Distributed Programming (aka InfoRep)
Chapter 3 - RMI

Cecilia ZANNI-MERK
INSA Rouen Normandie

Agenda

- A (very brief) introduction to networks
- Client-Server architectures
- Sockets
- Remote Method Invocation

Network Programming in Java

Two approaches

- Data stream communication (`java.net`, `java.io` packages)
- Remote method call (`java.rmi` package)

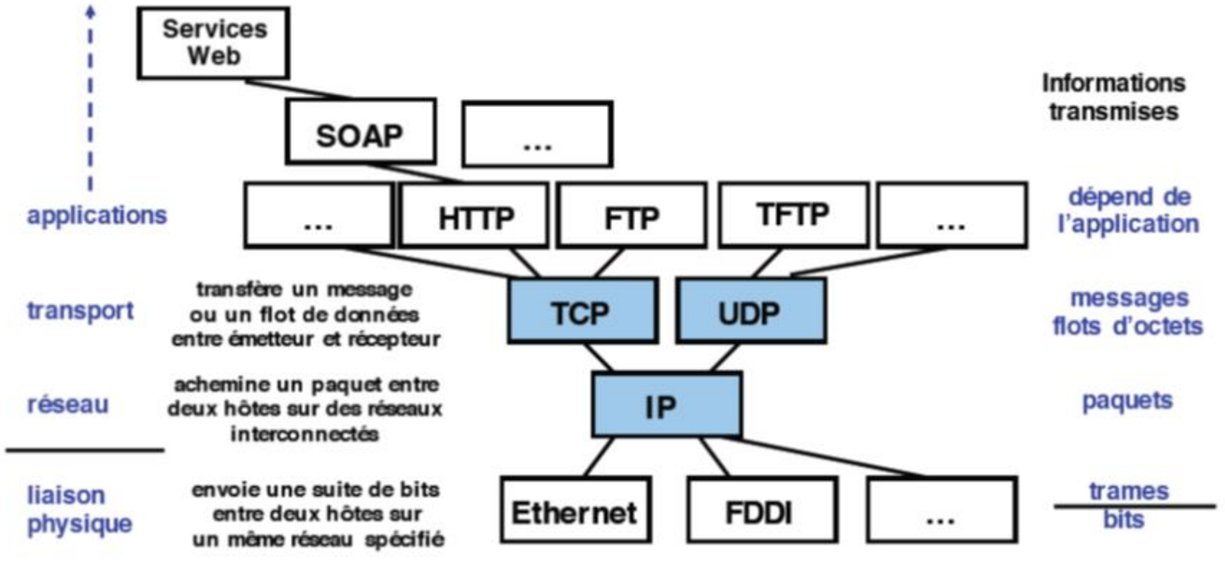
`java.net`

- classes for raw byte transmission
- operation with sockets communicating via network ports

`java.rmi`

- higher level of abstraction for remote method calling
- the virtual machine translates the calls into a communication with sockets (hidden for the programmer)
- the virtual machine provides an object location service

Computer networks in one slide


Services Web
SOAP
...
applications
...
HTTP
FTP
TFTP
...
Informations transmises
dépend de l'application
transport
transfère un message ou un flot de données entre émetteur et récepteur
TCP
UDP
messages flots d'octets
réseau
achemine un paquet entre deux hôtes sur des réseaux interconnectés
IP
paquets
liaison physique
envoie une suite de bits entre deux hôtes sur un même réseau spécifié
Ethernet
FDDI
...
trames
bits

HTTP : *HyperText Transfer Protocol* : protocole du Web
TFTP, FTP : (*Trivial) File Transfer Protocol*) : transfert de fichiers
TCP : *Transmission Control Protocol* : transport en mode connecté
UDP : *User Datagram Protocol* : transport en mode non connecté
IP : *Internet Protocol* : Interconnexion de réseaux, routage

Remote Method Invocation

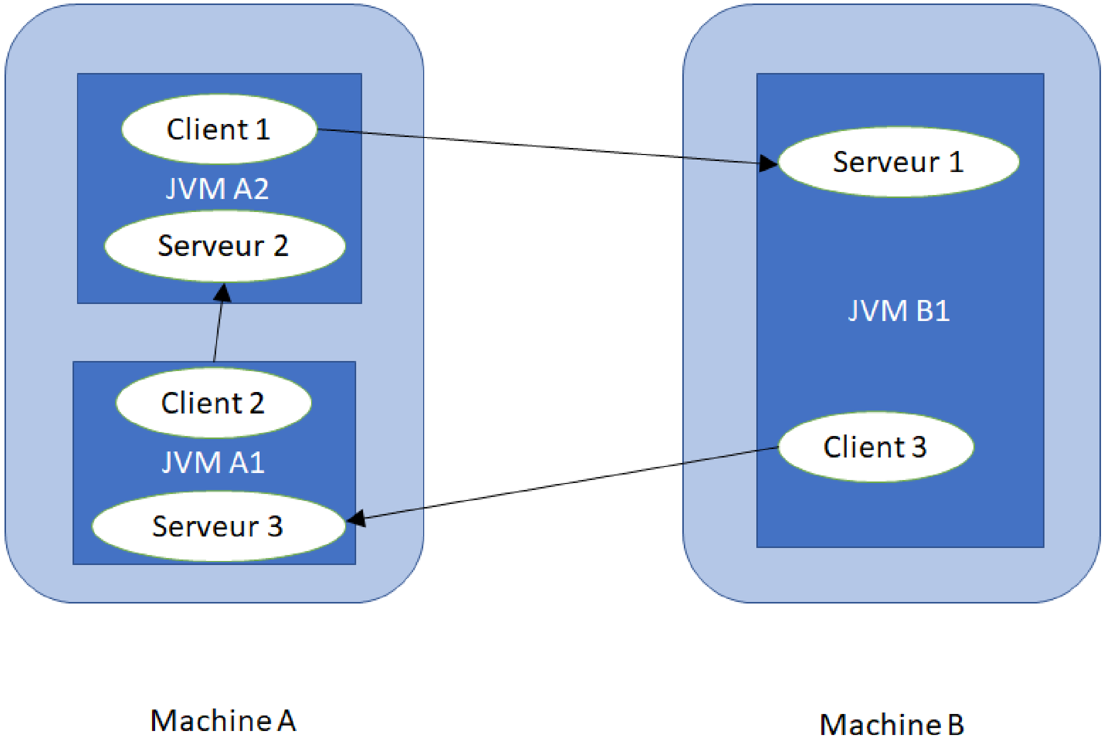
RMI (Remote Method Invocation)

A simple way to develop client-server programs, based on the RPC (Remote Procedure Call) protocol

The goal is to allow client applications (running locally) to invoke methods on remote objects, i.e. located in another application (in another JVM of the same physical machine or on another machine accessible via the Internet) commonly called server.

Servers and clients are objects

RMI (Remote Method Invocation)



Main concepts

An application running on an M1 machine can create an object and make it accessible to other applications: this application (and the M1 machine) thus plays the role of server. Other applications handling such an object are clients.

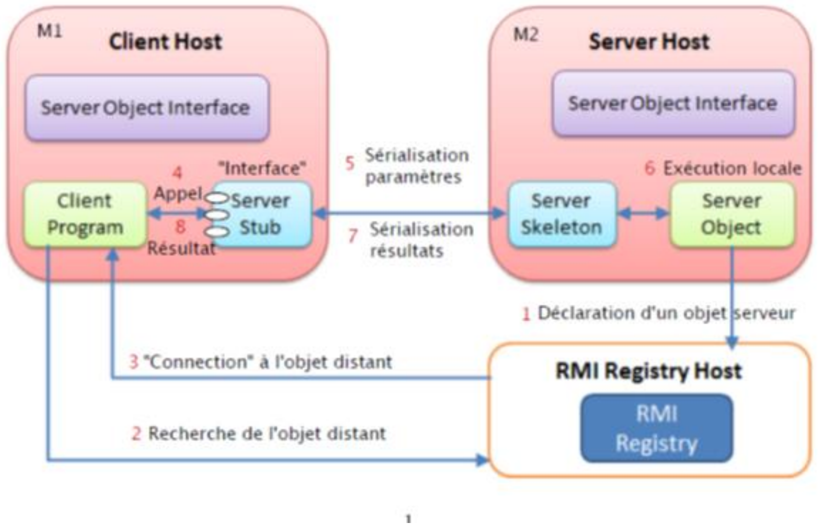
To manipulate a remote object, a client retrieves on its machine a representation of the object called **proxy** or **stub**.

Main concepts

The **proxy** is an object that will make the link between a local interface and the remote object: it is via this **stub** that the client will be able to invoke methods on the remote object. Such an invocation will be transmitted to the server (the TCP protocol is used) in order to execute it.

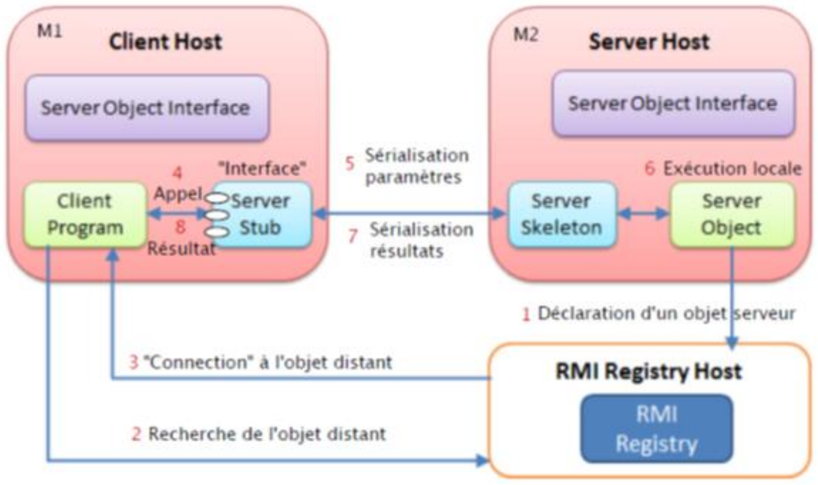
On the server side, a **skeleton** is in charge of receiving remote invocations, their realization and of sending the results to the client

Implementation



1. First, the server on the M2 machine will declare the service it is ready to provide
   Name server: `rmiregistry`
2. The client on the M1 machine will ask the name server to resolve the name of the M2 machine
3. Creation of an interface connected to the remote object (proxy)
4. The client on the M1 machine calls a method on this interface

Implementation



1

5. The stub on the M1 machine
   a. packages the method identifier and its arguments (serialization) ;
   b. the request is transmitted over the network;
6. The skeleton on the M2 machine
   a. receives and unpacks the message (deserialization);
   b. calls the requested method;
   c. receives the result of the method;
7. The skeleton
   a. packs this result;
   b. transmits the result to the proxy on the M1 machine;
8. The proxy on the M1 machine
   a. receives and unpacks the message;
   b. returns the result as an ordinary method.

JAVA RMI

Protocol and tool for remote method invocation in Java

A set of classes in packages:

```
java.rmi
java.rmi.server
java.rmi.registry
java.rmi.dgc
java.rmi.activation
```

A "server": `rmiregistry`

Transport protocol used: **JRMP** (Java Remote Method Protocol)

JAVA RMI

Object-oriented

Invocation of synchronous methods

Passing parameters of remote invocations

- Single type: passage by value
- Instance of a class that implements `Serializable`: passage by value
- Instance of a class that implements `Remote` and referenced as a distributed object: it is a **stub** that is sent
- Otherwise an error is generated

Directory service (`rmiregistry`); Default port: 1099

Development cycle

Server-side development consists of:

- Definition of an interface that contains the methods that can be called remotely
- Writing a class that implements this interface
- Writing a class that will instantiate the object and save it by assigning it a name in the RMI name register (`rmiregistry`)

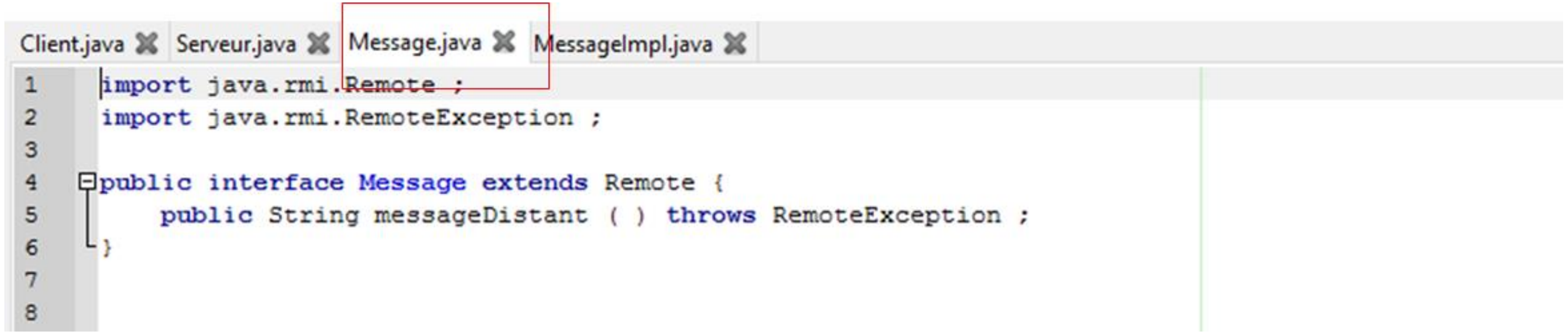
Client-side development consists of:

- Obtaining a reference on the remote object from its name
- Calling the method from this reference

The stub and the skeleton are generated automatically

Server example

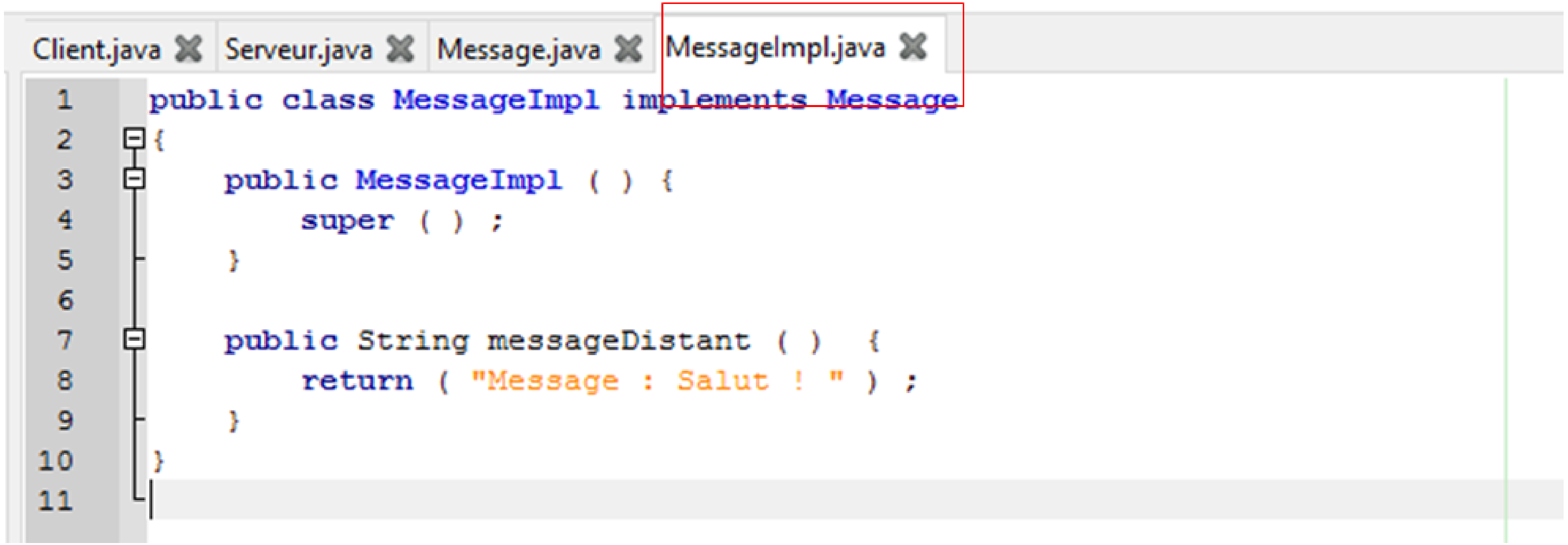
Definition of the access interface to the remote addressable object



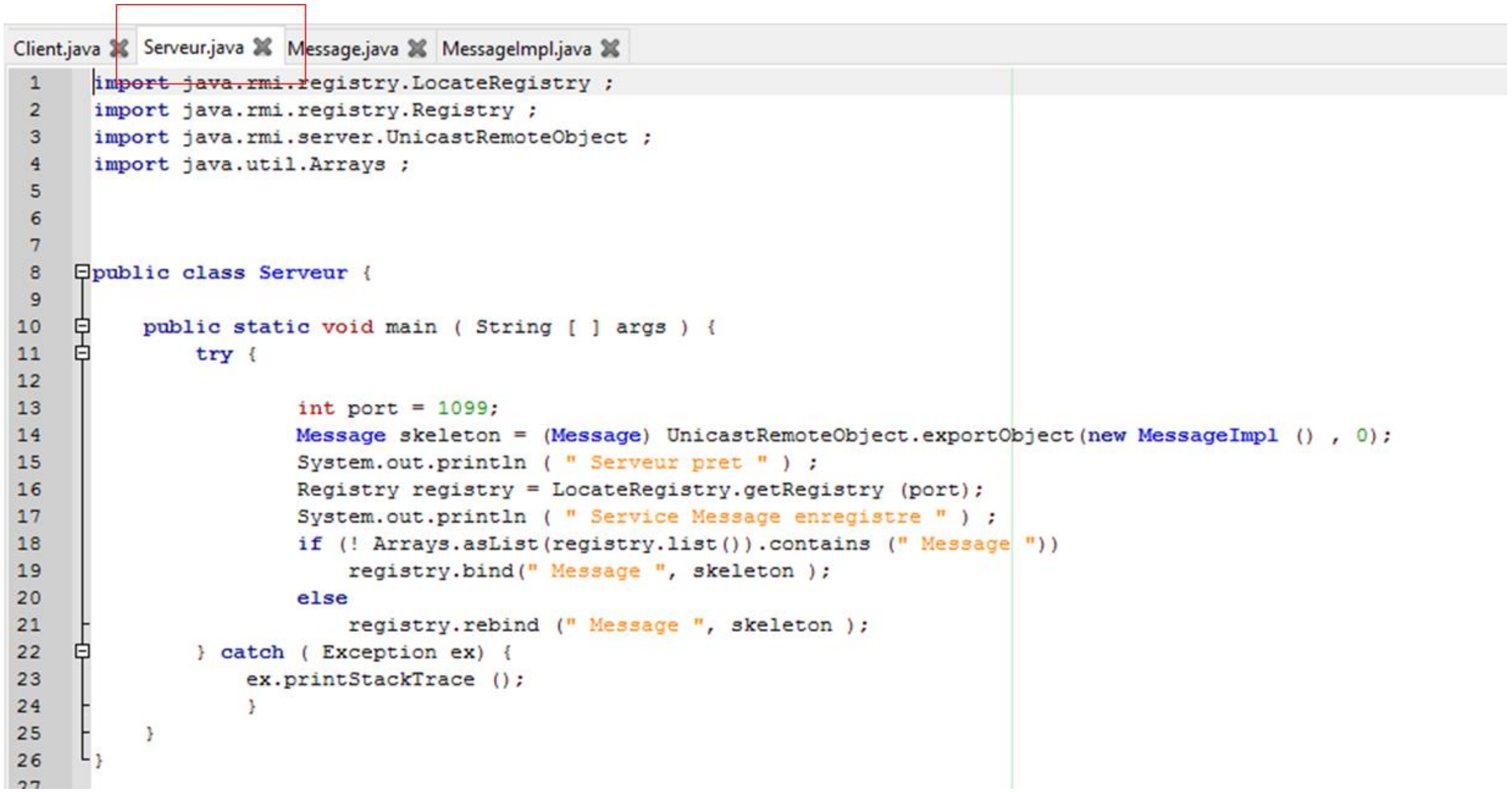
```java
import java.rmi.Remote ;
import java.rmi.RemoteException ;

public interface Message extends Remote {
    public String messageDistant ( ) throws RemoteException ;
}
```

Server example

Definition of the class implementing the code that will actually perform the operations defined in the interface



```java
public class MessageImpl implements Message
{
    public MessageImpl ( ) {
        super ( ) ;
    }

    public String messageDistant ( )  {
        return ( "Message : Salut ! " ) ;
    }
}
```


Client.java Serveur.java Message.java MessageImpl.java

```java
import java.rmi.registry.LocateRegistry ;
import java.rmi.registry.Registry ;
import java.rmi.server.UnicastRemoteObject ;
import java.util.Arrays ;



public class Serveur {

    public static void main ( String [ ] args ) {
        try {

                int port = 1099;
                Message skeleton = (Message) UnicastRemoteObject.exportObject(new MessageImpl () , 0);
                System.out.println ( " Serveur pret " ) ;
                Registry registry = LocateRegistry.getRegistry (port);
                System.out.println ( " Service Message enregistre " ) ;
                if (! Arrays.asList(registry.list()).contains (" Message "))
                    registry.bind(" Message ", skeleton );
                else
                    registry.rebind (" Message ", skeleton );
        } catch ( Exception ex) {
            ex.printStackTrace ();
            }
    }
}
```

Naming service

On the server, the RMI name registry must run before it can register an object or obtain a reference.

This registry can be launched as an application provided in the JDK (`rmiregistry` on the command line) or be launched dynamically in the class that registers the object.

The naming service is an RMI object; rmiregistry is a server

This launch must only take place once.

Naming service

The code to execute the registry is the `createRegistry()` method of the `java.rmi.registry.LocateRegistry` class.

This method expects a port number as a parameter.

The naming service offers several services: `bind()`, `rebind()`, `unbind()`, `list()`, `lookup()`

It can be retrieved by calling the static method : `Registry LocateRegistry.getRegistry([machine],[port]);`

Naming service

For security reasons, the `bind()`, `rebind()` and `unbind()` methods are only accessible from the same machine

The `bind()` and `rebind()` commands allow the object to be saved in the directory

`rebind()` overwrites the name of an existing reference

Launching the naming service

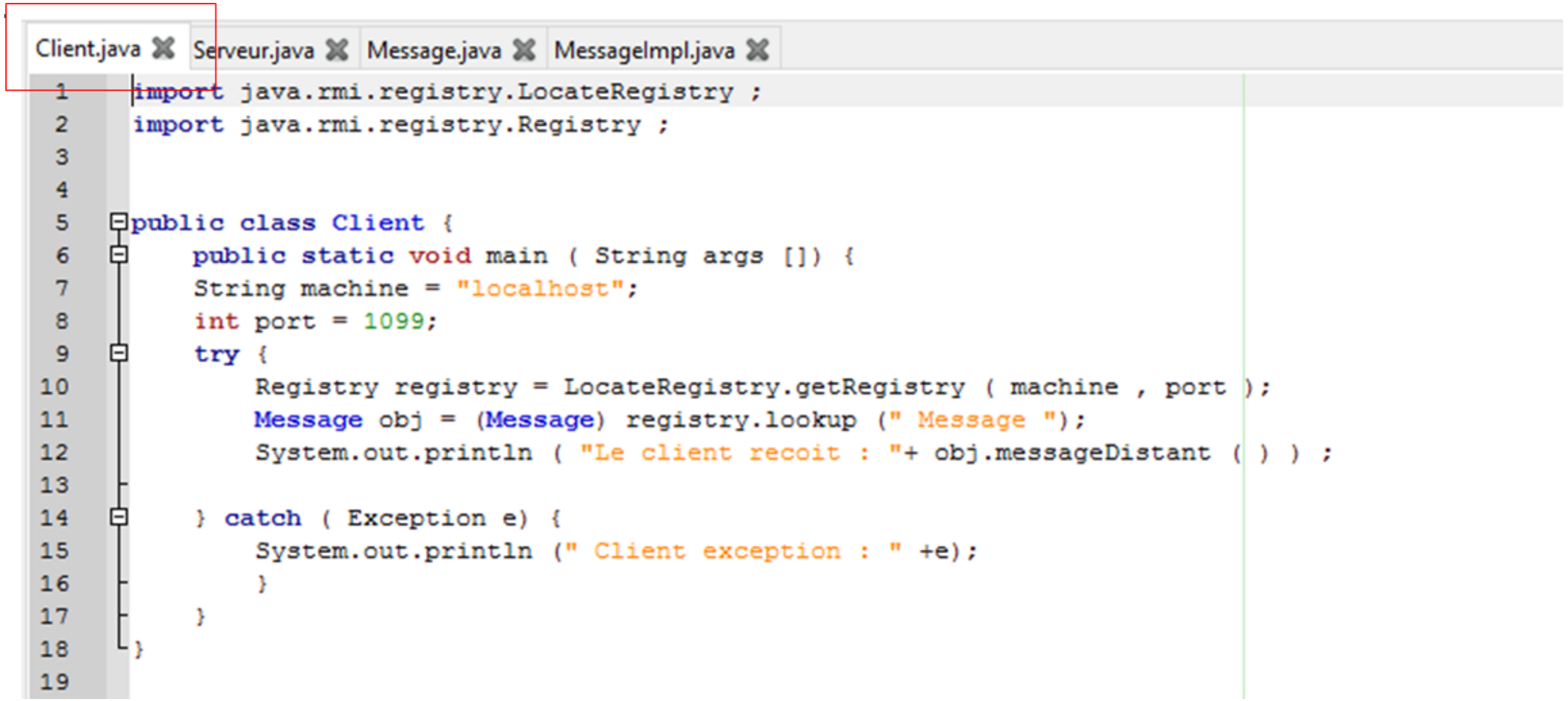
The `rmiregistry command` is supplied with the JDK. It must be run as a background task

Under Unix : `rmiregistry&`

Under Windows: `start rmiregistry`

The directory must have the stub code in its classpath, so it is usually launched at the root of the server code

Client example



```java
import java.rmi.registry.LocateRegistry ;
import java.rmi.registry.Registry ;


public class Client {
    public static void main ( String args []) {
    String machine = "localhost";
    int port = 1099;
    try {
        Registry registry = LocateRegistry.getRegistry ( machine , port );
        Message obj = (Message) registry.lookup (" Message ");
        System.out.println ( "Le client recoit : "+ obj.messageDistant ( ) ) ;

    } catch ( Exception e) {
        System.out.println (" Client exception : " +e);
        }
    }
}

```

Execution of the whole code

Launch the naming service

- The interface must be accessible to the name server via the classpath or codebase

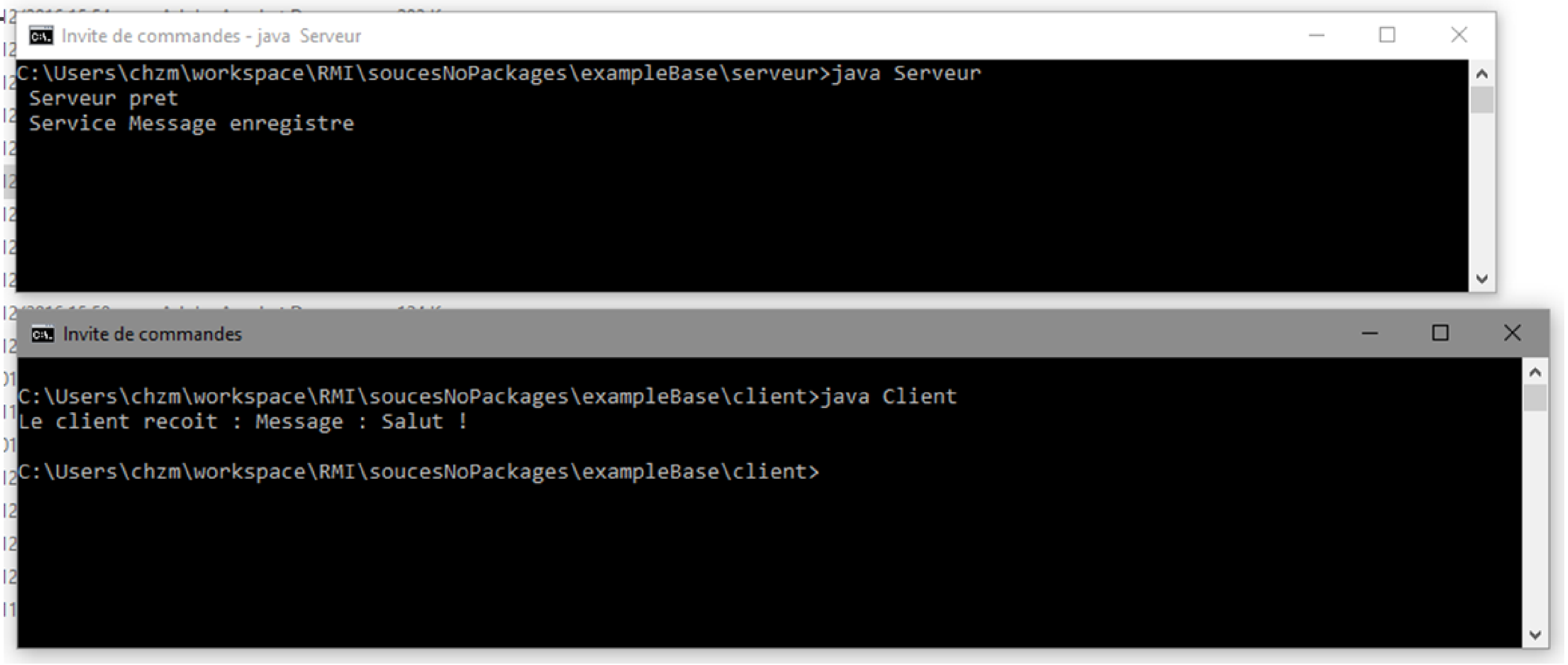
Launch the server

- The interface and implementation must be accessible via the classpath or codebase

Launch the client

- Access to the interface via classpath or codebase

Execution of the whole code



```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleBase\serveur>java Serveur
 Serveur pret
 Service Message enregistre
```

```

C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleBase\client>java Client
Le client recoit : Message : Salut !

C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleBase\client>
```

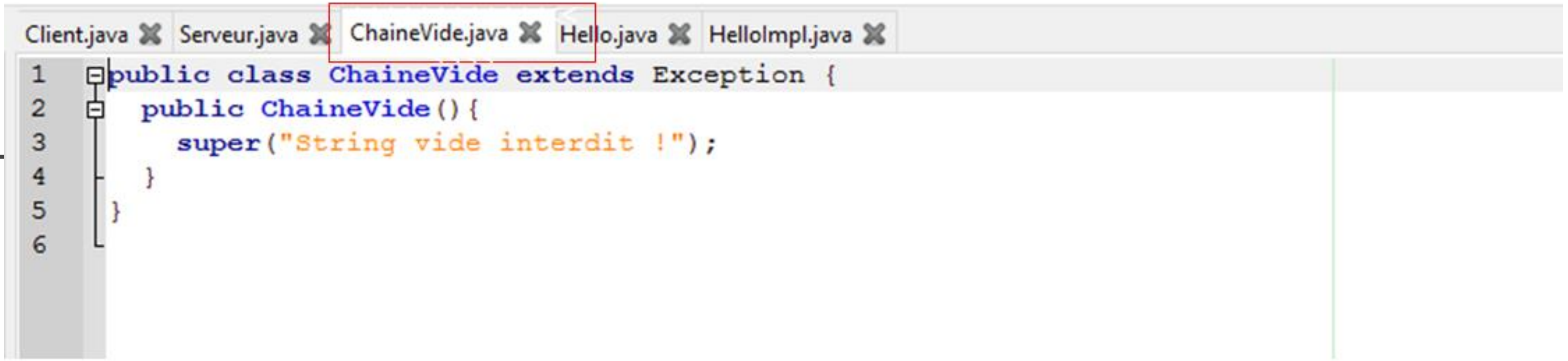
Exceptions

Exception management is transparent

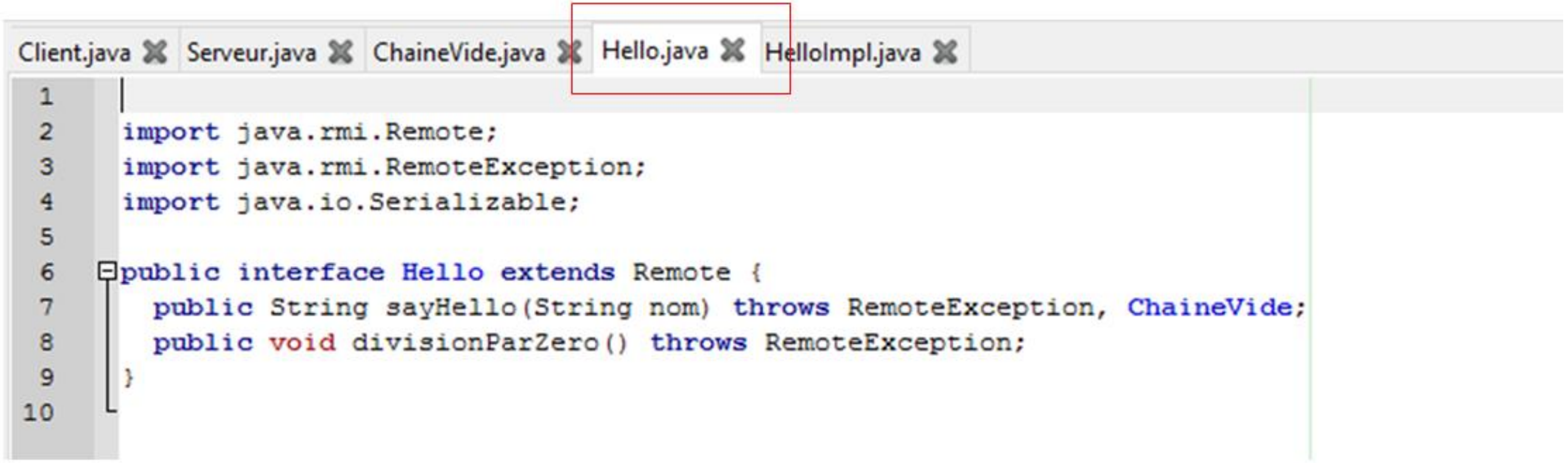
Existing exceptions are launched and transmitted identically to local processing

The creation of "specific" exceptions is identical to a local creation

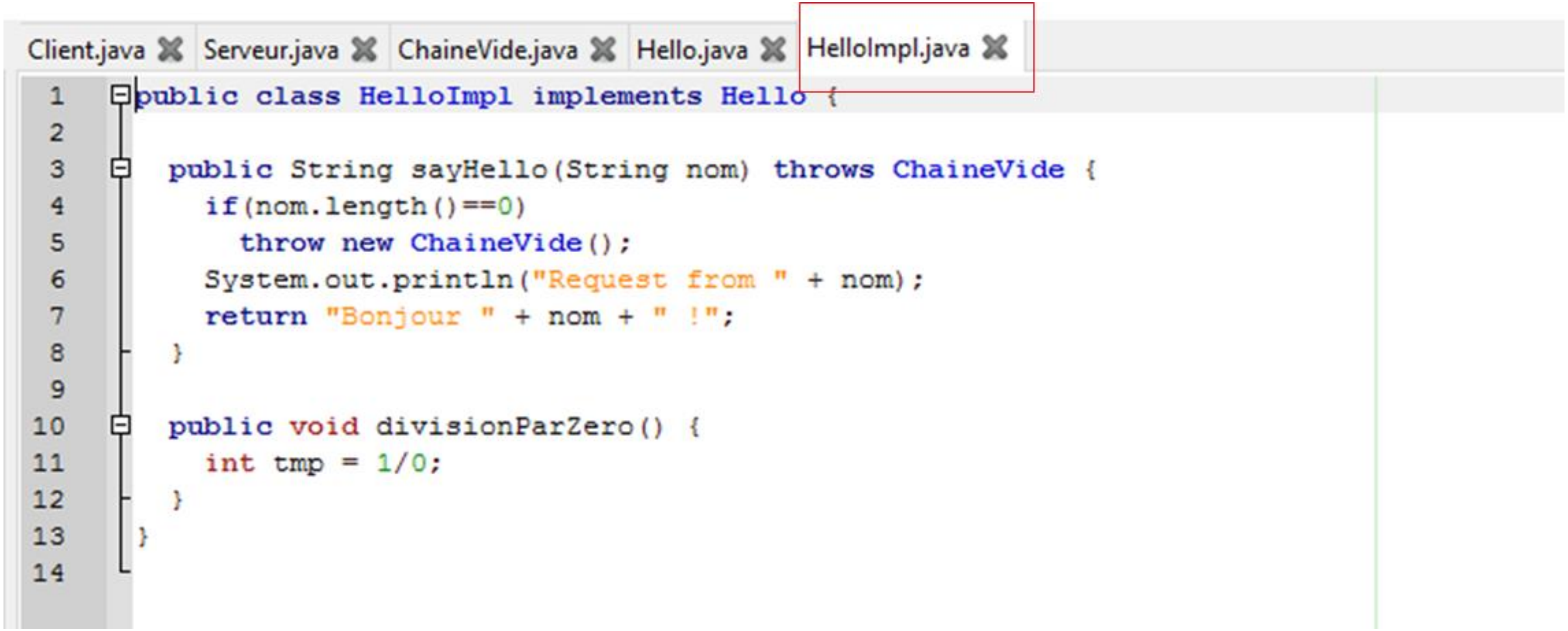
Any exception created must be accessible to the server, client and `rmiregistry` (via classpath or codebase).


Client.java | Serveur.java | ChaineVide.java | Hello.java | HelloImpl.java

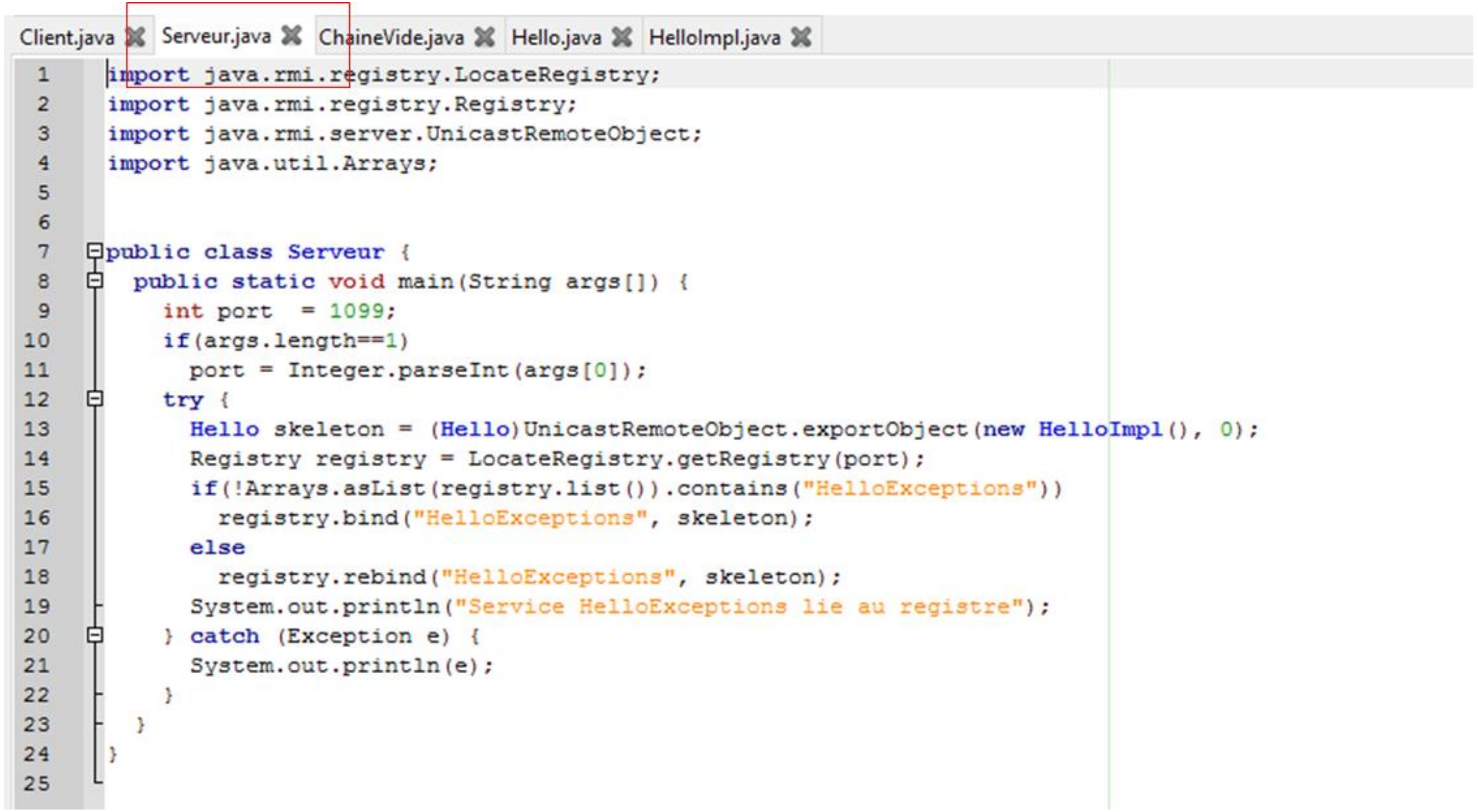
```java
public class ChaineVide extends Exception {
  public ChaineVide(){
    super("String vide interdit !");
  }
}

```


Client.java | Serveur.java | ChaineVide.java | Hello.java | HelloImpl.java

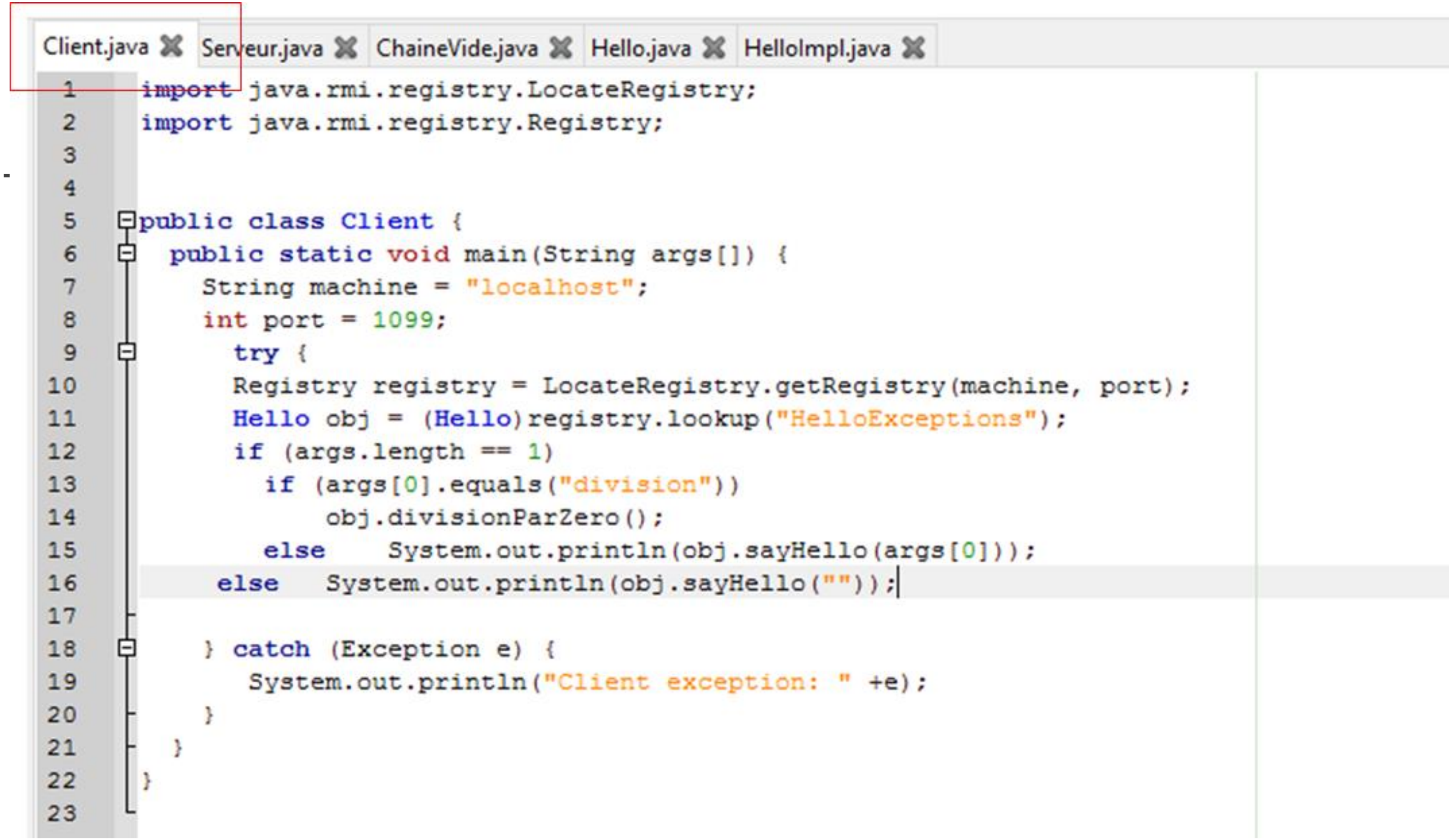
```java

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.io.Serializable;

public interface Hello extends Remote {
  public String sayHello(String nom) throws RemoteException, ChaineVide;
  public void divisionParZero() throws RemoteException;
}

```



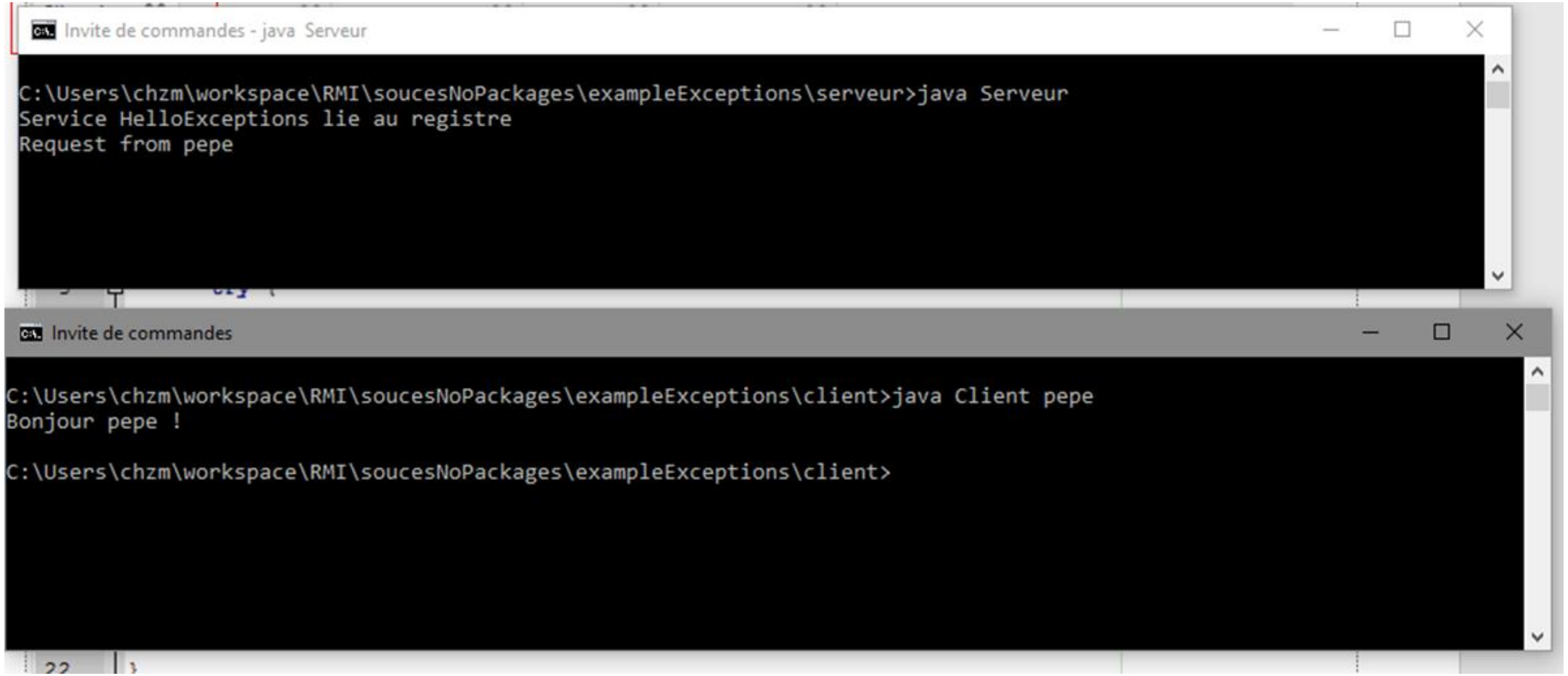
```java
public class HelloImpl implements Hello {

  public String sayHello(String nom) throws ChaineVide {
    if(nom.length()==0)
      throw new ChaineVide();
    System.out.println("Request from " + nom);
    return "Bonjour " + nom + " !";
  }

  public void divisionParZero() {
    int tmp = 1/0;
  }
}
```


Client.java Serveur.java ChaineVide.java Hello.java HelloImpl.java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.Arrays;


public class Serveur {
  public static void main(String args[]) {
    int port  = 1099;
    if(args.length==1)
      port = Integer.parseInt(args[0]);
    try {
      Hello skeleton = (Hello)UnicastRemoteObject.exportObject(new HelloImpl(), 0);
      Registry registry = LocateRegistry.getRegistry(port);
      if(!Arrays.asList(registry.list()).contains("HelloExceptions"))
        registry.bind("HelloExceptions", skeleton);
      else
        registry.rebind("HelloExceptions", skeleton);
      System.out.println("Service HelloExceptions lie au registre");
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```



```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;


public class Client {
  public static void main(String args[]) {
    String machine = "localhost";
    int port = 1099;
      try {
      Registry registry = LocateRegistry.getRegistry(machine, port);
      Hello obj = (Hello)registry.lookup("HelloExceptions");
      if (args.length == 1)
        if (args[0].equals("division"))
            obj.divisionParZero();
        else    System.out.println(obj.sayHello(args[0]));
     else    System.out.println(obj.sayHello(""));

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

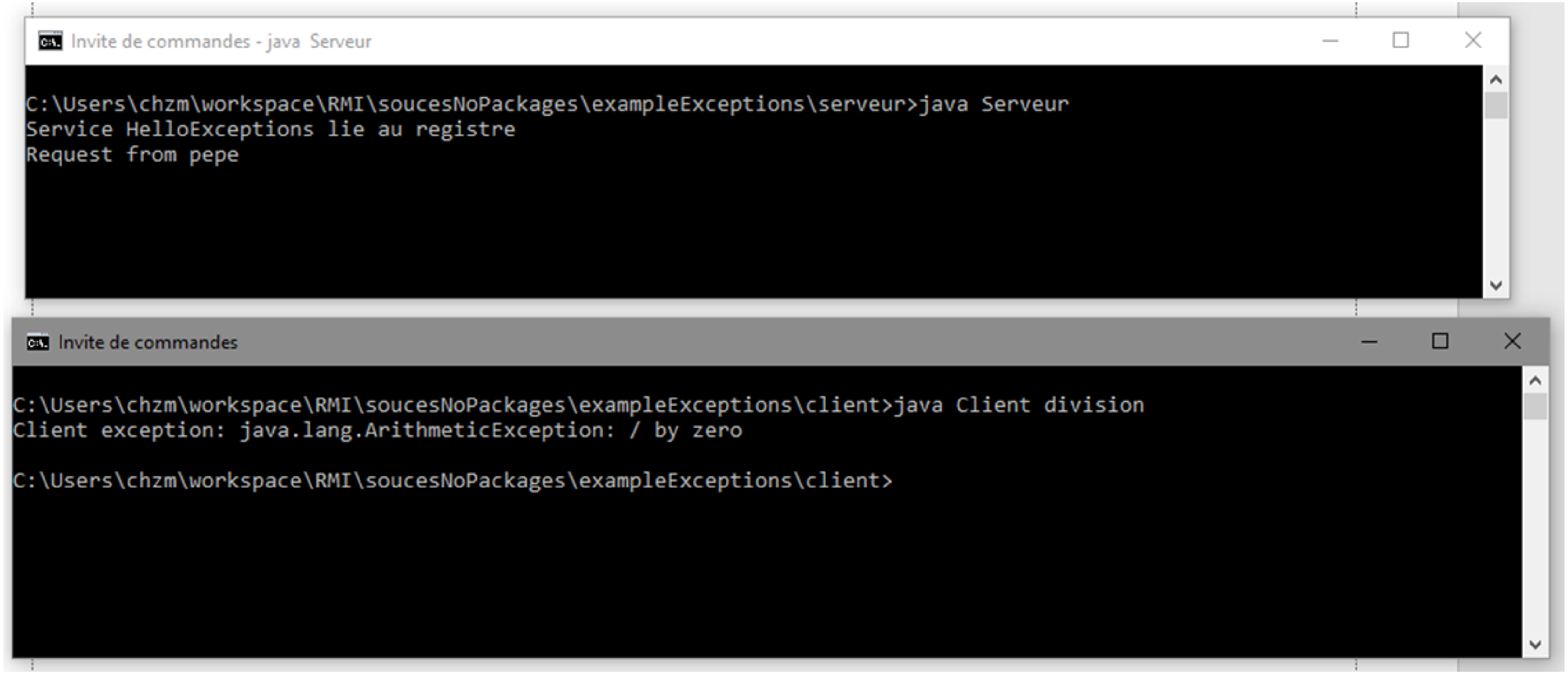


Invite de commandes - java Serveur

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\serveur>java Serveur
Service HelloExceptions lie au registre
Request from pepe
```

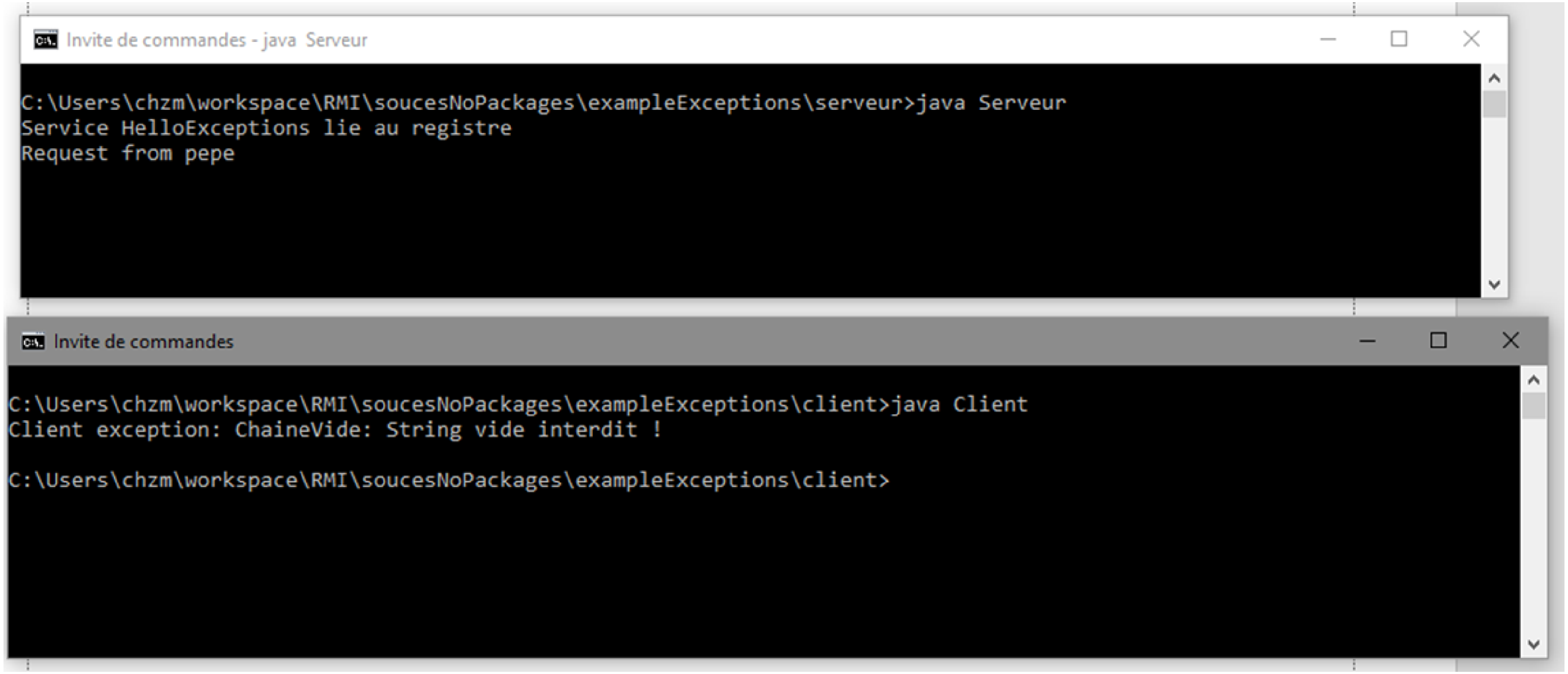
Invite de commandes

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\client>java Client pepe
Bonjour pepe !

C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\client>
```



```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\serveur>java Serveur
Service HelloExceptions lie au registre
Request from pepe
```

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\client>java Client division
Client exception: java.lang.ArithmeticException: / by zero

C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\client>
```


Invite de commandes - java Serveur

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\serveur>java Serveur
Service HelloExceptions lie au registre
Request from pepe
```

Invite de commandes

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\client>java Client
Client exception: ChaineVide: String vide interdit !

C:\Users\chzm\workspace\RMI\soucesNoPackages\exampleExceptions\client>
```

Parameter passing

In RMI, it is possible to pass arguments in the two classical ways (by value or by reference)

All remote function arguments must be

either distant objects,

or serialized

Parameter passing

The purpose of serialization is to transfer an object to a binary support.

For primitive types, this is done in a canonical way (integers, strings...)

For structured types, Java defines a mechanism called **serialization** to link and write an object to and from a recursive binary representation.

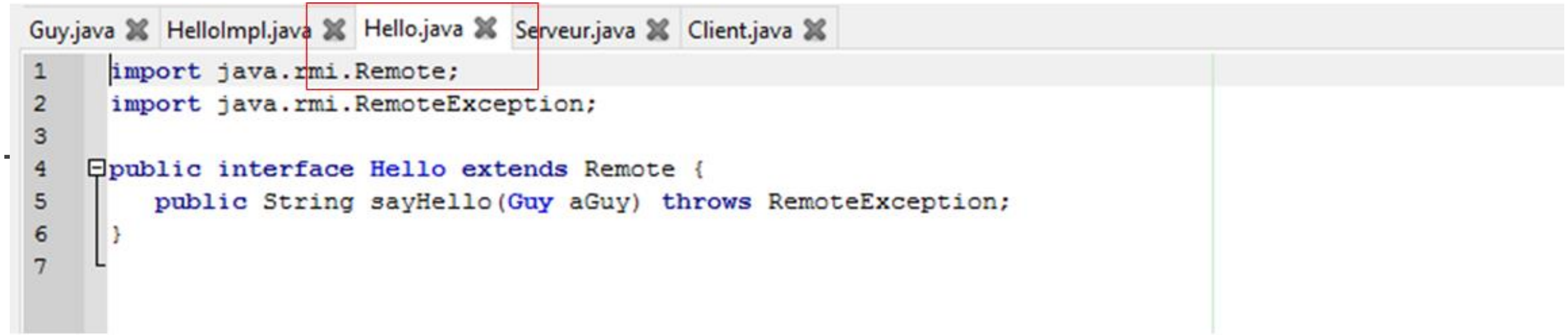
Passing a `Serializable` object

The passing of a `Serializable` object is done in a transparent way

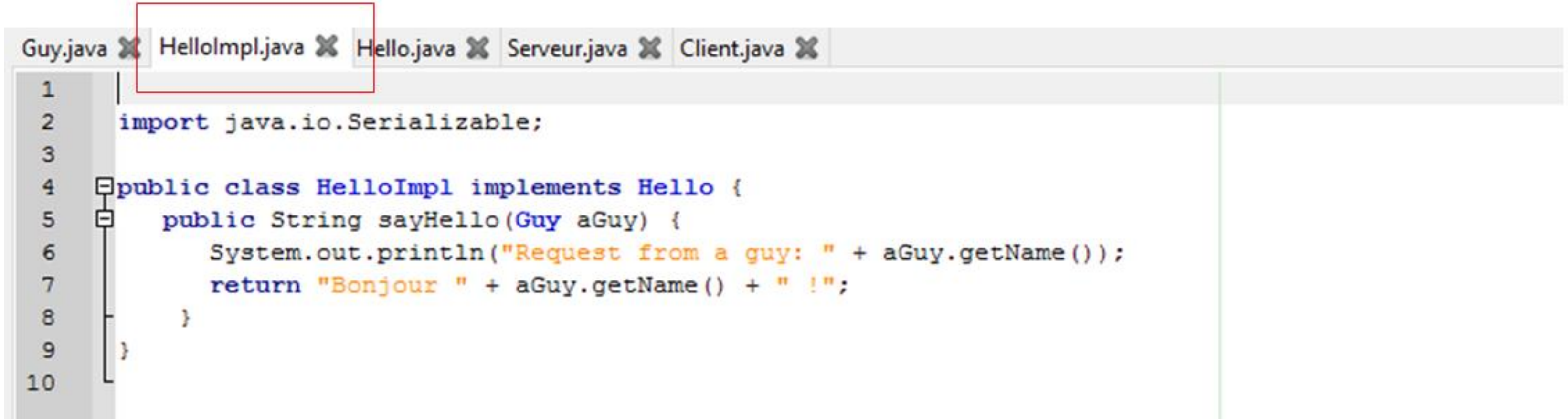
- Identical to a basic type passage
- It is a passing by value for an object

Any object not Serializable will not be able to be passed as a method parameter in RMI

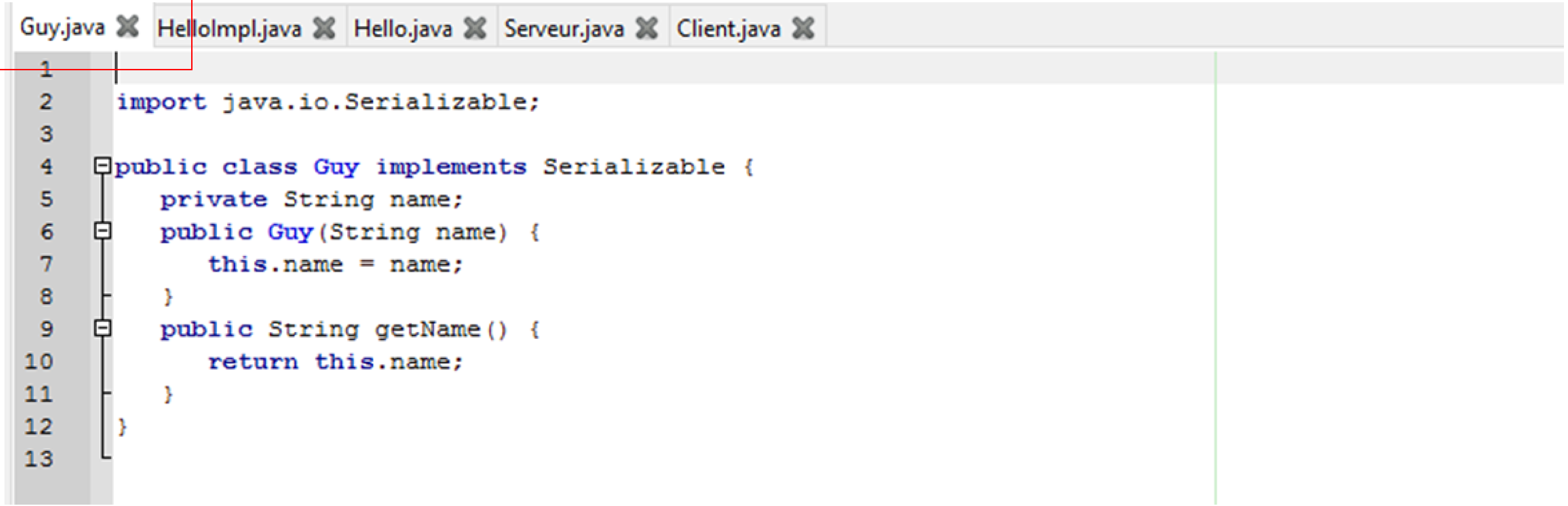
At runtime, the `rmiregistry` (and obviously the server and the client) must have access to the class to serialize via the classpath or the codebase



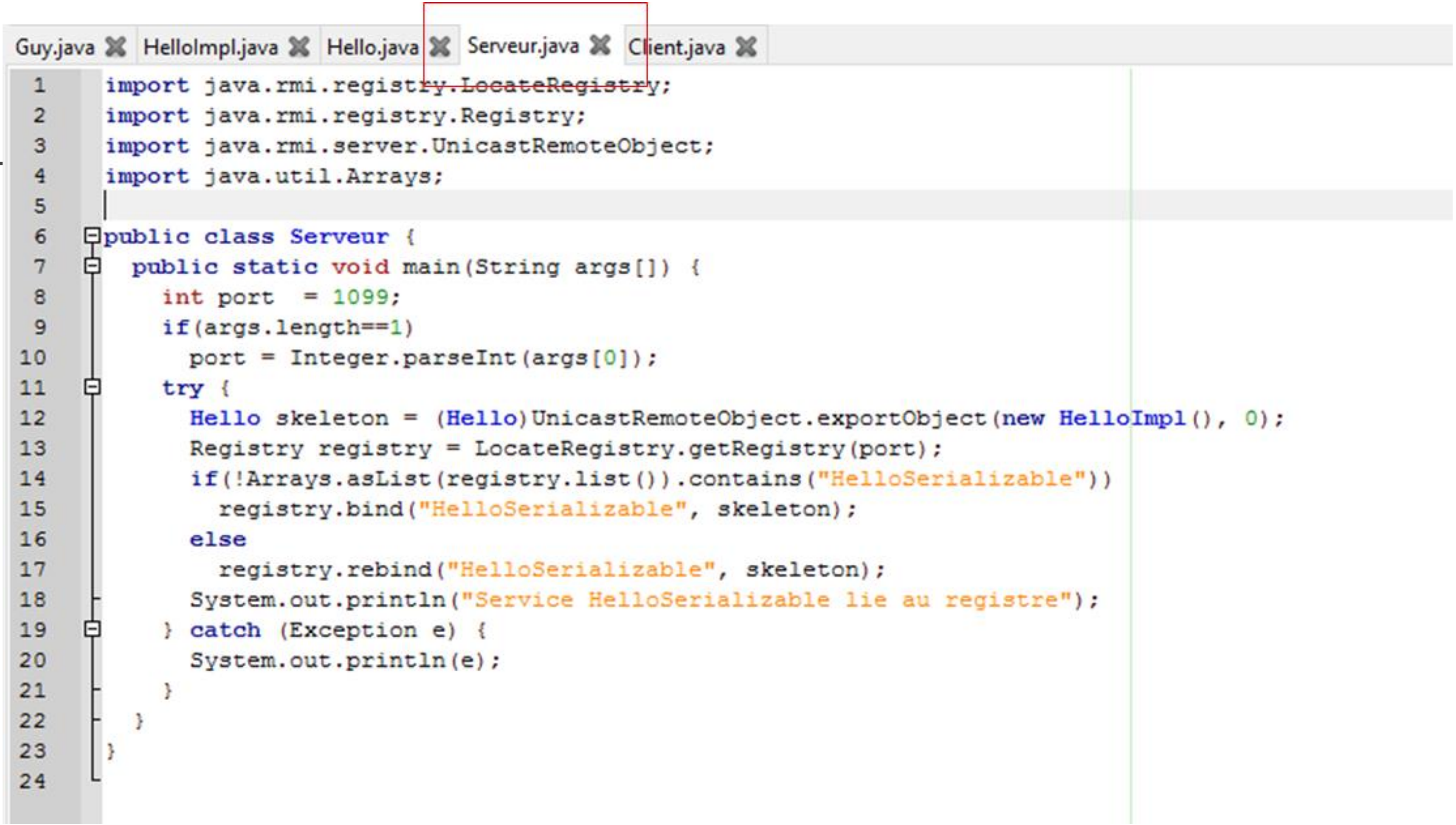
```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    public String sayHello(Guy aGuy) throws RemoteException;
}

```



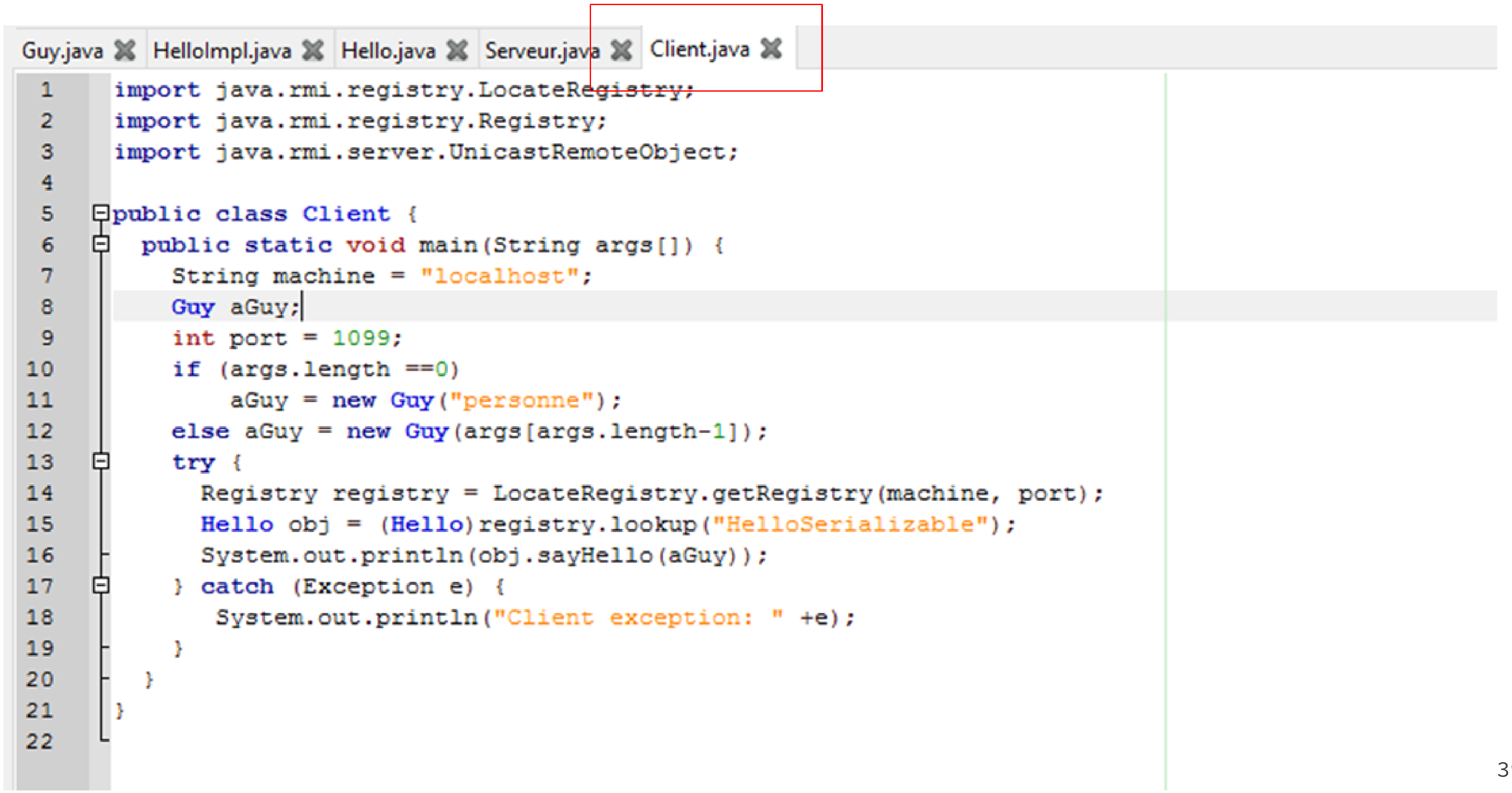
```java

import java.io.Serializable;

public class HelloImpl implements Hello {
    public String sayHello(Guy aGuy) {
        System.out.println("Request from a guy: " + aGuy.getName());
        return "Bonjour " + aGuy.getName() + " !";
    }
}

```



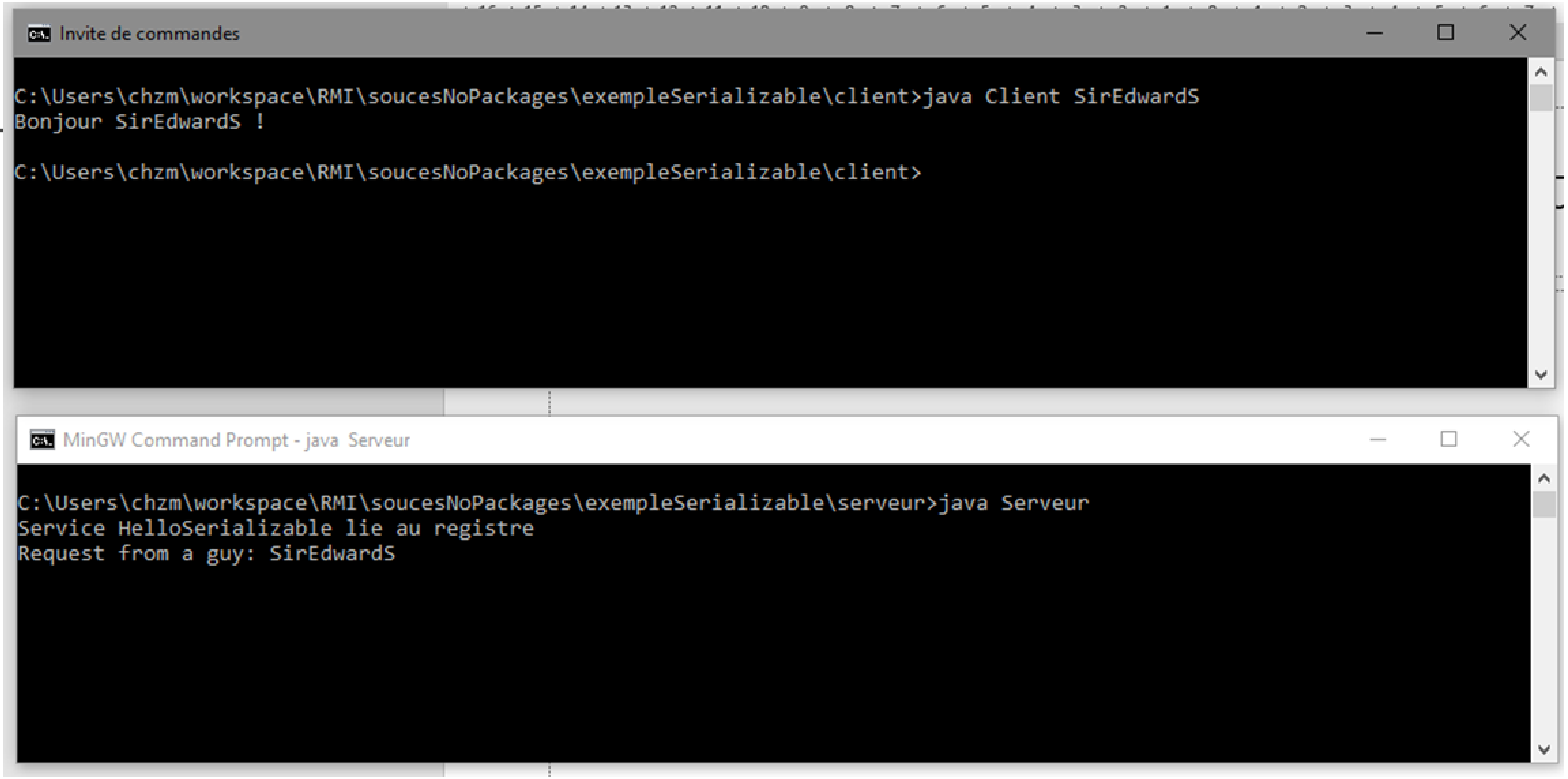
```java

import java.io.Serializable;

public class Guy implements Serializable {
    private String name;
    public Guy(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}

```


Guy.java HelloImpl.java Hello.java Serveur.java Client.java

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.Arrays;

public class Serveur {
  public static void main(String args[]) {
    int port  = 1099;
    if(args.length==1)
      port = Integer.parseInt(args[0]);
    try {
      Hello skeleton = (Hello)UnicastRemoteObject.exportObject(new HelloImpl(), 0);
      Registry registry = LocateRegistry.getRegistry(port);
      if(!Arrays.asList(registry.list()).contains("HelloSerializable"))
        registry.bind("HelloSerializable", skeleton);
      else
        registry.rebind("HelloSerializable", skeleton);
      System.out.println("Service HelloSerializable lie au registre");
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```



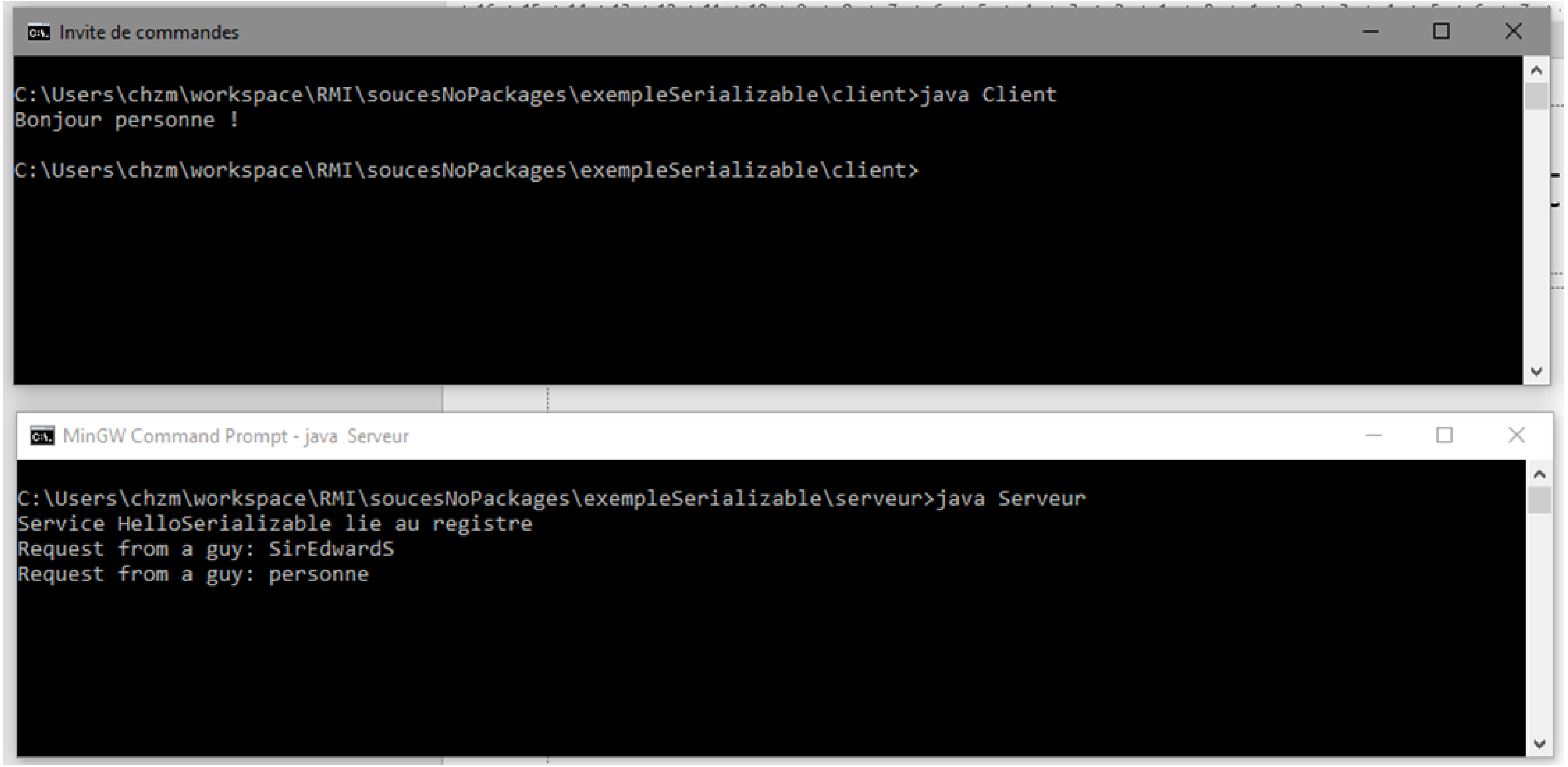
```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class Client {
  public static void main(String args[]) {
    String machine = "localhost";
    Guy aGuy;
    int port = 1099;
    if (args.length ==0)
        aGuy = new Guy("personne");
    else aGuy = new Guy(args[args.length-1]);
    try {
      Registry registry = LocateRegistry.getRegistry(machine, port);
      Hello obj = (Hello)registry.lookup("HelloSerializable");
      System.out.println(obj.sayHello(aGuy));
    } catch (Exception e) {
       System.out.println("Client exception: " +e);
    }
  }
}
```


Invite de commandes

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleSerializable\client>java Client SirEdwardS
Bonjour SirEdwardS !

C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleSerializable\client>
```

MinGW Command Prompt - java Serveur

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleSerializable\serveur>java Serveur
Service HelloSerializable lie au registre
Request from a guy: SirEdwardS
```



Invite de commandes

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleSerializable\client>java Client
Bonjour personne !

C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleSerializable\client>
```

MinGW Command Prompt - java Serveur

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleSerializable\serveur>java Serveur
Service HelloSerializable lie au registre
Request from a guy: SirEdwardS
Request from a guy: personne
```

Multi-threaded RMI

RMI server objects can reply to multiple clients simultaneously

- Because of their implementation, RMI methods exposed as `Remote` will be in multithreaded mode

To manage simultaneous accesses, we must beware of concurrent accesses to the attributes

To manage multiple threads the server can

- Coordinate them
- Give them priorities

Multi-threaded RMI

The server needs to lock concurrent access to attributes from remote methods

The client can synchronize the call to the remote object on its side, using `synchronized` methods

This synchronization will be local to its context, and not shared by other clients

- The lock will be placed on the stub and not on the remote object itself

Callbacks

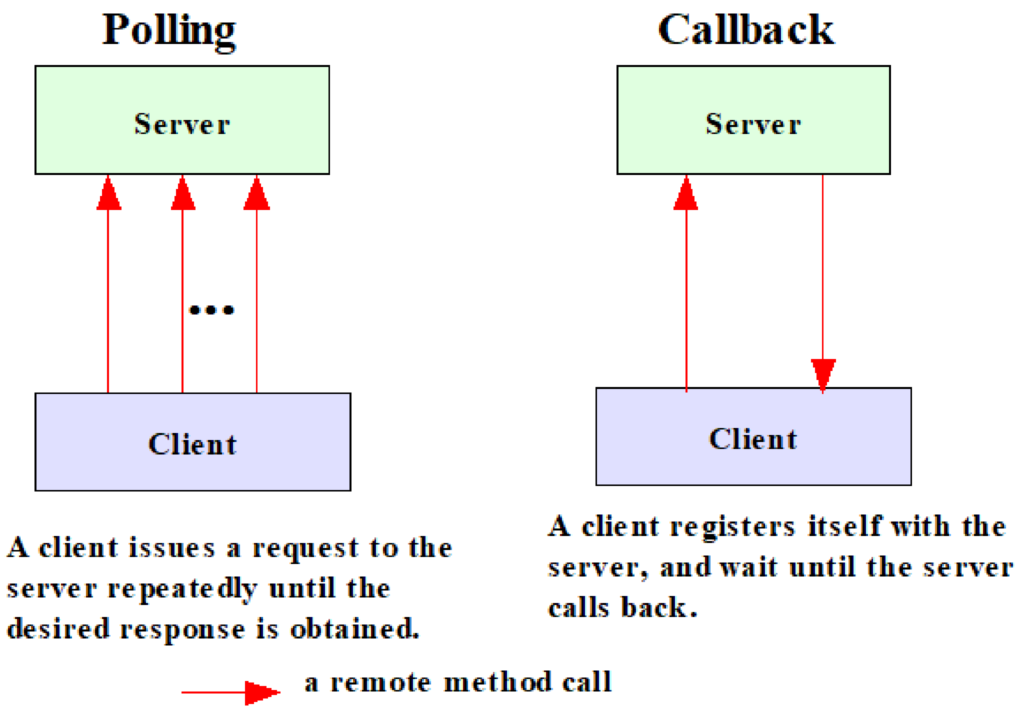
In the client-server model, the server is passive: the communication is initiated by the client; the server waits for the requests to arrive and provides the answers.

Some applications require the server to initiate communication during certain events, such as :

monitoring, games, auctions, elections / votes, chat, ….

Polling vs Callback

In the absence of ca   a
passive server sever   ıat
an event has occurre


Polling
Server
•••
Client
A client issues a request to the server repeatedly until the desired response is obtained.
Callback
Server
Client
A client registers itself with the server, and wait until the server calls back.
a remote method call

Definition

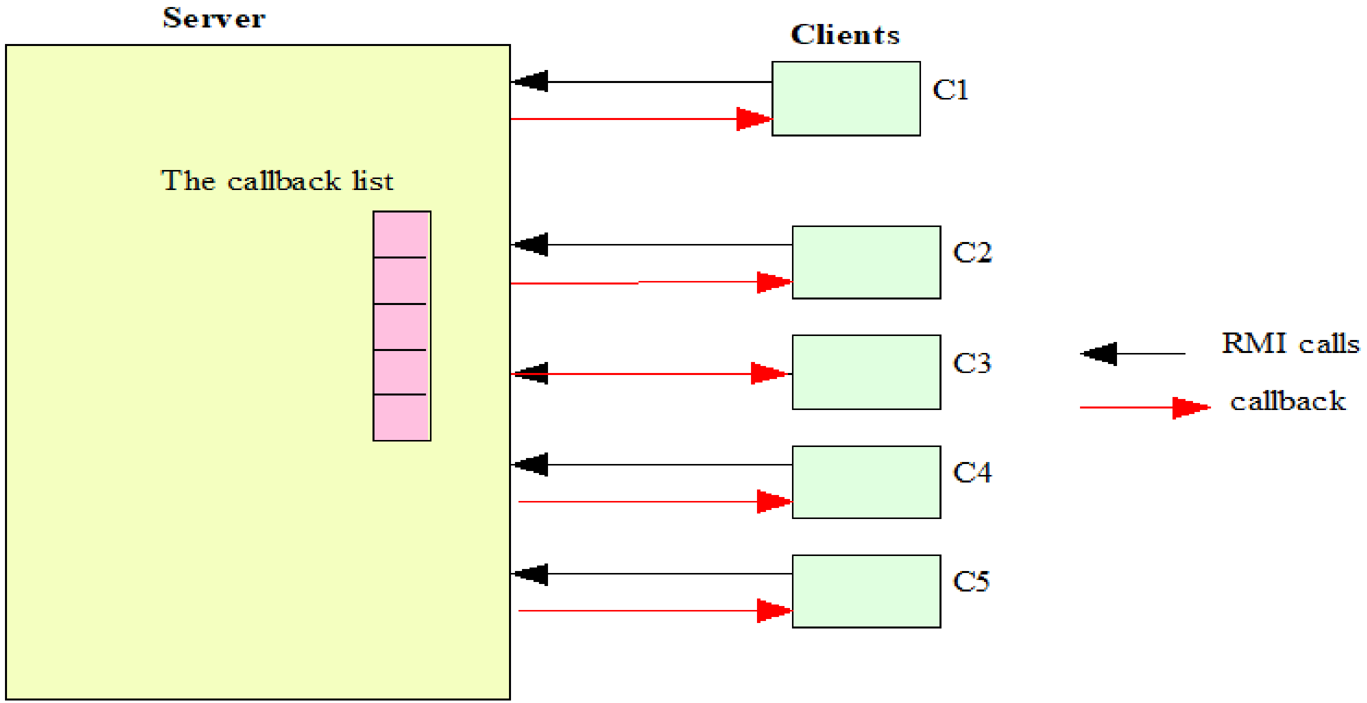
In distributed object-oriented systems, a **Callback** is a method called by the server (resp. the client) on an object transmitted as a parameter by the client (resp. the server) and requiring to be executed by the client (resp. the server).

The client, during its remote method call, passes as a parameter a reference on the object (a stub) that it proposes to the server. The latter can then invoke a method on this object to notify the client.

Setting up

Clients will re

The server will
occur.



Setting up

How can we notify a (remote) object of the occurrence of an event?

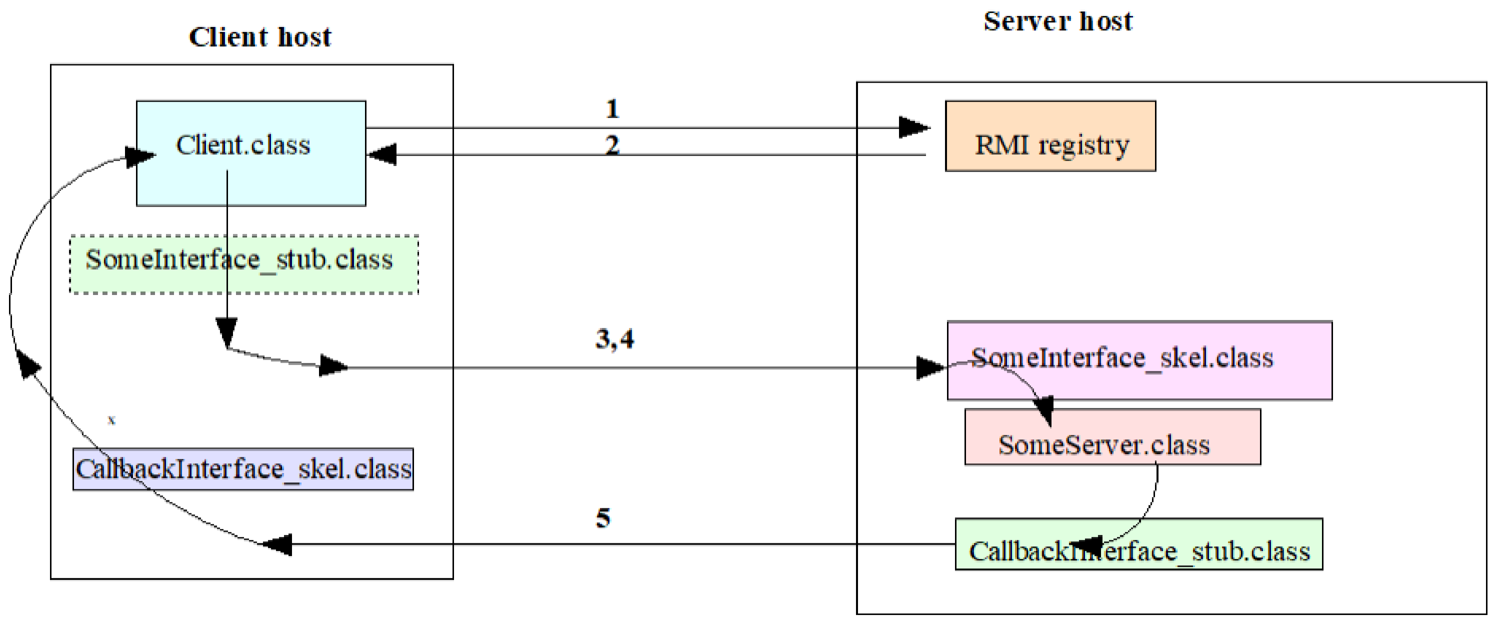
In fact, it is enough to pass the reference of the object to be called back to the server in charge of following the events. When the event occurs, the server will invoke the client notification method

Setting up

Thus, for each type of event, a specific interface is created (for the client who wants to be notified), and the potential clients to be notified must register with an implementation of this interface.

This implies that clients and servers are all in turn servers and clients.

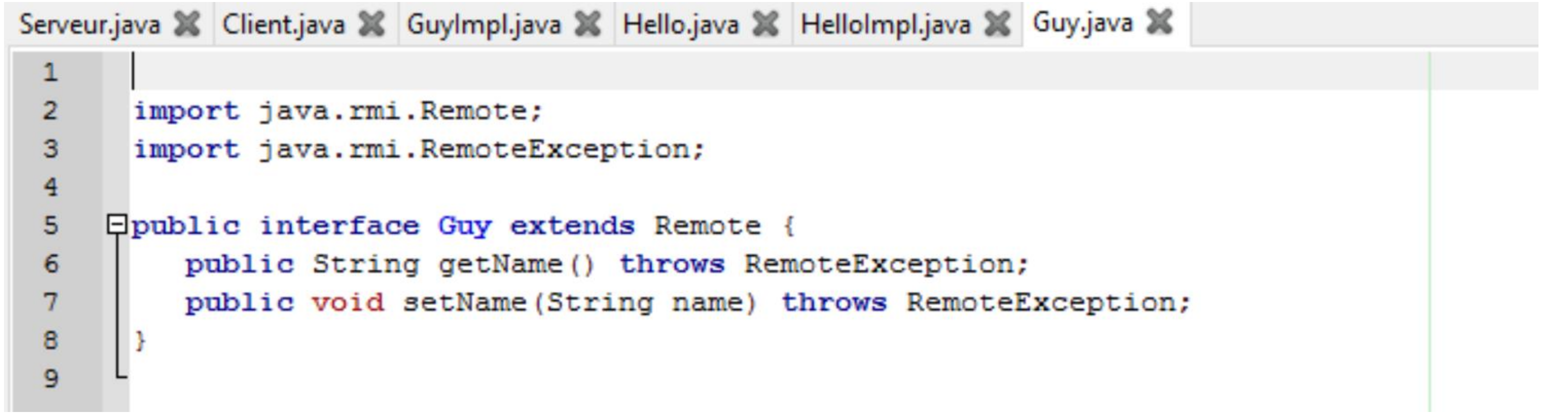
Setting up



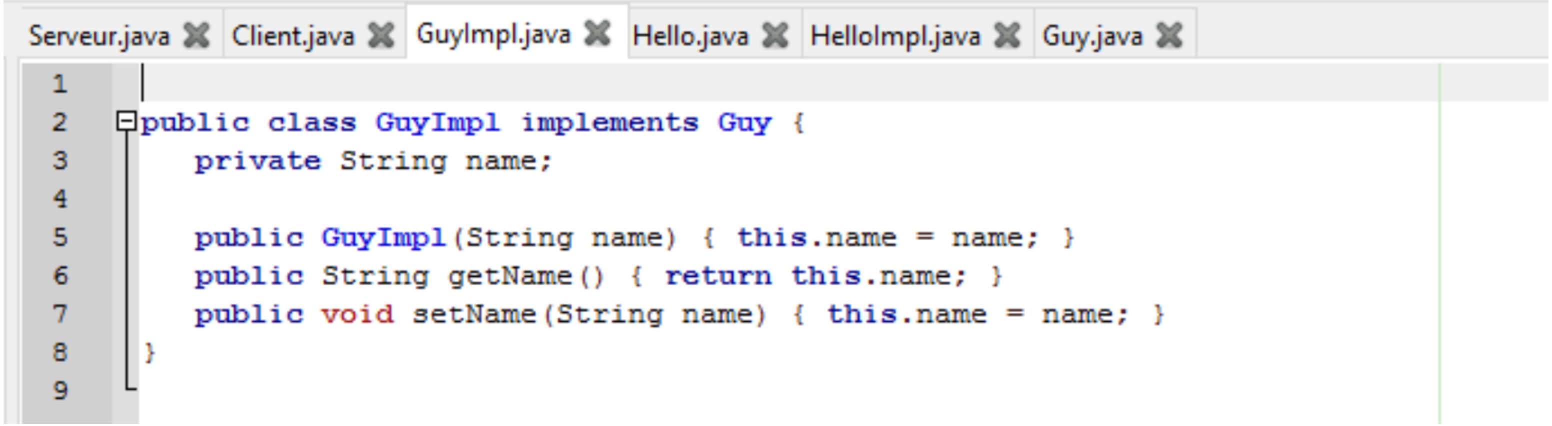
Client host
Server host
Client.class
SomeInterface_stub.class
CallbackInterface_skel.class
RMI registry
SomeInterface_skel.class
SomeServer.class
CallbackInterface_stub.class
1
2
3,4
5
x

An example

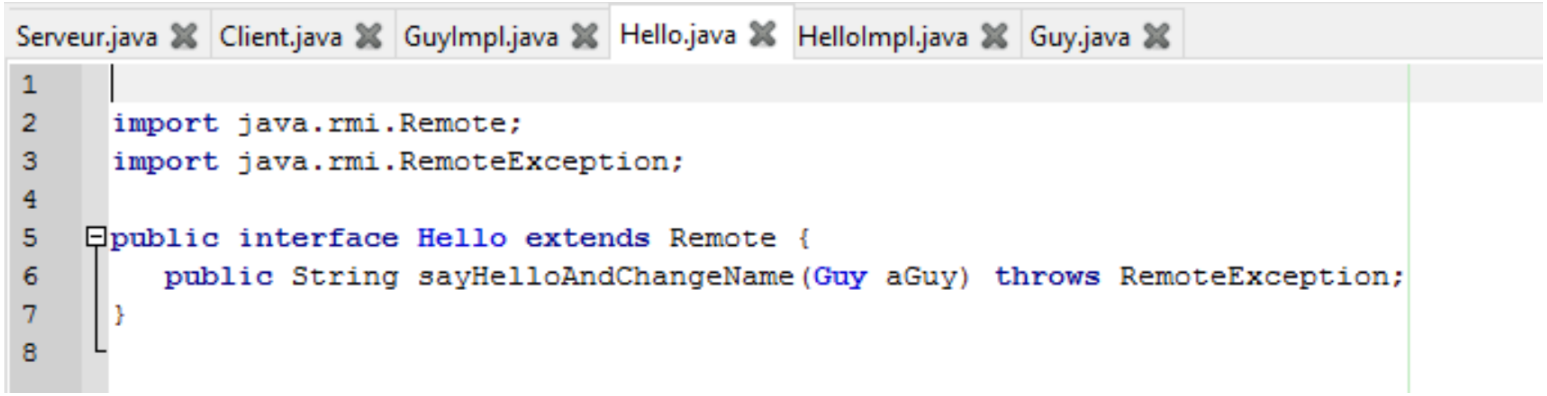
- The idea is to allow the server to call a client that has contacted it before:
  - Increase asynchronies: publish/subscribe schema:
    - customer calls the server with immediate return (subscribe)
    - server calls the client back when the service is executed (publish)
  - Increase interactions: the server can ask the client for additional data
  - Event programming



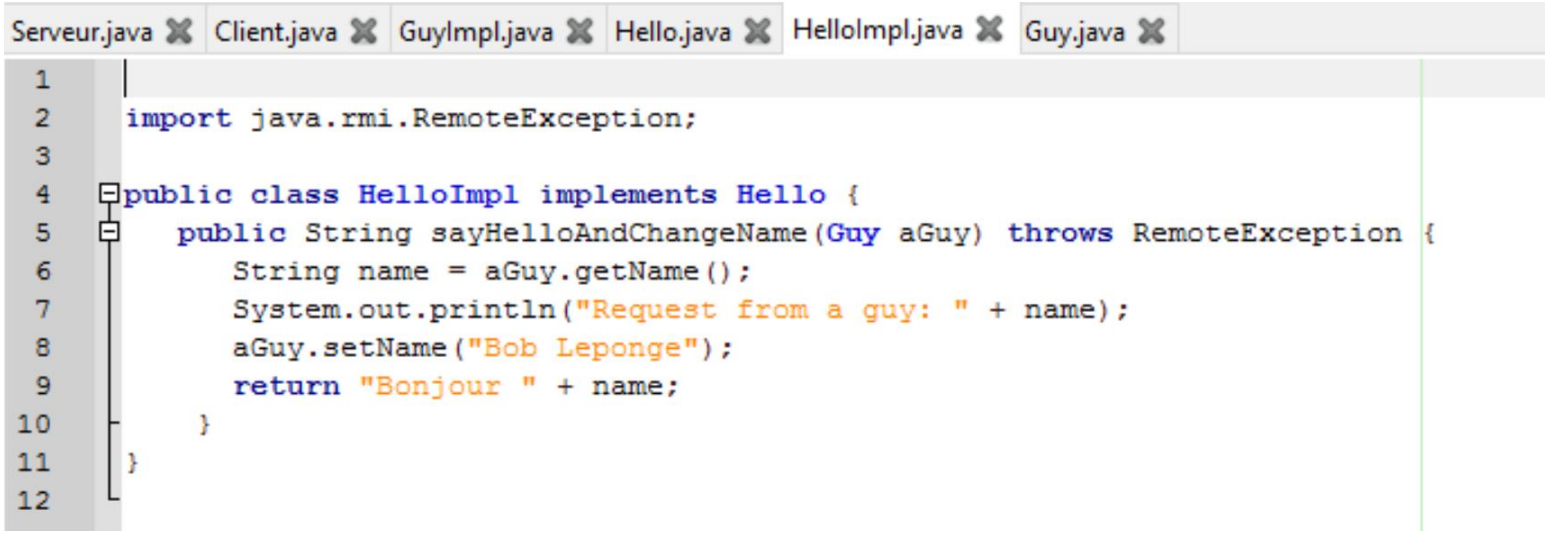
```java

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Guy extends Remote {
   public String getName() throws RemoteException;
   public void setName(String name) throws RemoteException;
}

```



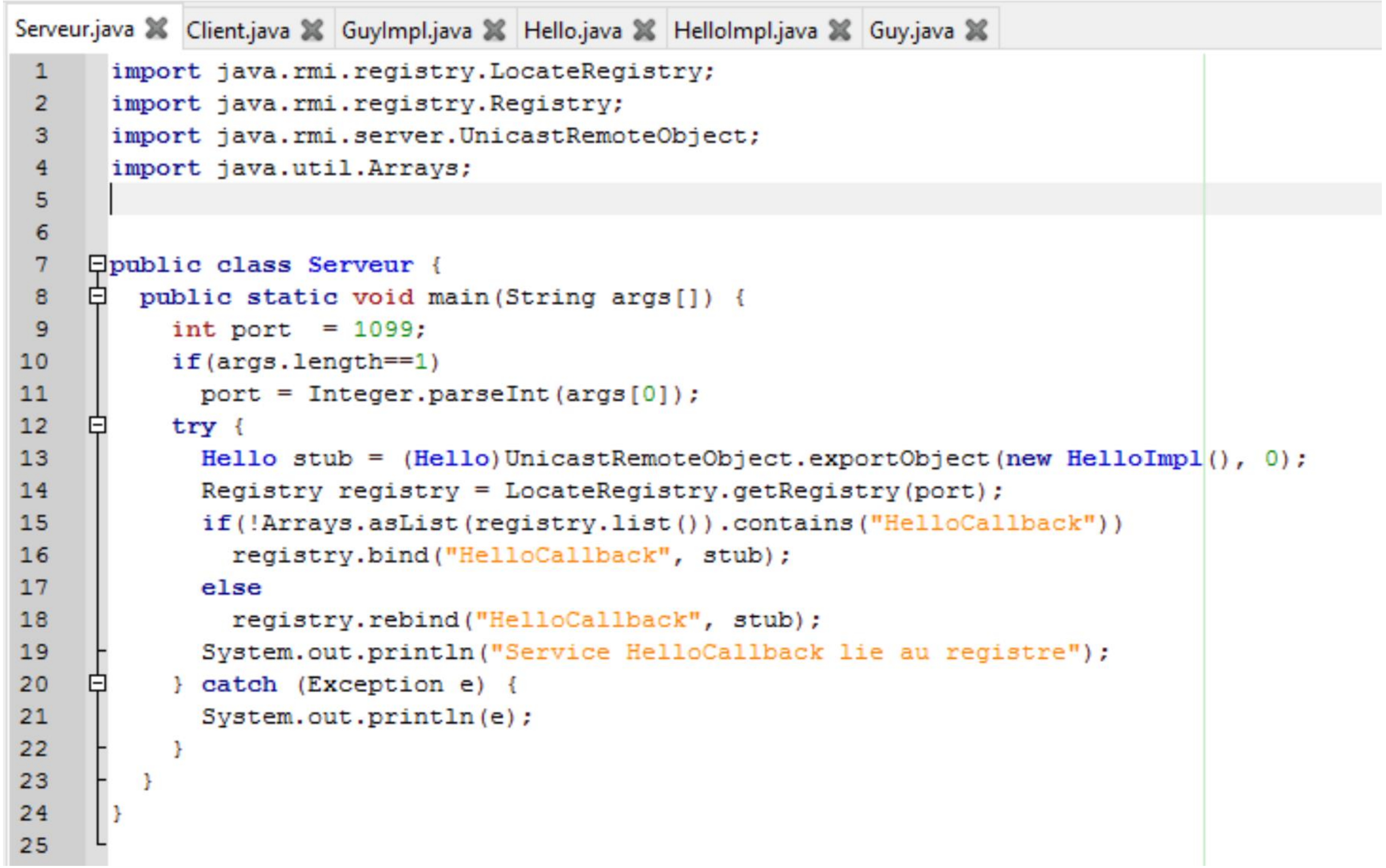
```java

public class GuyImpl implements Guy {
   private String name;

   public GuyImpl(String name) { this.name = name; }
   public String getName() { return this.name; }
   public void setName(String name) { this.name = name; }
}

```



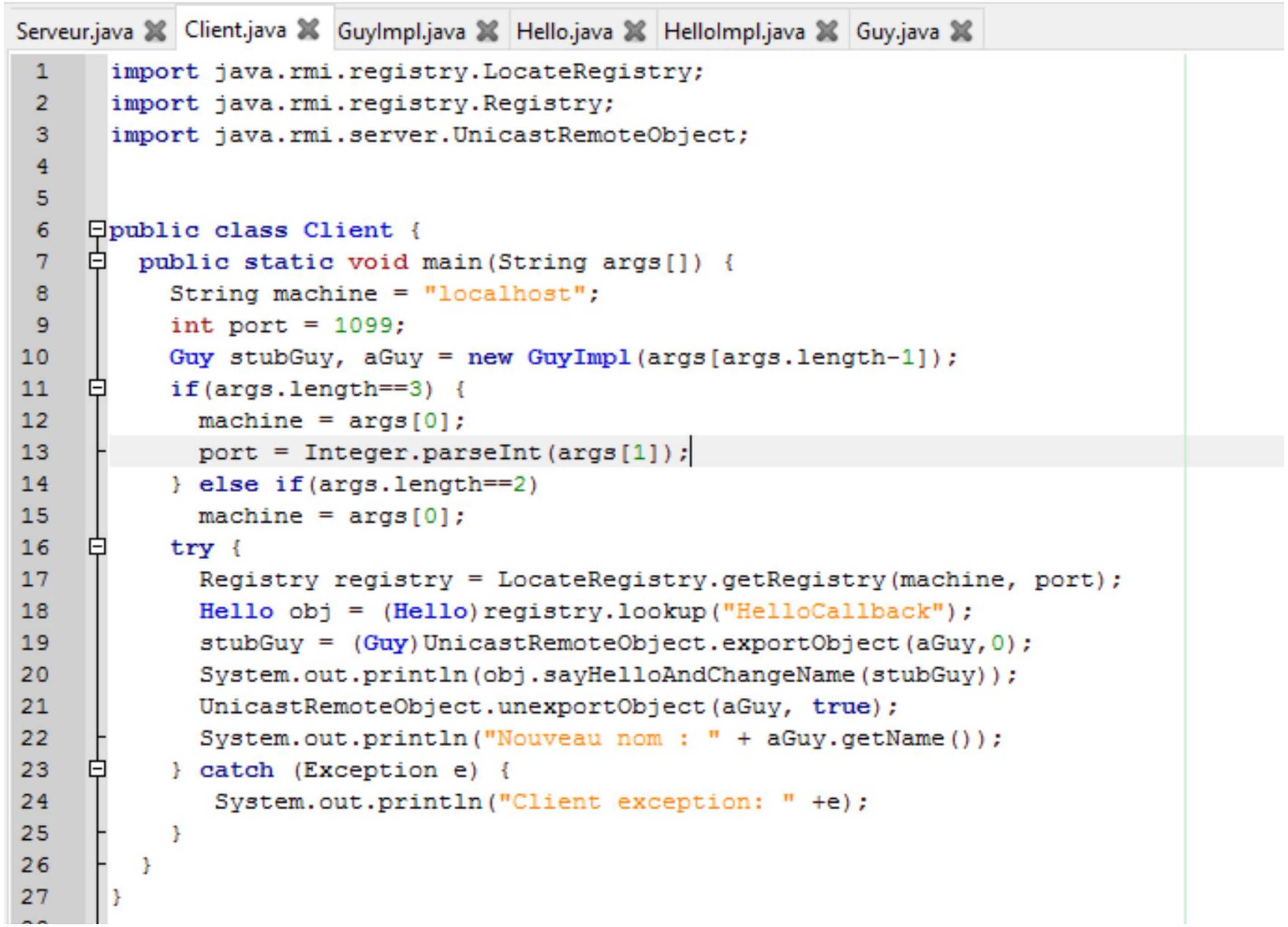
```java

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
   public String sayHelloAndChangeName(Guy aGuy) throws RemoteException;
}

```



```java

import java.rmi.RemoteException;

public class HelloImpl implements Hello {
   public String sayHelloAndChangeName(Guy aGuy) throws RemoteException {
      String name = aGuy.getName();
      System.out.println("Request from a guy: " + name);
      aGuy.setName("Bob Leponge");
      return "Bonjour " + name;
    }
}

```

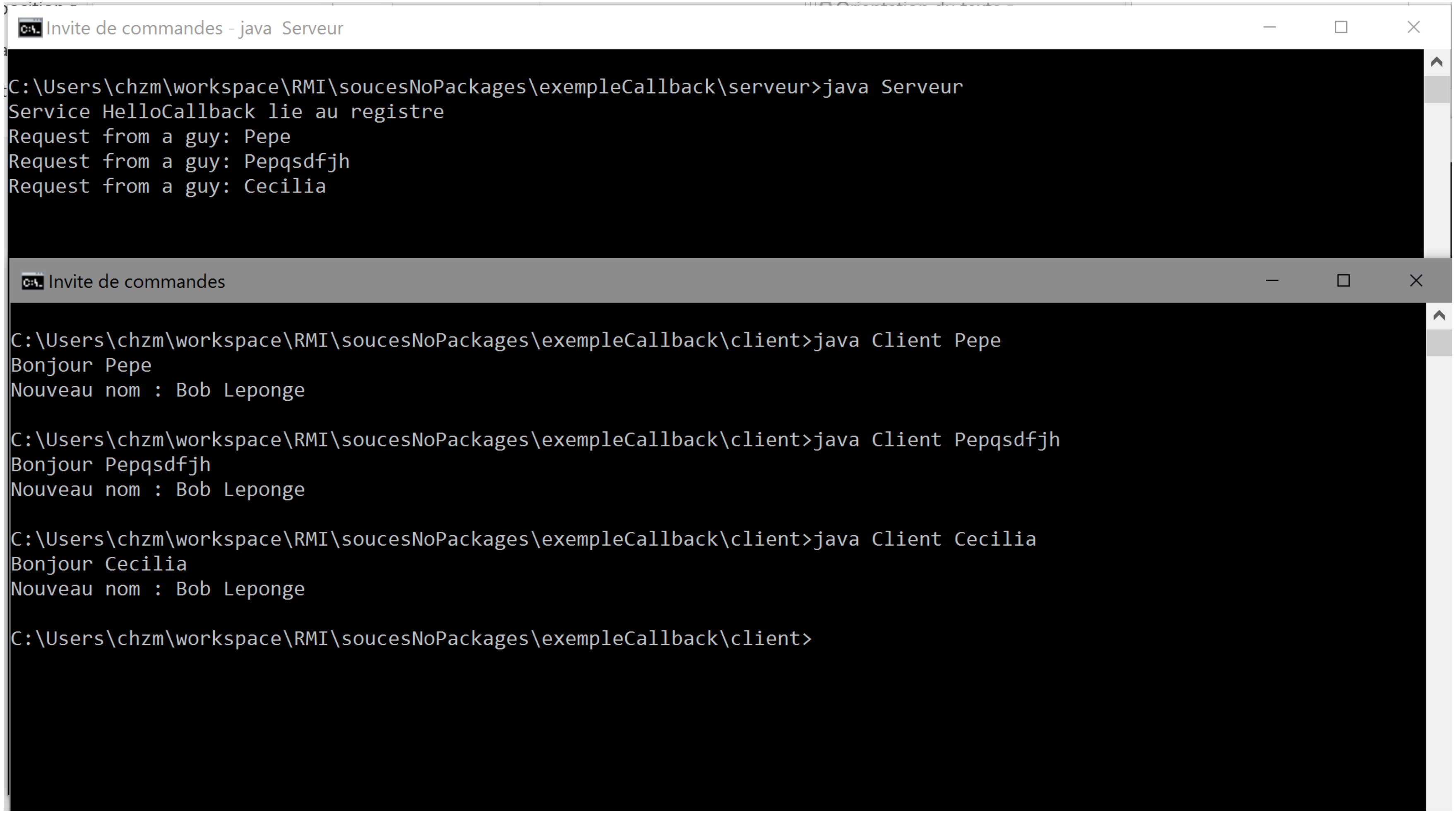


```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.Arrays;


public class Serveur {
  public static void main(String args[]) {
    int port  = 1099;
    if(args.length==1)
      port = Integer.parseInt(args[0]);
    try {
      Hello stub = (Hello)UnicastRemoteObject.exportObject(new HelloImpl(), 0);
      Registry registry = LocateRegistry.getRegistry(port);
      if(!Arrays.asList(registry.list()).contains("HelloCallback"))
        registry.bind("HelloCallback", stub);
      else
        registry.rebind("HelloCallback", stub);
      System.out.println("Service HelloCallback lie au registre");
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```



Serveur.java | Client.java | GuyImpl.java | Hello.java | HelloImpl.java | Guy.java

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;


public class Client {
  public static void main(String args[]) {
    String machine = "localhost";
    int port = 1099;
    Guy stubGuy, aGuy = new GuyImpl(args[args.length-1]);
    if(args.length==3) {
      machine = args[0];
      port = Integer.parseInt(args[1]);
    } else if(args.length==2)
      machine = args[0];
    try {
      Registry registry = LocateRegistry.getRegistry(machine, port);
      Hello obj = (Hello)registry.lookup("HelloCallback");
      stubGuy = (Guy)UnicastRemoteObject.exportObject(aGuy,0);
      System.out.println(obj.sayHelloAndChangeName(stubGuy));
      UnicastRemoteObject.unexportObject(aGuy, true);
      System.out.println("Nouveau nom : " + aGuy.getName());
    } catch (Exception e) {
       System.out.println("Client exception: " +e);
    }
  }
}
```


Invite de commandes - java Serveur

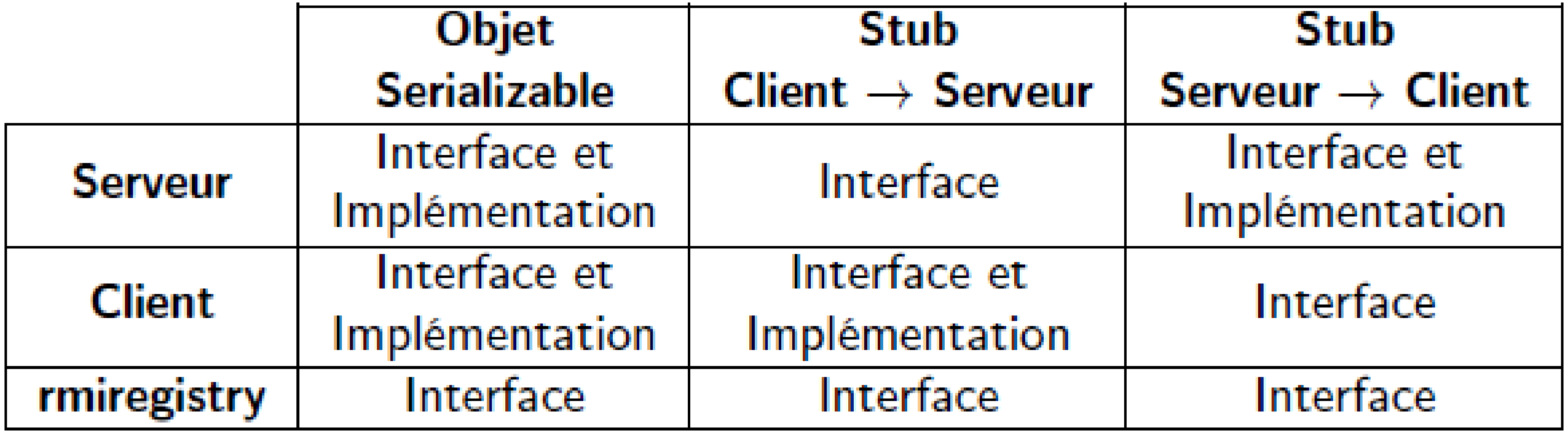
```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleCallback\serveur>java Serveur
Service HelloCallback lie au registre
Request from a guy: Pepe
Request from a guy: Pepqsdfjh
Request from a guy: Cecilia
```

Invite de commandes

```
C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleCallback\client>java Client Pepe
Bonjour Pepe
Nouveau nom : Bob Leponge

C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleCallback\client>java Client Pepqsdfjh
Bonjour Pepqsdfjh
Nouveau nom : Bob Leponge

C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleCallback\client>java Client Cecilia
Bonjour Cecilia
Nouveau nom : Bob Leponge

C:\Users\chzm\workspace\RMI\soucesNoPackages\exempleCallback\client>
```

Summary on Callbacks

- for an object passed as a parameter



	Objet Serializable	**Stub Client → Serveur**	**Stub Serveur → Client**
Serveur	Interface et Implémentation	Interface	Interface et Implémentation
Client	Interface et Implémentation	Interface et Implémentation	Interface
rmiregistry	Interface	Interface	Interface

References on Callbacks

- https://docs.oracle.com/cd/E13211_01/wle/rmi/callbak.htm
- http://darwinsys.com/java/rmi/

Stub downloading

- RMI is designed to allow the dynamic provision of stubs to the client. This allows the modification of methods remotely without affecting the client program itself.
- The stub can be hosted on a web server and downloaded via HTTP or can also be dynamically loaded from a location defined as a parameter for the execution of the client
  - `-Djava.rmi.server.codebase=<chemin d’accès>`
- Security measures are required to protect the client and server. A java security policy file must be defined on the server host and also on the client host.
  - `-Djava.security.policy=<fichier policy>`
- A Java security manager must be instantiated in both client and server programs (see Eclipse tutorial)

Final Remarks

- RMI is a powerful object distribution mechanism
- Some points that have not been seen in this course:
    - Interoperability: RMI on IIOP (Internet Inter-Orb Protocol)
    - "Customization" of sockets (for example to encrypt/compress communication)
    - HTTP tunneling to make invocations of rmi methods through a firewall

TO DO **before** February 01, 2021

Do no forget to test your program with the server and the client on different machines!

1. Download and test all the source code presented in this chapter
2. If needed, follow the tutorial “Step by Step RMI with Eclipse

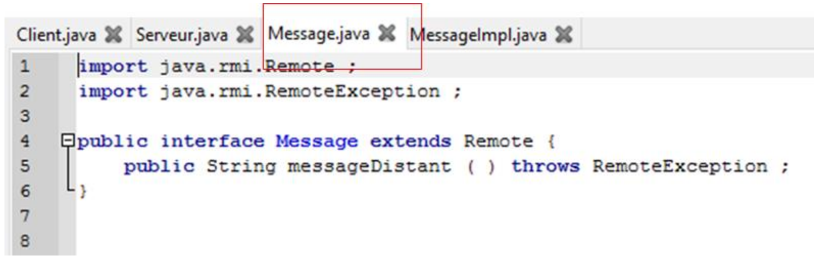
Please, remark the different syntax to declare remote objects in the classical examples and in the Eclipse examples. Both forms are acceptable

Several different forms to do things

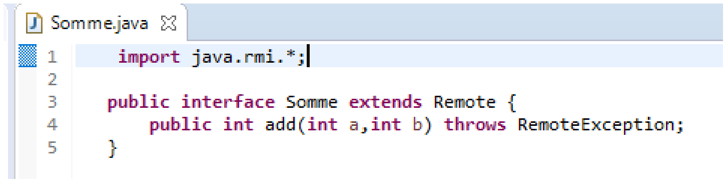
In the following:

To the left, the version we discussed in the chapter

To the right, the version proposed in the Eclipse tutorial



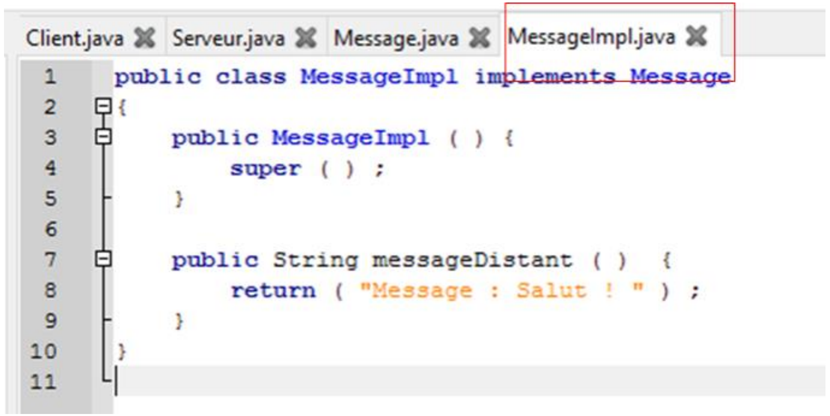
```
import java.rmi.Remote ;
import java.rmi.RemoteException ;

public interface Message extends Remote {
    public String messageDistant ( ) throws RemoteException ;
}
```



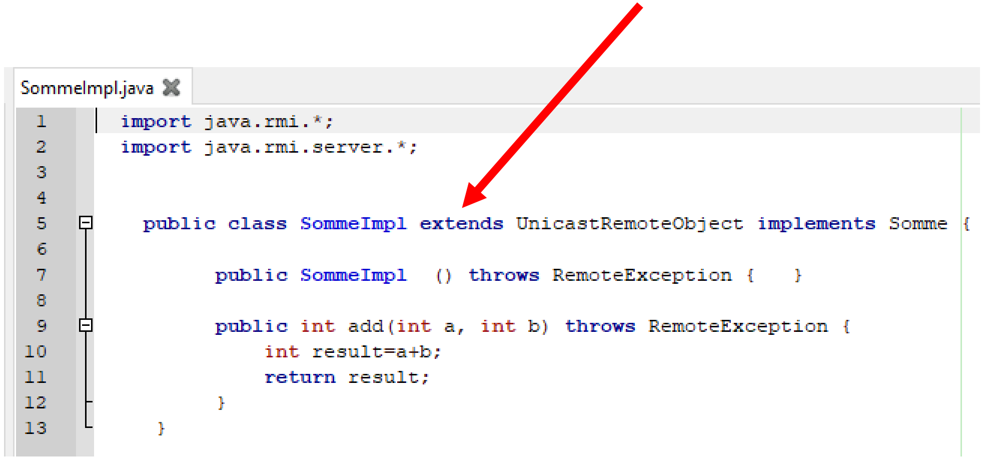
```
 import java.rmi.*;

public interface Somme extends Remote {
    public int add(int a,int b) throws RemoteException;
}
```

The **interface** always has the same aspect

Several different forms to do things


Client.java Serveur.java Message.java MessageImpl.java

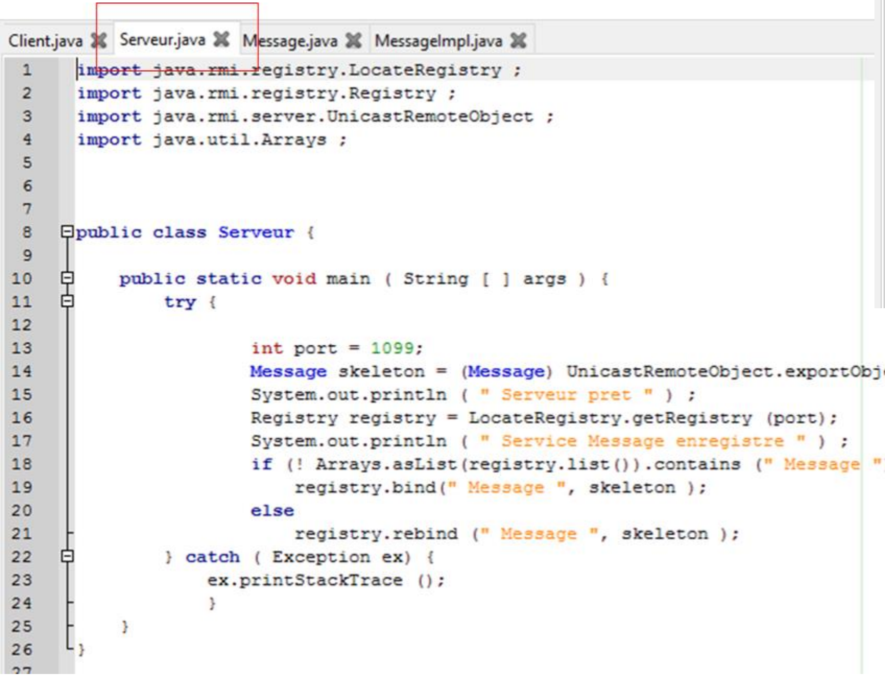
```
public class MessageImpl implements Message
{
    public MessageImpl ( ) {
        super ( ) ;
    }

    public String messageDistant ( )  {
        return ( "Message : Salut ! " ) ;
    }
}

```


SommeImpl.java

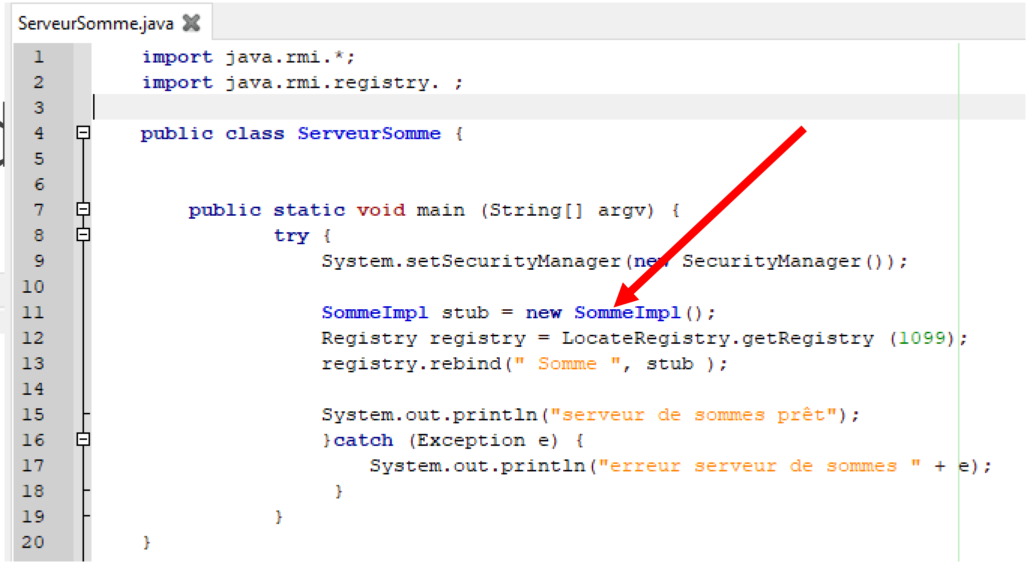
```
import java.rmi.*;
import java.rmi.server.*;


 public class SommeImpl extends UnicastRemoteObject implements Somme {

        public SommeImpl   () throws RemoteException {    }

        public int add(int a, int b) throws RemoteException {
            int result=a+b;
            return result;
        }
 }
```

The first version of the implementation is simple (but it will make the server source code more complex)
The second version of the implementation is more complex (but it will make the server source code more simple)

Several different forms to d



```java
import java.rmi.registry.LocateRegistry ;
import java.rmi.registry.Registry ;
import java.rmi.server.UnicastRemoteObject ;
import java.util.Arrays ;



public class Serveur {

    public static void main ( String [ ] args ) {
        try {

                int port = 1099;
                Message skeleton = (Message) UnicastRemoteObject.exportObject(new MessageImpl () , 0);
                System.out.println ( " Serveur pret " ) ;
                Registry registry = LocateRegistry.getRegistry (port);
                System.out.println ( " Service Message enregistre " ) ;
                if (! Arrays.asList(registry.list()).contains (" Message "))
                    registry.bind(" Message ", skeleton );
                else
                    registry.rebind (" Message ", skeleton );
        } catch ( Exception ex) {
            ex.printStackTrace ();
            }
    }
}
```



```java
import java.rmi.*;
import java.rmi.registry. ;

public class ServeurSomme {


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

                SommeImpl stub = new SommeImpl();
                Registry registry = LocateRegistry.getRegistry (1099);
                registry.rebind(" Somme ", stub );

                System.out.println("serveur de sommes prêt");
                }catch (Exception e) {
                    System.out.println("erreur serveur de sommes " + e);
                 }
            }
}
```

Supervised exercises (aka TDs) to be done on Feb 01st, 2021

RMI Encryption Service. The purpose of this exercise is to implement a distributed encryption service with RMI; it will propose the following 3 methods :

a. `encrypt(int):int` which, as encryption, will increment by 1 the integer transmitted as parameter ;
b. `encryptDocument(Document):Document` that will encrypt a text document passed by value by simply adding a string of characters; the business logic of a Document object is provided in the file **Annexes.tar.gz**
c. `EncryptFile(File):void` which will encrypt a file passed by reference, again by adding a string ; the business logic of a File type object is also provided in the file **Annexes.tar.gz**