JAVA - Network

DUT Info - Option ISI

Serialization of objects - concepts

- Since the JDK 1.1
- Serialization allows to store (on hard disk or another storage unit) an object as a flow of data.
- Deserialization is the opposite process allowing to get back a serialized object as it was at the serialization moment.

Serialization of objects - concepts

- Allows :
  - to store objects on storage units
  - to transfert objets on the network
  - to avoid binary format of files

Serialization of objects - concepts

- Store the state of objects
- Retrieve objects with the same state

Serialization of objects - practical

- Need to implement the *Serializable interface*
- Linked objects also have to be serializables

Objects flows

- Read/Write of objects
  - ObjectOutputStream
  - ObjectInputStream

Exemple - write

```
FileOutputStream fout = new
  FileOutputStream(“ tmp ”) ;
ObjectOutput s = new
  ObjectOutputStream(fout) ;
s.writeObject(“ Today ”) ;
S.writeObject(new Date()) ;
s.flush() ;
```

Remark : Primitive data are send into the flow using *writeInt*, *writeFloat* (respectively *readInt*, *readFoat*) primitives.

Exemple - read

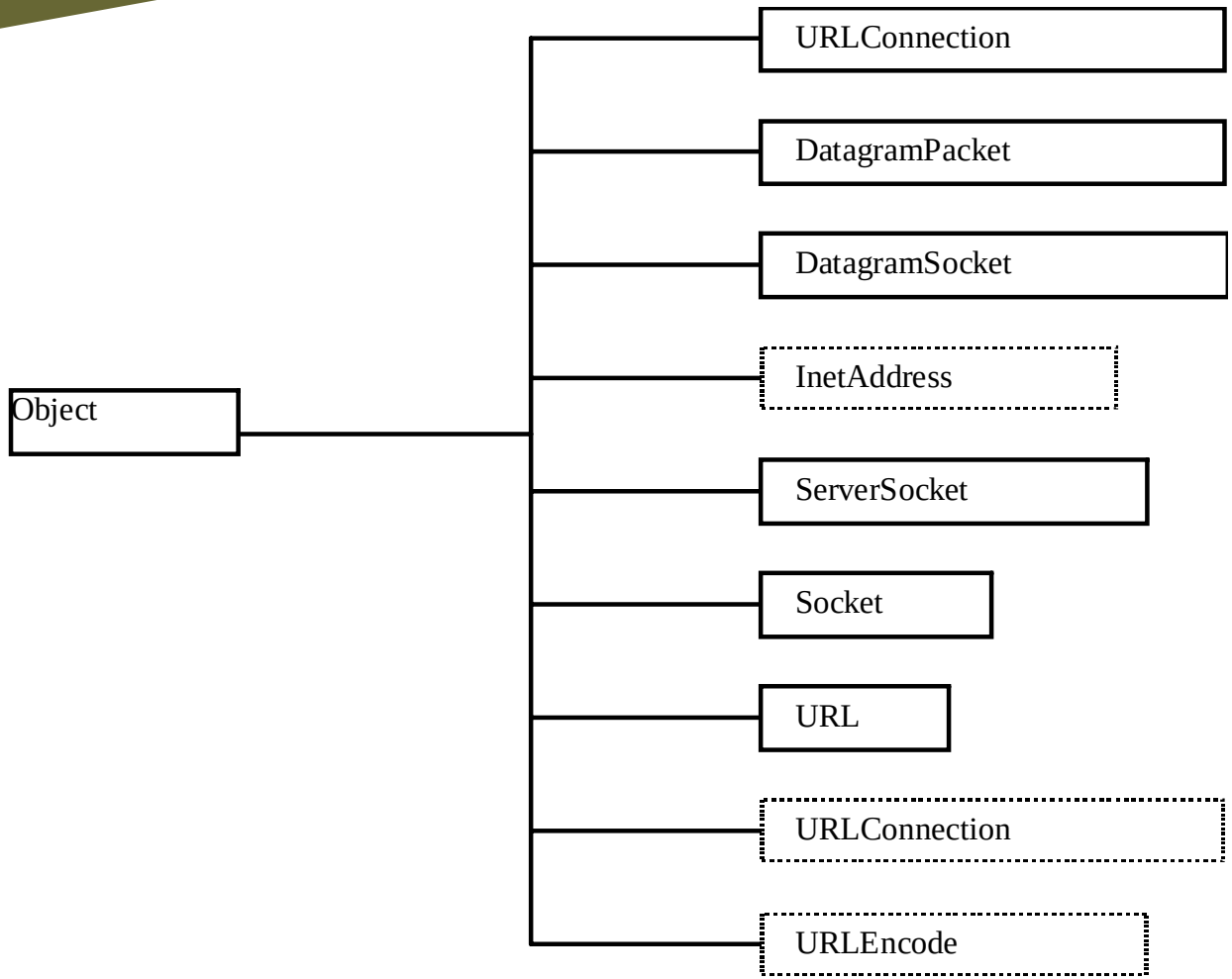
```
FileInputStream fin = new
  FileInputStream(“ tmp ”) ;
ObjectInput s = new
  ObjectInputStream(fin) ;
String today = (String)s.readObject();
Date date = (Date)s.readObject() ;
s.flush() ;
```

Remark : We need to read data with the same order as they were writen.

Security

- All fields of an object are serialized, including those into the *private* section.
  - Security problem if we do not want to store these information
  - Solution
    - Each critical attribute needs to be identified before with the keyword : *transcient*.
    - Rewrite (overwrite) methods *writeObjet*() and *readObject*() in order to they record/read only desired data

package java.net.*



Class URL

- Access to a machine on Internet using an URL *(Uniform Resource Locator)*, made of :
- *protocole* : http://
- *host* : iparla.iutbayonne.univ-pau.fr
- *port* : 1547
- *path* : ~mgrinfo2/pub/index.html

Complete URL : *http://iparla.iutbayonne.univ-pau.fr :1547/~mgrinfo2/pub/index.html*

Class URL : main methods

- Main methods of the class URL are :
  - *URL(String url)* : create an object using the string parameter (exception *MalformedURLException*).
  - *URL(String, String, String)* : protocol , host, path of the resource.
  - *String toString()* : return the URL as a strings.
  - *String getHost()* : return the name of the host linked to this URL.
  - *String getProtocol()* : return le protocol of this URL.
  - *String getPort()* : return the number of the associate port.
  - *InputStream openStream()* : realize the connection to the URL previously instantiated. An *InputStream* object is returned and permits to retrieve information specified into the URL. If the connection fails, the exception *IOException* is raised.

Class Socket (client)

```
String urlServeur = new
  String(“ www.iutbayonne.univ-
  pau.fr ”);

int portSocketServeur = 1547;

Socket socketClient (urlServeur,
  portSocketServeur);
```

Remark : Notice that the *bind* does not have to be done. It is automatic if the port number if given with the URL.

Class Socket : main methods

- *Socket(String host, int port)* : Create a flow on the socket and connect it to the host machine on the given port.
- *void close()* : close the socket.
- *void connect(SocketAddress endpoint)* : Connects the socket to the server.
- *void connect(SocketAddress endpoint, int timeout)* : Connects the socket to the server with a specific timeout.
- *InetAddress getInetAddress()* : Return the address onto is connected the socket.
- *InputStream getInputStream()* : Return an input stream for the socket.
- *InetAddress getLocalAddress()* : Return the local address onto which is connected the socket.
- *int getLocalPort()* : Return the local port onto which is connected the socket.
- *OutputStream getOutputStream()* : Return an output stream for the socket.
- *int getPort()* : Return the number of the port onto which is connected the socket. *boolean isConnected()* : Return the state of the connection of the socket.
- *void shutdownInput()* : Close the input stream of the socket.
- *void shutdownOutput()* : Close the output stream of the socket.

Class ServerSocket

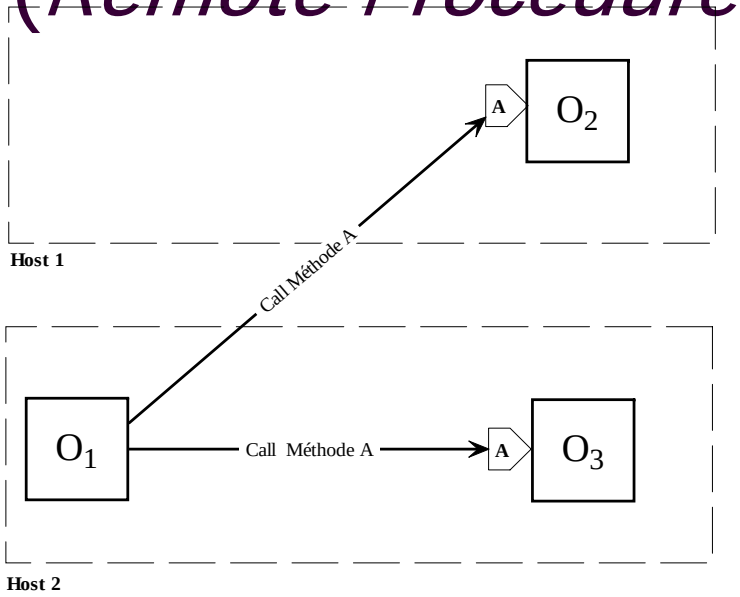
```
int portEcoute =1547;
ServerSocket ss = new
  ServerSocket(portEcoute);

while (true) {
  Socket SocketTravail = ss.accept();
  …
}
```

Class ServerSocket : main methods

- *ServerSocket(int port)* : Create a server socket with a specific port.
- *Socket accept()* : The server socket is switch on accept mode listening for a connection and doing automaticaly the *accept*.
- *void close()* : close the server socket..
- *InetAddress getInetAddress()* : return the local address of the server corresponding to the server socket.
- *int getLocalPort()* : return the number of the port of the server socket.
- *boolean isClosed()* : return the state of the server socket.
- *void Close()* : close the server socket.

JAVA - RMI

- RMI : *Remote Method Invocation*
- Goal: allows to invoke distant object using RPC (*Remote Procedure Call*).



RMI

- With an ideal world, we could :
  - invoke a method of a distant object (DO) as it was local
    - DO.method();
  - use a distant object even if we do not know where it is
    - DO = Service.search(« object A »);
  - Send/Receive distant objects
    - DO1 = OLocal.method(OD2);

RMI do it !

- Core API since the JDK 1.1
- This mechanism is comparable with CORBA, not free but with more interoperability.
- Allows to call Java objects executed in distinct/distant virtual machines
- Use *socket* mechanism.

RMI : Distants Objects

- A distant object (*server*) is described with one or several interfaces.
  - An interface describes available distant methods (may be not all methods)
- Is manipulated as a local object.

RMI : Distant Objects

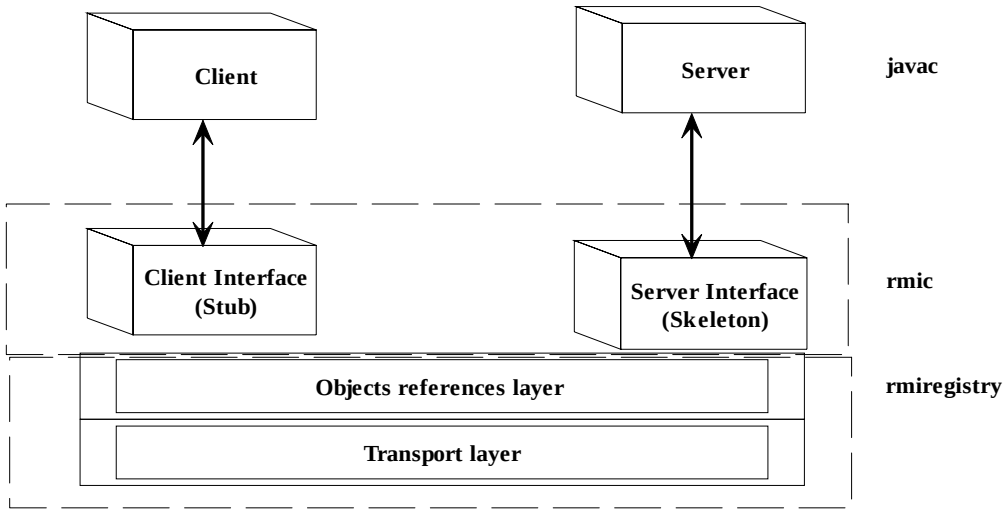
- Parameters are ALWAYS copies and not references.
  - The class must be *serializable*
  - excepted if types are primitive.

RMI : Distant Objects

- Client objects does not interact directly with the object but use its interface.
  - An interface define names, return types, parameters of available methods. The is its signature.
- A DO must implements the interface : *java.rmi.Remote*
  - *RemoteException management when RPC are done*

RMI : Stubs/Skeletons

- Stub/Skeleton
  - programs realizing RPC and doing transtyping parameters if needed (*marshalling/demarshalling*)
  - *rmic* -> Stub/Skeleton



RMI : Stubs/Skeletons

- Stubs = Local representation of the DO
  - Parameters are « *marshallised* » and are serialized in order to be send to the skeleton
- The skeleton « *unmarshallises », call the distant method and return the value the the calling object.*

RMI : Object References Layer

- Allows the retrieve the reference to a DO using its local reference (Stub)
- *rmiregister* (once per JVM)
  - directory of DO available.

RMI : Transport Layer

- Connect the two JVM
- Follow connections
- Listen/Answer to invocations.

RMI : practical

- Creation of the service interface

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

RMI : practical

- Implementation of the method

```
import java.rmi.*;
import java.rmi.server.*;
public class EchoImpl extends UnicastRemoteObject implements Echo {
    public EchoImpl(String nameService) throws RemoteException {
            super();
            try {
               Naming.rebind(nameService,this);
            } catch(Exception e) {
                 System.out.println("Error rebind- "+e);
               }
    }
public String Echo (String chaine) throws RemoteException {    return
    "Echo : " + chaine;    }
}
```

RMI : practical

- Implementation of the server, declaration of theservice

```
import java.rmi.*;
import java.rmi.server.*;
public class AppliServer {
    public static void main(String args[]) {
      try {
        System.setSecurityManager(new RMISecurityManager());
        EchoImpl od = new EchoImpl("Echo");
      } catch(Exception e) {
            System.out.println("Erreur server : "+e.getMessage());
        }
    }
}
```

RMI : practical

- Let ’s compile all !
  - *javac Echo.java*
  - *javac EchoImpl.java*
  - *javac AppliServer.java*
- stubs/skeletons creation (after compilation)
  - *rmic EchoImpl*
    - -> EchoImpl_Stub.class and EchoImpl_Skel.class

RMI : practical

- The client

```
import java.rmi.*;
import java.rmi.registry.*;
public class clientEcho {
    public static void main (String argv[]) {
      System.setSecurityManager(new RMISecurityManager());

              String url="rmi://"+argv[0]+"/Echo";
       try {
         Echo od = (Echo)Naming.lookup(url);
         System.out.println(od.Echo(argv[1]));
       } catch(Exception e) {
            System.out.println("Errorr client - service not find");
       }
   }
```

RMI : practical

- Execution (after client compilation)
  - *rmiregistry&*
  - We start the server with another *security.policy* as the origin one !
    - java -Djava.security.policy=./wideopen.policy AppliServer &
  - On another machine (or the same)
    - java -Djava.security.policy=./wideopen.policy clientEcho [nomserveur|localhost] hello

RMI : practical

- java.security.policy

```
grant {
// Allow everything for now permission
java.security.AllPermission;
};
```

Let ’s programming a bit, but well !

- A lot of files
  - .java
  - .class
  - stubs & skeletons
- Solution :
  - Group using packages
  - Create a .jar

Packages

- Package namepackage
  - *class toto -> namepackage.toto*
  - key word: package namepackage (first line of the source code)
  - Compilation :
    - *javac file.java -d .* (if the directory « namepackage » if into the current directory.
    - .*class are automatically moved into it.*

    The class is names *namepackage.toto instead of toto*.

Archives Jar

- To group the set of classes of the same project
  - *jar cvf package.jar files (class and even java)*
- Use
  - *java -classpath DirectoryOfTheArchive.jar NameOfTheClass.class*