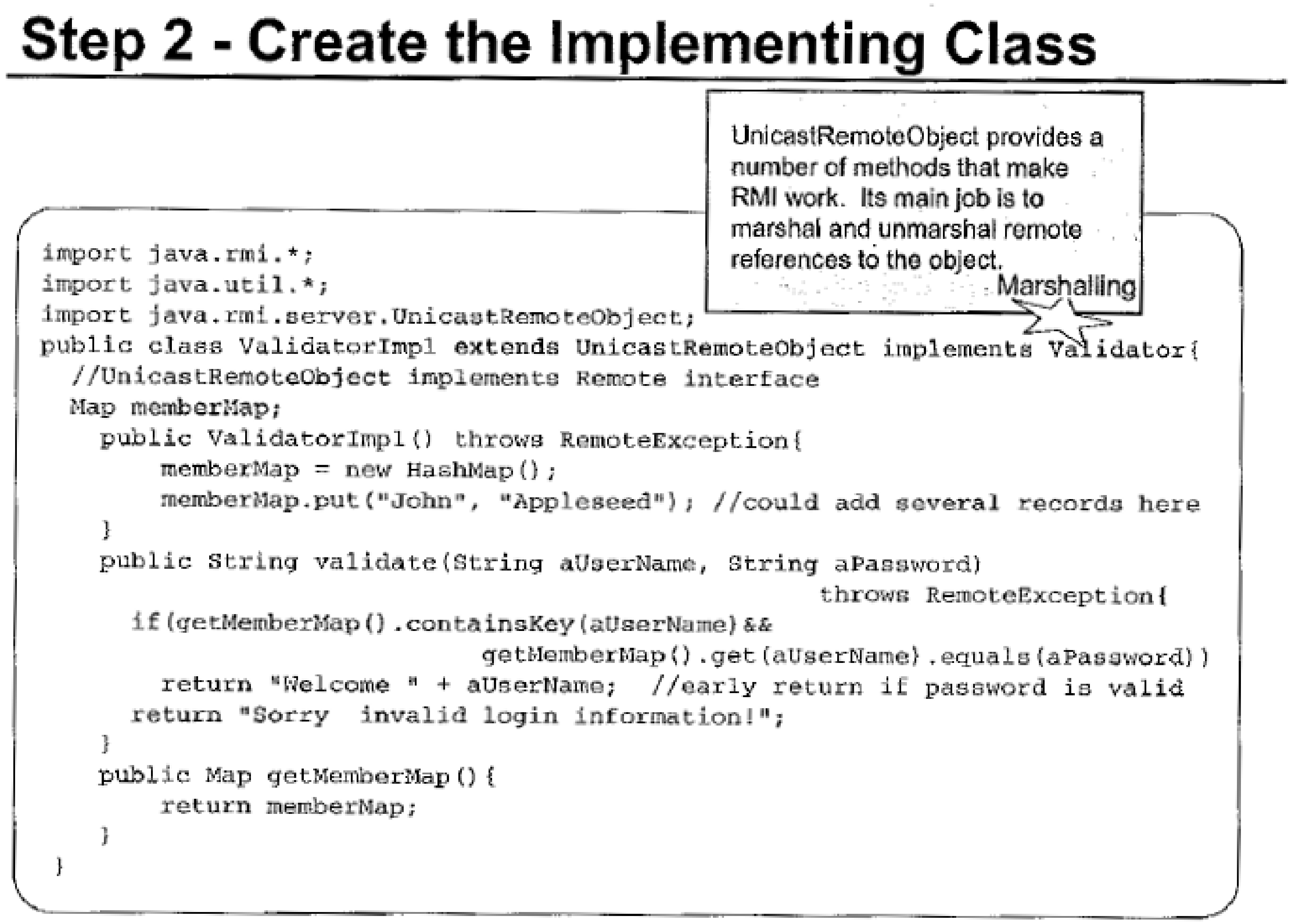
```
import java.rmi.*;

public interface Validator extends Remote{
     String validate(String aUserName, String aPassword)
                                        throws RemoteException;
}
```

All method arguments and return types in remote interfaces must implement Serializable, or be references to remote objects themselves.

Remote Method Invocation

Any method defined in the remote interface must throw a *RemoteException*. Remote methods depend on many things that are not under our control: for example, the state of the network and other necessary services such as DNS. Therefore, all code in the implementing class has to be in a try{} block, or the method must throw the exception. RemoteException is the superclass of all remote exceptions. There are 16 exceptions defined in the *java.rmi package*; they are all checked exceptions.

Remote Method Invocation



Step 2 - Create the Implementing Class

UnicastRemoteObject provides a number of methods that make RMI work. Its main job is to marshal and unmarshal remote references to the object.

Marshalling

```java
import java.rmi.*;
import java.util.*;
import java.rmi.server.UnicastRemoteObject;
public class ValidatorImpl extends UnicastRemoteObject implements Validator{
  //UnicastRemoteObject implements Remote interface
  Map memberMap;
    public ValidatorImpl() throws RemoteException{
        memberMap = new HashMap();
        memberMap.put("John", "Appleseed"); //could add several records here
    }
    public String validate(String aUserName, String aPassword)
                                                throws RemoteException{
      if(getMemberMap().containsKey(aUserName)&&
                              getMemberMap().get(aUserName).equals(aPassword))
        return "Welcome " + aUserName;  //early return if password is valid
      return "Sorry  invalid login information!";
    }
    public Map getMemberMap(){
        return memberMap;
    }
 }
```

Step 2 – Create the Implementing Class

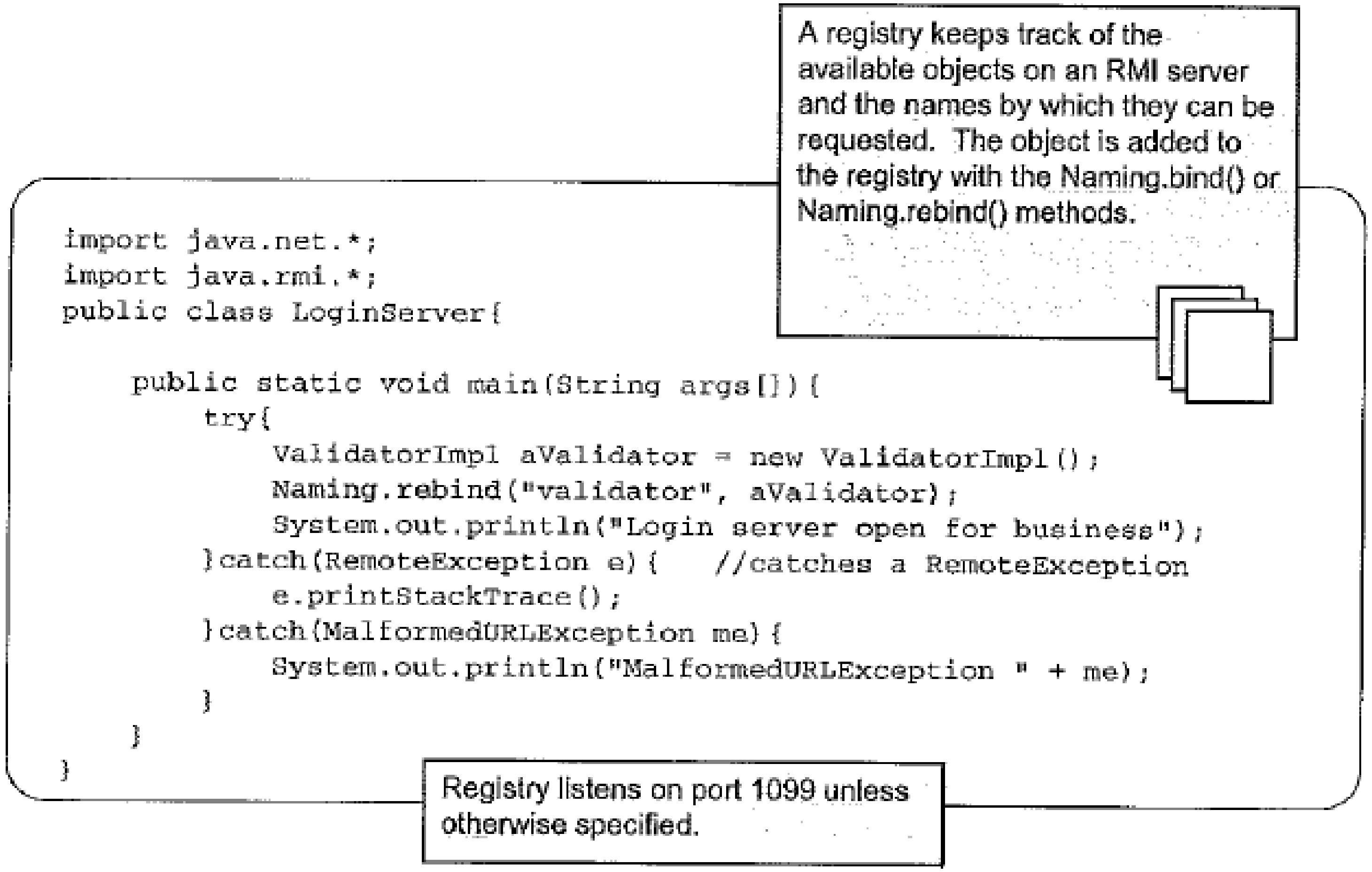
Occasionally, you might not want to extend the `UnicastRemoteObject` class, perhaps because your implementation class already extends another class. In that situation, you need to manually instantiate the remote objects and pass them to the static `exportObject` method. Instead of extending `UnicastRemoteObject`, call:

```
 UnicastRemoteObject.exportObject(this, 0);
```

in the constructor of the remote object. The second parameter is 0 to indicate that any suitable port can be used to listen to client connections.

REMOTE METHOD INVOCATION

Step 3 – Create Server Application



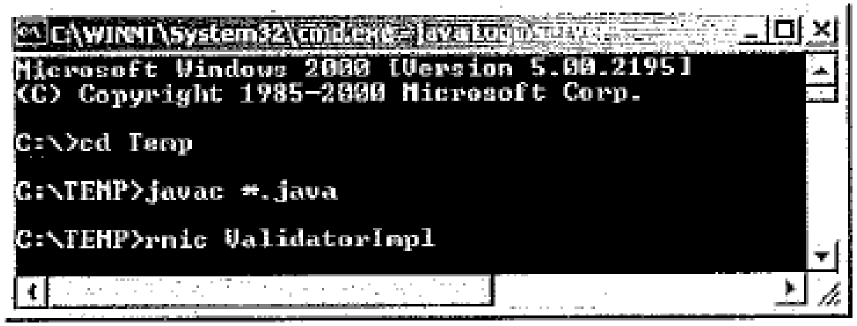
A registry keeps track of the available objects on an RMI server and the names by which they can be requested. The object is added to the registry with the Naming.bind() or Naming.rebind() methods.

```
import java.net.*;
import java.rmi.*;
public class LoginServer{

    public static void main(String args[]){
        try{
            ValidatorImpl aValidator = new ValidatorImpl();
            Naming.rebind("validator", aValidator);
            System.out.println("Login server open for business");
        }catch(RemoteException e){    //catches a RemoteException
            e.printStackTrace();
        }catch(MalformedURLException me){
            System.out.println("MalformedURLException " + me);
        }
    }
}
```

Registry listens on port 1099 unless otherwise specified.

Step 4 - Generate the Stubs/Skeletons

- Stub represents the interface on the client
  `ValidateImpl_Stub.class`
- Skeleton bridges the interface to the server
  `ValidateImpl_Skel.class`
- rmic compiler generates stub and skeleton classes
  `C:\Temp>rmic ValidateImpl`



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>cd Temp

C:\TEMP>javac *.java

C:\TEMP>rmic ValidatorImpl
```

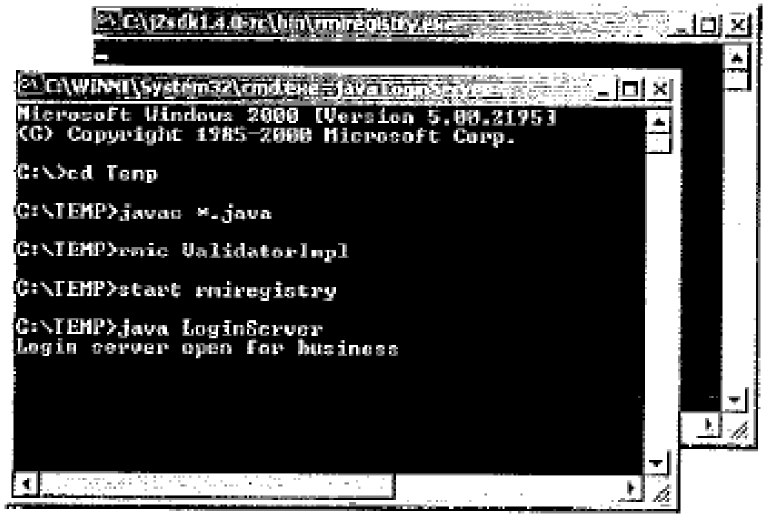
Step 5 – Start RMI Registry and Server Application

•Start the rmi registry (Naming Registry). It runs in its own window.

```
C:>start rmiregistry
```

•Finally start our own server

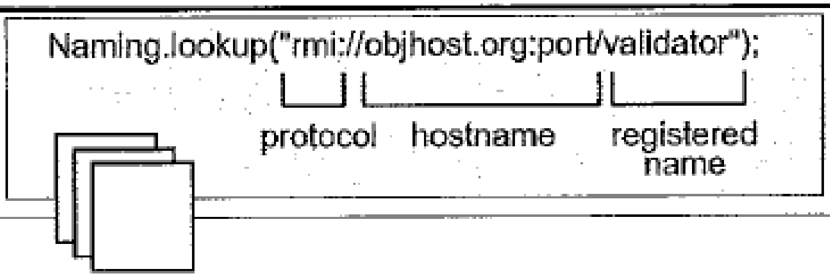
```
C:>java LoginServer
Outputs: Login Server open for business
```


C:\j2sdk1.4.0-rc\bin\rmiregistry.exe
C:\WINNT\System32\cmd.exe - java LoginServer

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>cd Temp

C:\TEMP>javac *.java

C:\TEMP>rmic ValidatorImpl

C:\TEMP>start rmiregistry

C:\TEMP>java LoginServer
Login server open for business
```

Remote Method Invocation

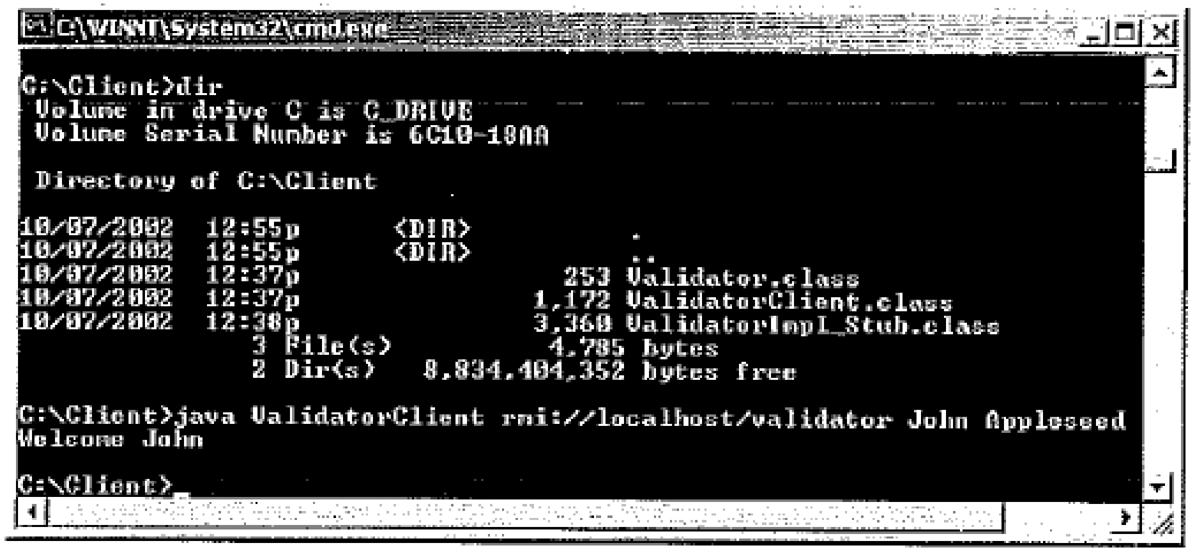
Step 6 - Create the Client



```
import java.rmi.*;
import java.net.*;

public class ValidatorClient{
  public static void main(String args[]){
    if(args.length==0||!args[0].startsWith("rmi:")){
     System.out.println("Usage: java ValidatorClient" +
                          "rmi://host.domain.port/validator username password");
    }
    try{
      Object remote  = Naming.lookup(args[0]);
      Validator reply = (Validator)remote;
      System.out.println(reply.validate(args[1], args[2]));
    }catch(MalformedURLException me){
      System.out.println(args[0] + " is not a valid URL");
    }catch(RemoteException nbe){
       System.out.println("Could not find requested object on the server");
    }
  }
}
```

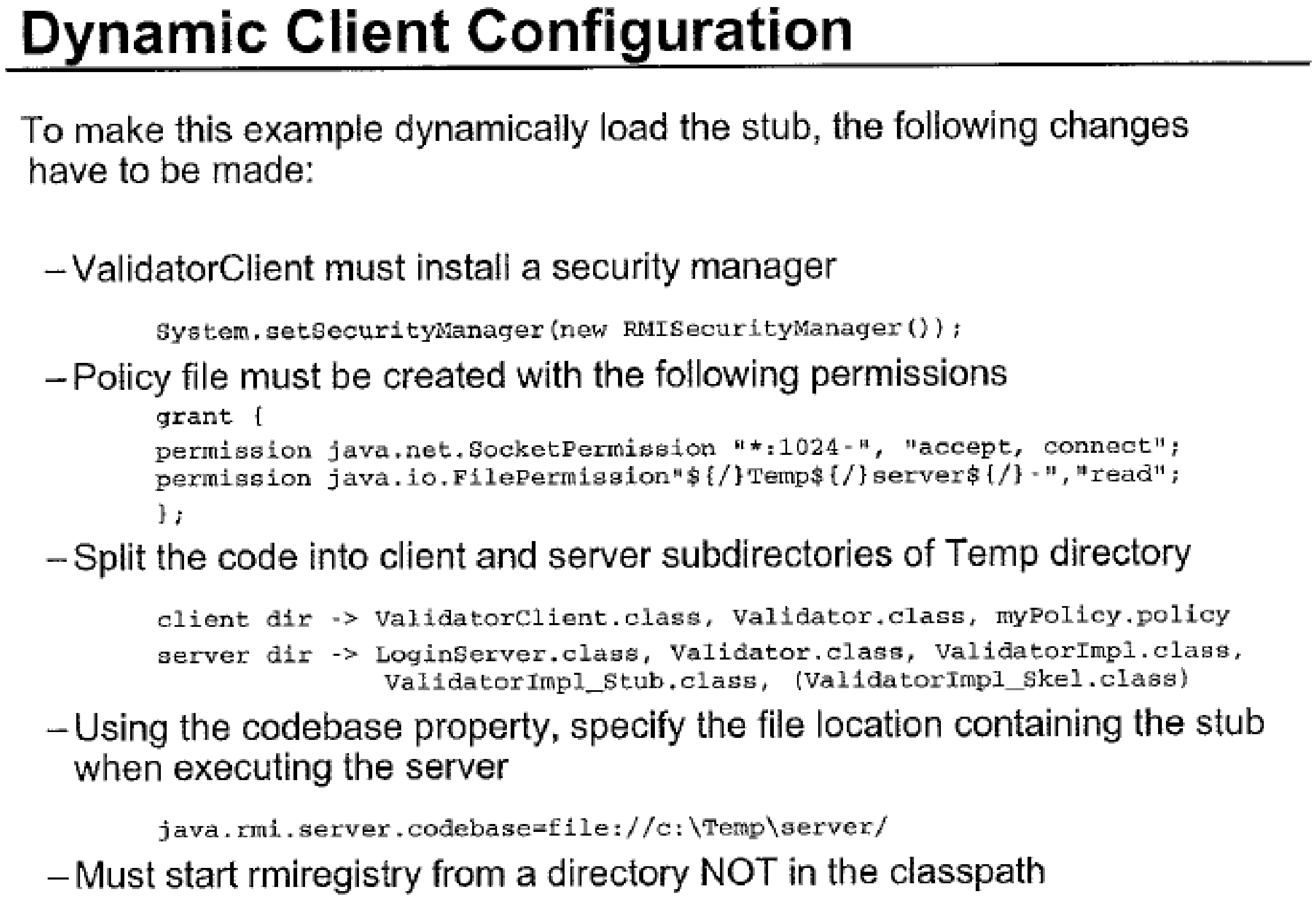
Step 7 – Execute Client Application

This is a Closed Server System example, so both the `Validator` interface and the `ValidatorImpl_Stub` that was generated have to be in the client JVM's CLASSPATH.



```
C:\Client>dir
 Volume in drive C is C_DRIVE
 Volume Serial Number is 6C10-18AA

 Directory of C:\Client

10/07/2002  12:55p      <DIR>          .
10/07/2002  12:55p      <DIR>          ..
10/07/2002  12:37p                 253 Validator.class
10/07/2002  12:37p               1,172 ValidatorClient.class
10/07/2002  12:38p               3,360 ValidatorImpl_Stub.class
               3 File(s)          4,785 bytes
               2 Dir(s)   8,834,404,352 bytes free

C:\Client>java ValidatorClient rmi://localhost/validator John Applessed
Welcome John

C:\Client>_
```

REMOTE METHOD INVOCATION



Dynamic Client Configuration

To make this example dynamically load the stub, the following changes have to be made:

– ValidatorClient must install a security manager

```
System.setSecurityManager(new RMISecurityManager());
```

– Policy file must be created with the following permissions

```
grant {
permission java.net.SocketPermission "*:1024-", "accept, connect";
permission java.io.FilePermission"${/}Temp${/}server${/}-","read";
};
```

– Split the code into client and server subdirectories of Temp directory

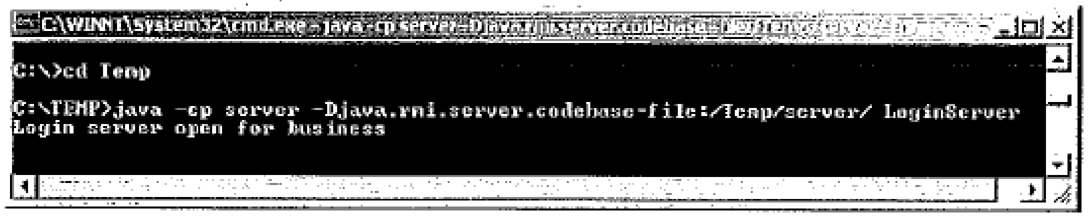
```
client dir -> ValidatorClient.class, Validator.class, myPolicy.policy
server dir -> LoginServer.class, Validator.class, ValidatorImpl.class,
              ValidatorImpl_Stub.class, (ValidatorImpl_Skel.class)
```

– Using the codebase property, specify the file location containing the stub when executing the server

```
java.rmi.server.codebase=file://c:\Temp\server/
```

– Must start rmiregistry from a directory NOT in the classpath

Dynamic Client Configuration Example

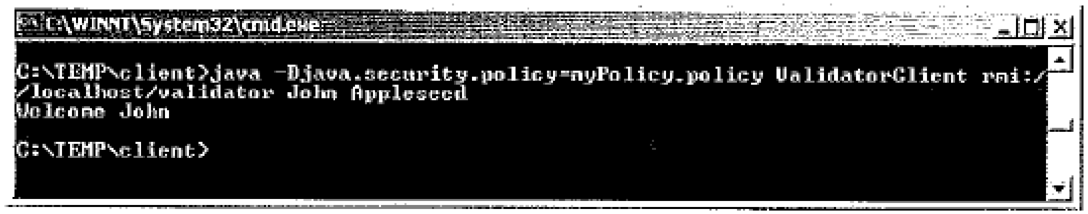

C:\WINNT\System32\cmd.exe - java -cp server -Djava.rmi.server.codebase=file:/Temp/server/ [illegible]

```
C:\>cd Temp

C:\TEMP>java -cp server -Djava.rmi.server.codebase=file:/Temp/server/ LoginServer
Login server open for business
```

server started
from c:\Temp

rmiregistry started
from c:\

client started
from c:\temp\client



```
C:\TEMP\client>java -Djava.security.policy=myPolicy.policy ValidatorClient rmi:/
/localhost/validator John Appleseed
Welcome John

C:\TEMP\client>
```

The trailing forward slash at the end of the codebase url
-Djava.rmi.server.codebase=file:/Temp/server/ signifies that it is a directory. Unless a jar file is being referenced, the forward slash must be used.

KONIEC

DZIĘKUJĘ ZA UWAGĘ!!!

Szczecin, 19 kwietnia 2018 r.