Programmation socket en Java

Module
Systèmes Communicants et Synchronisés

Master Informatique
1ère année

L. Philippe & V. Felea

Généralités

- présentes dès les premières spécifications du langage (v 1.1)
  - paquetage : *java.net*
- interface d'utilisation plus simple, surtout TCP
- Java objet → Sockets objets
  - *Socket* = socket de communication
  - *ServerSocket* = socket de connexion
  - *InetAddress* = adresse IP
- passage d'objets

Classe *InetAddress* (1)

- manipulation des adresses IP
  - accès DNS (*D*omain *N*ame *S*ystem)
- classe : *java.net.InetAddress*
- définition: public class *InetAddress* implements *Serializable*
- méthodes :
  - *static InetAddress getLocalHost()*
    throws UnknownHostException
  - *static InetAddress getByName(String host)*
    throws UnknownHostException
  - *static InetAddress[] getAllByName(String host)*
    throws UnknownHostException

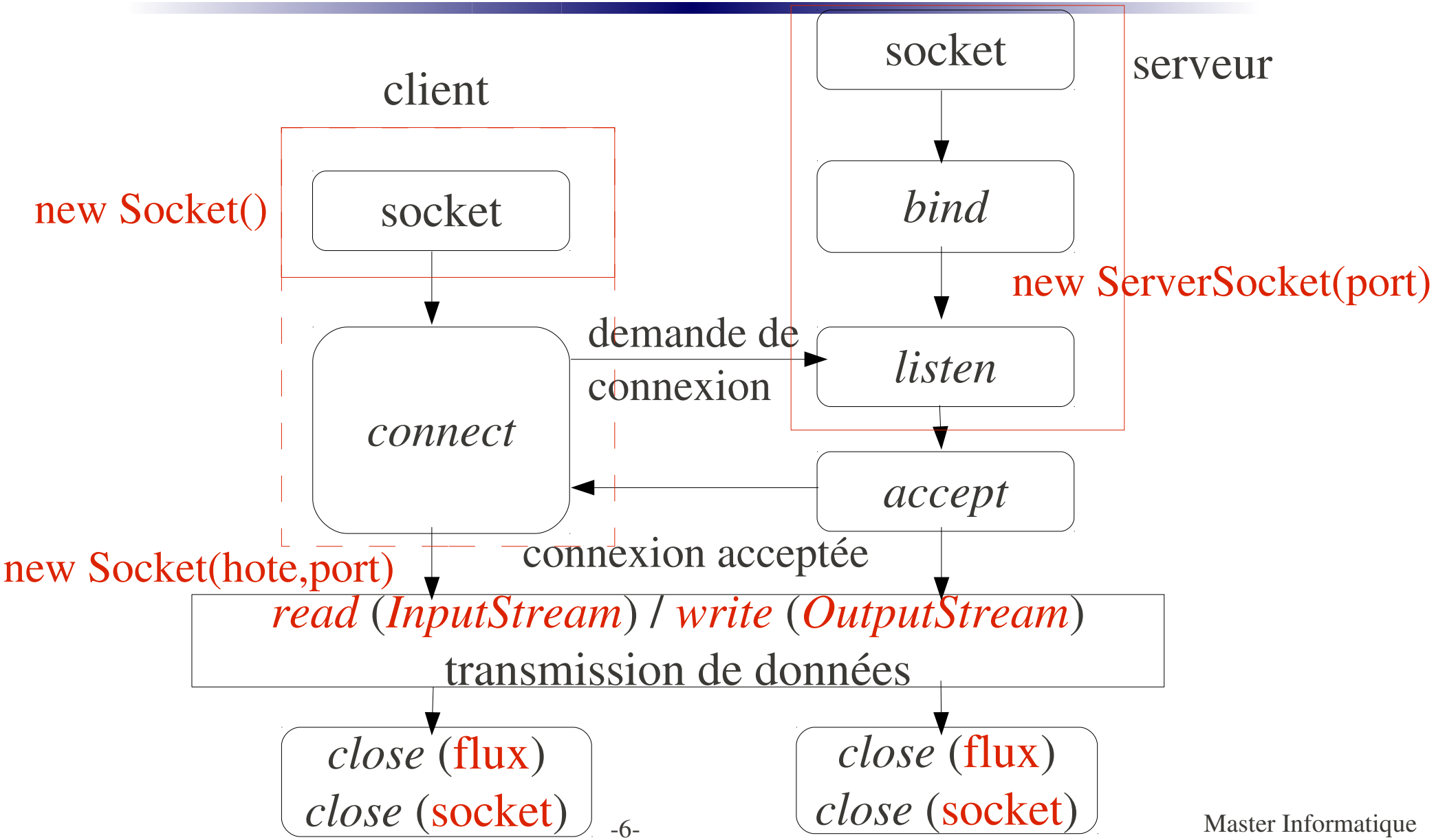
Classe *InetAddress* (2)

- méthodes d'accès aux données (méthodes d'instance)
  - *public String getHostName()*
  - *public String getHostAddress()*
  - *public byte[] getAddress()*

Sockets en mode connecté (1)

- plus souvent utilisées
- deux classes de sockets
  - communication : classe *Socket*
    - client : *socket* + *connect*
    - serveur : retour *accept*
  - connexion : classe *ServerSocket*
    - *socket* + *bind* + *listen*

Sockets en mode connecté (2)



Classe *ServerSocket* – constructeurs

- *public ServerSocket()*
- *public ServerSocket(int port) throws IOException*
  - *port* = 0 crée une socket sur un des ports libres
- *public ServerSocket(int port, int backlog) throws IOException*
  - *backlog* = taille maximale de la file des connexions entrantes (connexion refusée si la file est pleine) – par défaut 50
- *public ServerSocket(int port, int backlog, InetAddress addr) throws UnknwnHostException, IOException*

Classe *ServerSocket* – méthodes

- *public Socket accept() throws IOException*
  - bloquante si pas de demande de connexion dans la file d'attente des connexions
- *public void close() throws IOException*

- *public InetAddress getInetAddress()*
- *public int getLocalPort()*

Classe *Socket* – constructeurs

- *public Socket() throws IOException* (socket non liée)
- *public Socket(String host, int port) throws UnknownHostException, IOException*
- *public Socket(InetAddress address, int port) throws IOException*
- *public Socket(String host, int port, InetAddress localAddr, int localPort) throws UnknownHostException, IOException*
- *public Socket(InetAddress address, int port, InetAddress localAddr, int localPort) throws IOException*

Classe *Socket* – méthodes (1)

- méthodes de gestion de la connexion
  - *public InetAddress getInetAddress()*
  - *public InetAddress getLocalAddress()*
  - *public int getPort()*
  - *public int getLocalPort()*

  - *public void close()*
  - *public void shutdownInput() throws IOException*
  - *public void shutdownOutput() throws IOException*

Classe *Socket* – méthodes (2)

- communication repose sur des flux (streams)
- méthodes de gestion de la communication (obtenir les flux de communication)
  - *public InputStream getInputStream( ) throws IOException*
  - *public OutputStream getOutputStream( ) throws IOException*

Les flux

- stream = flux
- objets encapsulant des flux d'octets
- accès aux I/O : fichiers, périphériques, etc.
- données mémorisées en file
  - accès séquentiel
  - pas de marqueur
- input / output
- peuvent être spécialisés : chaîne, Object

Classe *InputStream* – méthodes

- *int available()* : nombre d'octets pouvant être lus sans blocage
- *abstract int read()* : lit **l'octet** suivant (-1 retourné si fin de flux)
- *int read(byte[] b)* : lit un nombre d'octets, les mémorisant en *b* et retourne le nombre d'octets réellement lus
- *int read(byte[] b, int offset, int len)* : lit au maximum *len* octets écrits en *b* à partir de *offset*
- *long skip(long n)* : ignore *n* octets du flux, et retourne le nombre d'octets réellement ignorés

- *void close()*

  exception levée : *IOException*

Classe *OutputStream* – méthodes

- *abstract void write(int b)* : écrit **l'octet** (les 8 bits de poids faible de l'entier *b*)
- *void write(byte[] b)*

  = *write(b, 0, b.length)*
- *void write(byte[] b, int offset, int len)*
- *void close()*

exception levée : *IOException*

Exemple – serveur (réception)

```
import java.net.Socket; import java.net.ServerSocket;
import java.io.InputStream; import java.io.IOException;
public class ServeurBase {
  public static void main(String [] args) {
    ServerSocket srv ;
    try {
      srv = new ServerSocket(5555) ;
      Socket sockComm = srv.accept() ;
      InputStream is = sockComm.getInputStream();
      byte[] tabloServ = new byte[4];
      int recu = is.read(tabloServ);
      // afficher tabloServ
      is.close(); // ferme le flux
      sockComm.close() ;  srv.close(); // ferme la socket de comm et de connex
    } catch(IOException e) {…}
  } // main
} // class
```

Exemple – client (envoi)

```
import java.net.Socket;
import java.io.OutputStream; import java.io.IOException;
public class ClientBase {
  public static void main(String [] args) {
    Socket sock ;
    String machineServ = "nomMachine";
    try {
        sock = new Socket(machineServ, 5555) ;
        OutputStream os = sock.getOutputStream();
        byte[] tabloClient = new byte[4];
        tabloClient[0] = 1; // initialisation
        os.write(tabloClient);  // envoi sur le flux de sortie de la socket
        os.close();  // fermeture de flux
        sock.close() ;  // fermeture de socket
    } catch(IOException e) { }
  }
}
```

Notions sur les objets sérialisables (1)

- un objet sérialisable (transmissible par valeur hors de sa JVM)
  - implémente l'interface *java.io.Serializable*
  - interface ayant rôle de marqueur (pas d'attributs, ni d'interface)
- les objets référencés dans un objet sérialisable doivent aussi être sérialisables
- comment rendre effectivement un objet sérialisable ?
  - pour les variables de types primitifs (int, boolean, …) : rien à faire
  - pour les objets dont les attributs sont constitués de telles variables : la classe de l'objet doit implémenter l'interface *java.io.Serializable*
  - on peut éliminer une variable de la représentation sérialisée en la déclarant *transient*
  - pour un attribut non sérialisable, il faut fournir des méthodes *readObject*() et *writeObject*()

Notions sur les objets sérialisables (2)

- exemples de sérialisation
  - passage en paramètres
  - écriture sur un fichier
- le support de sérialisation est un flux (stream) : classes *java.io.ObjectOutputStream* et *java.io.ObjectInputStream*
  - (méthodes privées) *writeObject(ObjectOutputStream)*, *readObject(ObjectInputStream)*
  - peuvent préciser un comportement différent de celui par défaut (*defaultWriteObject*, *defaultReadObject*)

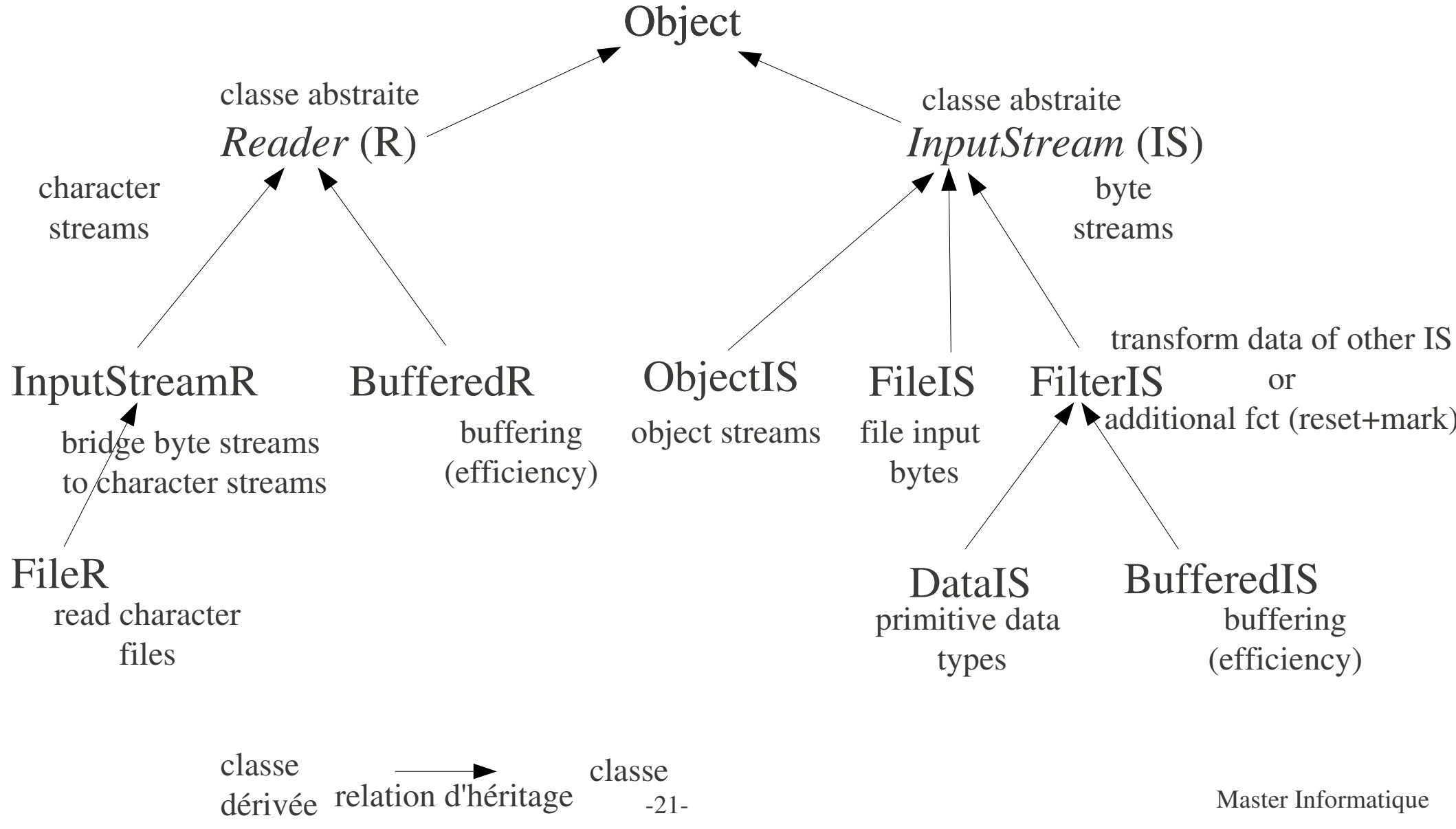
Exemple

```
public class SeriaCls implements Serializable
        String str1;
        transient String str2;
        public SeriaCls(...)//constructeur
        public String toString() // formatage
        private void writeObject(ObjectOutputStream outStr) throws IOException {
                outStr.defaultWriteObject();
                outStr.writeObject(str2); // forcer écriture même si attribut transient
        }
        private void readObject(ObjectInputStream inStr)
                        throws IOException, ClassNotFoundException {
                inStr.defaultReadObject();
                str2 = (String)inStr.readObject();
        }
}
// classe de test (out / in)
out.writeObject(new SeriaCls(...));    SeriaCls objSeria = (SeriaCls)in.readObject();
```

Classes sérialisables

- interface *Serializable*
  - objets sans dépendance locale
  - non sérialisables
    - Thread
    - Stream
    - Socket
    - et les classes dérivées...
  - encapsulation : transformation en suite d'octets
- communication d'objet

Transfert de données – hiérarchie classes (flux d'entrée : *input*)



Transfert de données

- *Reader/Writer* : type abstrait pour le flux en lecture/écriture pour des ensembles de caractères
  - *BufferedReader/Writer* : flux bufferisé
- *InputStream/OutputStream* : type abstrait pour le flux de lecture/écriture pour des ensemble d'octets
  - *ObjectInput/OutputStream* – types référence sérialisés
  - *DataInput/OutputStream* – types primitifs
  - *BufferedInput/OutputStream* – flux bufferisé
- *InputStreamReader / OutputStreamWriter*
  - réalise la conversion entre flux d'octets et flux de caractères

Classes *Object*[*Input/Output*]*Stream*

- méthodes
  - *Object readObject() throws IOException, ClassNotFoundException*
  - *void writeObject(Object o) throws IOException*
- types d'objets
  - ok pour write : *Object* = super classe
  - transtypage (casting) pour *readObject*

```
MaClasse mc = (MaClasse) flux.readObject();
```

Classes *Object*[*Input/Output*]*Stream*

- [dé]sérialisation d'objets : les objets écrits/lus depuis le flux doivent implémenter *java.io.Serializable*
- flux obtenus
  - à partir de la socket (*Input/OutputStream) : flux d'octets*
  - création d'un nouvel objet qui encapsule le flux d'octets : flux d'objets

```
// Socket s
OutputStream os = s.getOutputStream();
InputStream is = s.getInputStream();

ObjectInputStream ois = new ObjectInputStream(is);
ObjectOutputStream oos = new ObjectOutputStream(os);
```

Exemple classe sérialisable

```
import java.io.Serializable;

public class Donnees implements Serializable {

    private long val;
    private String chaine;

    public long getVal() { return val; }
    public String getChaine() { return chaine; }
}
```

Exemple – serveur

```
// import ...
public class ServerObj {
   public static void main(String [] args) {
       ServerSocket srv ;
       int port = 5555 ;
       srv = new ServerSocket(port) ;
       try {
          Socket s = srv.accept() ;

          InputStream is = s.getInputStream();
          ObjectInputStream ois = new ObjectInputStream(is);
          try {
            Donnees maDon;
            maDon = (Donnees) (ois.readObject());
            System.out.println("J'ai recu: " + maDon.getVal() + " " + maDon.getChaine());
             ...
```

Exemple – client

```
// import ...
public class ClientObj {
    public static void main(String [] args) {
        Socket s;
        Donnees d = new Donnees(45, "test Sockets Java");
        String hote = "machine" ; int port = 5555;

        try {
            s = new Socket (hote, port) ;
            OutputStream os = s.getOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(os);

            oos.writeObject(d);
            …
```

Classes *DataInputStream/DataOutputStream*

- échanges de types primitifs d'une manière portable
- obtenues à partir de
  - *InputStream/OutputStream* – objets paramètres du constructeur
- méthodes
  - *readBoolean, readChar, readInt, readLong, …*
  - *writeBoolean, writeChar, writeInt, writeLong, ...*

Classes *InputStreamReader/OutputStreamWriter*

- classes dérivées de java.io *Reader/Writer*
- pont entre octets et caractères
  - *InputStreamReader* : lit des octets et les décode en caractères
  - *OutputStreamWriter* : écrit des caractères en octets
  - précision de l'encodage des caractères (charset)
- meilleure utilisation avec *BufferedReader* et *BufferedWriter*
- exemple

```
BufferedReader in
  = new BufferedReader(new InputStreamReader(is));

BufferedWriter out
  = new BufferedWriter(new OutputStreamWriter(os));
```

Utilisation *BufferedReader/BufferedWriter*

- méthodes lecture (exception levée *IOException*)
  - *int read()* : un seul caractère
  - *int read(char[])* : un ensemble de caractères
  - *String readLine()* : une ligne de texte
  - *long skip(long)* : ignore des caractères
- méthodes écriture (exception levée *IOException*)
  - *write(int c)* : un caractère
  - *write(char[], int off, int len)* : un sous-tableau de caractères
  - *write(String)* : une chaîne
  - *newLine()* : un séparateur de lignes

Exemple – serveur

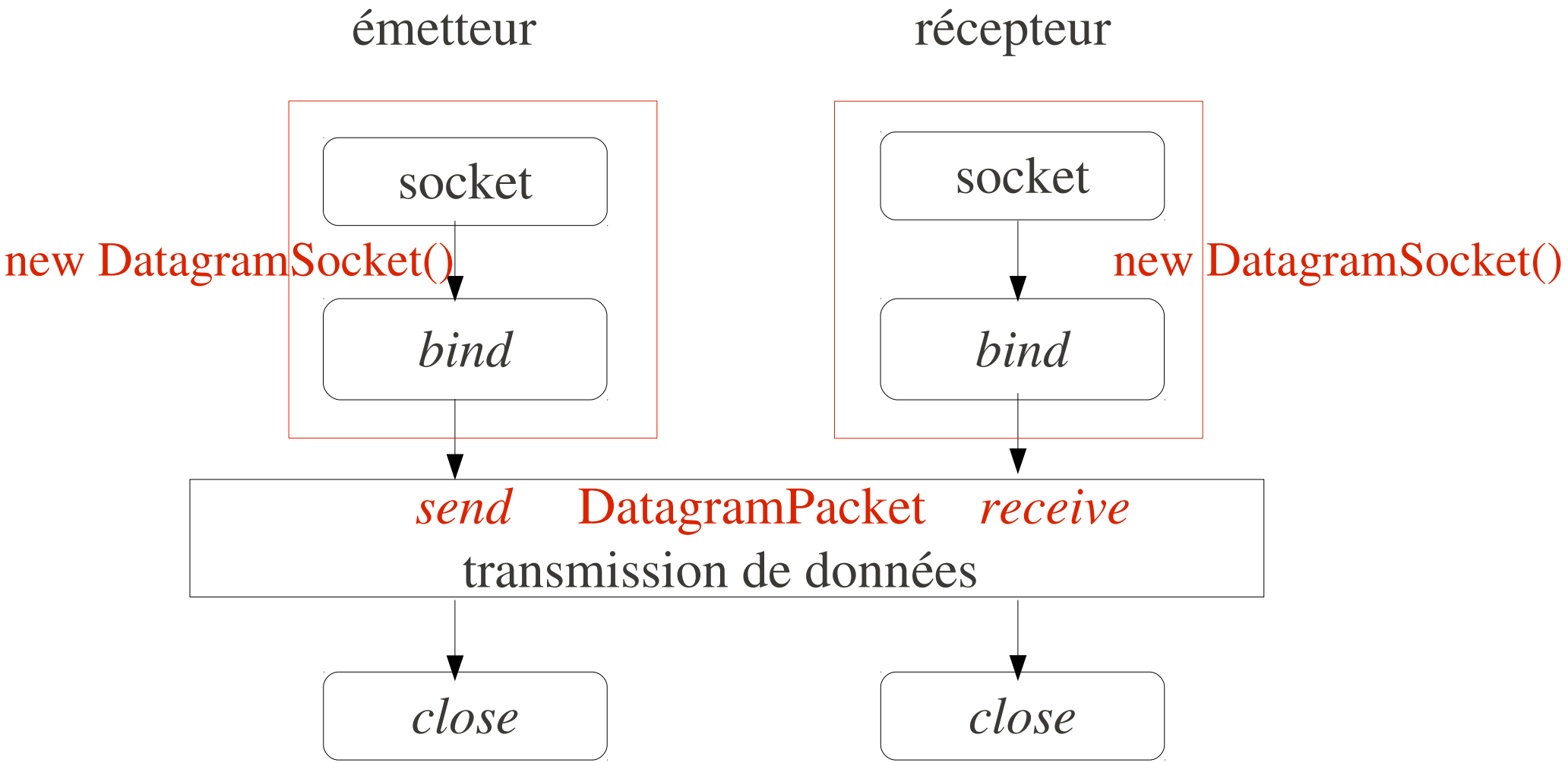
```
// import ...

public class ServerBufferedReader {
    public static void main(String [] args) {
        ServerSocket srv ;
        int port = 5555 ;
        try {
            srv = new ServerSocket(port) ;
            Socket s = srv.accept() ;
            InputStream is = s.getInputStream();
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String message = br.readLine();
            System.out.println("J'ai recu le message: "+ message);
            OutputStream os =  s.getOutputStream();
            os.write(message.length());
            ...
```

Mode non-connecté (1)

- classe : *DatagramSocket*
- constructeurs
  - *public DatagramSocket() throws SocketException*
  - *public DatagramSocket(int port) throws SocketException*
- communication : échange de paquets (type *DatagramPacket*)
  - *void send(DatagramPacket p) throws IOException*
  - *void receive(DatagramPacket p) throws IOException*

Mode non-connecté (2)



Classe *DatagramPacket* (1)

- paquet échangé en UDP
- contient : IP, port, données, taille des données
- constructeurs
  - *public DatagramPacket(byte[] buf, int length)*
    - construit une datagramme pour recevoir des paquets de taille donnée
  - *public DatagramPacket(byte[] buf, int length, InetAddress address, int port)*
    - construit une datagramme pour envoyer des paquets de taille donnée vers le port d'une destination

Classe *DatagramPacket* (2)

- méthodes
  - *public InetAddress getAddress()*
  - *public int getPort()*
  - *public void setAddress(InetAddress addr)*
  - *public void setPort(int port)*
  - *public byte[] getData()*
  - *public int getLength()*
  - *void setData(byte[])*
  - *void setLength(int length)*

Exemple – serveur

```
// import ...
public class UdpServer {
    public static void main(String[] args) {
       int port = 5555;
       byte[] buf = new byte[1000];
       // construit un paquet datagram pour recevoir les données
       DatagramPacket dp =  new DatagramPacket(buf, buf.length);

       DatagramSocket socket;
       try {
          socket = new DatagramSocket(port);
          // bloque en attente d'un paquet datagram
          socket.receive(dp);
          String rcvd = Dgram.toString(dp);
          System.out.println("Received :"+rcvd);
        }
       ...
```

```
return new String(dp.getData(),
                  0, dp.getLength());
```

Exemple – client

```
// import ...
public class UdpClient {
   public static void main(String[] args) throws IOException {
      int port = 5555;
      InetAddress addr = InetAddress.getByName("machine");
      // crée une socket UDP
      DatagramSocket socket = new DatagramSocket();

      try {
         System.out.println("Sending the udp socket...");
         // create datagram packet to be sent and send it
         socket.send(Dgram.toDatagram("HI", addr, port));
      }
       ...
```

```
String toDatagram(String s,
          InetAddress destIA, int destPort) {
   byte[] buf = s.getBytes();
   return new DatagramPacket(buf, buf.length,
          destIA, destPort);
}
```

Pour aller plus loin...

- accès aux données par *java.io*
  - Input/Output stream : flux simples
  - *ObjectStream*, *StreamReader*, *FileStream* : évolués
- accès aux types de base : *java.nio*
  - depuis 1.4 (New Input Output)
  - classes *Buffer* (java.nio), *Charset* (java.nio.charset), *Channel* (java.nio.channels)
  - déclinées en *CharBuffer*, *IntBuffer*, *DoubleBuffer*...
  - intérêt : performances !