Les entrées-sorties

- **Description des principales classes de flux d'E/S**
  - **- écriture et lecture d'octets**
  - **- écriture et lecture de caractères**

Les classes étudiées ici sont utilisées
pour les entrées-sorties sur réseaux ou dans un fichier

Yannick.Parchemal@parisdescartes.fr

La classe File : contructeurs

Les fichiers sont représentés par des instances de la classe File

```
public  class  java.io.File  {….
   // constantes
   public static final char separatorChar ; // vaut '/' sous Linux et '\' sous Windows
   // Constructeurs
   /** construit le File correspondant au chemin 'path' */
   public File(String  path){…}
   /** construit le File correspondant au chemin obtenu à partir du répertoire 'path'
      *  et du nom 'name' */
   public File(String  path,String name){…}
```

```
File f = new File("projet"+File.separatorChar+"données");
  ou
File f = new File("projet","données");
```

La classe File

Les fichiers et répertoires sont représentés par des instances de la classe File

```
// Quelques méthodes de la class File
   public boolean exists( ); // le fichier existe-t-il ?
   public boolean isDirectory( ); // est-ce un répertoire ...
   public boolean isFile( );  // ou un fichier ?
   public boolean canRead( ); // autorisation de lecture ?
   public boolean canWrite( ); // autorisation d'écriture ?

   public String getAbsolutePath( );      // rend l'adresse absolue

   public String[] list( );   // rend la liste des fichiers si c'est un répertoire

   public boolean delete( ); // supprimer le fichier
   public boolean renameTo(File  dest); // renomme le fichier

   public void mkdir( ); // créer le répertoire correspondant à ce File
   public void mkdirs( ); // créer les répertoires correspondant à ce File
}
```

Classe File : exemple

```
/** demande à l'utilisateur un nom de fichier à partir du terminal 'term'
Réitère sa demande tant que le nom n'est pas un nom de fichier existant
rend le fichier correspondant à ce nom*/
public File getExistingFile(Terminal term){
File res = null;
do {
String nomFichier = term .readString("Donner un nom" );
File file = new File(nomFichier );
boolean existe= (file.exists( ));
if( !existe)
      term .println("Le fichier "+ file.getAbsolutePath( )+" n'existe pas !!\n"+
                  " Donner un autre nom");
      else res=file;
}
while (res == null);
return res;
}
```

InputStream

InputStream est la classe mère de toutes les classes de lecture de flux d'octets.

```
package java.io;
public  abstract  class  java.io.InputStream {
…..
    // lecture d'un octet (retourne -1 si fin du flux atteint)
    public abstract int read( ) throws IOException;

   // lecture de plusieurs octets par appel répété à read( ) (retourne le nombre d'octets lus)
    public int read(byte  b[ ] ) throws IOException{…}

   // fermeture du flux
   public void close( ) throws IOException{…}

    // retour au début
    public void reset( ) throws IOException{…}
```

Remarque : les classes de lecture de flux d'octets dérivent de InputStream. Elles doivent donc au minimum définir la méthode read( )

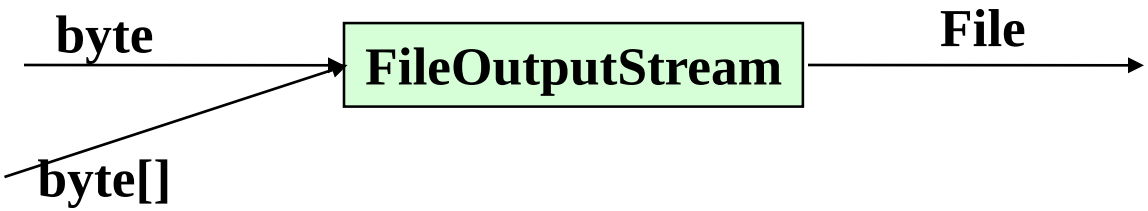
OutputStream

```
public  abstract  class  OutputStream
{
……
//écriture d'un byte 'b'
public abstract void write(int  b) throws IOException;

// écriture des éléments du tableau de byte 'tB'
// Consiste ici en l'appel répété de write(int)
// redéfini plus efficacement dans les sous-classes
public void write(byte  tB [ ] ) throws IOException{…}

// fermeture du flux
public void close( ) throws IOException{…}

// vider le buffer
public void flush( ) throws IOException{…}
}
```

Les classes de flux d'écriture de données dérivent de OutputStream.
Elles doivent donc au minimum redéfinir la méthode write(int)

FileOutputStream

Pour écrire des octets et des tableaux d'octets dans un fichier




```
public  class  java.io.FileOutputStream extends  java.io.OutputStream
{// Construction d'un flux de sortie vers un fichier
    public FileOutputStream(File  file);
...
// Deux méthodes … qui font appel à des fonctions natives systèmes
   public void write(int b) ;
   public void write(byte [ ]b) throws IOException
…….
}
```

Les données arrivant à un FileOutputStream sont immédiatement écrites sur disque

Exemple d'écriture dans un fichier

```
// création du File puis du FileOutputStream
File file = new File("monFichier");
FileOutputStream fos = new FileOutputStream(file);

// ecriture d'un byte dans le fichier
fos.write(56); // attention : un byte doit être un entier compris en 0 et 255
// ecriture d'un autre byte dans le fichier
fos.write(125);

// on crée un tableau de byte et on le rempli
byte [ ] tb = new byte[1000];
for (int i=0;i<tb.length;i++) tb[i] = (byte) (i%100);
// on écrit les 1000 bytes dans le fichier
fos.write(tb);

// et on ferme le flux de sortie
fos.close( );

// le fichier "monFichier" contient donc maintenant 1002 bytes.
```

Exemple d'écriture dans un fichier une fonction …

```
public void ecrireDansFichier(File file){

FileOutputStream fos = new FileOutputStream(file);
fos.write(56);
fos.write(125);

byte [ ] tb = new byte[1000];
for (int i=0;i<tb.length;i++) tb[i] = (byte) (i%100);
fos.write(tb);

fos.close( );
}
```

Problème : le flux ne sera pas fermé en cas d'erreur …

Exemple d'écriture dans un fichier une fonction avec gestion d'exception

```
public void ecrireDansFichier(File file){

FileOutputStream fos = new FileOutputStream(file);

try {
      fos.write(56);
      fos.write(125);

      byte [ ] tb = new byte[1000];
      for (int i=0;i<tb.length;i++) tb[i] = (byte) (i%100);
      fos.write(tb);
      }
finally {
    fos.close( );
              }
}
```

Solution avec finally : le flux sera fermé en cas d'erreur

Exemple d'écriture dans un fichier une fonction avec gestion d'exception Depuis java 7 : try-with-resources Statement

```
public void ecrireDansFichier(File file){

try (FileOutputStream fos = new FileOutputStream(file)) {
        fos.write(56);
        fos.write(125);

        byte [ ] tb = new byte[1000];
        for (int i=0;i<tb.length;i++) tb[i] = (byte) (i%100);
        fos.write(tb);
        }
}
```

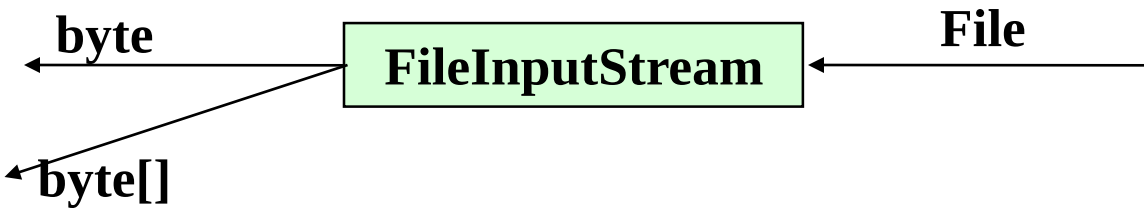
FileOutputStream implémente l'interface AutoCloseable

Une méthode dans l'interface AutoCloseable :

void close( ) throws Exception

FileInputStream

Pour lire des octets et des tableaux d'octets dans un fichier




```
public  class  java.io.FileInputStream extends  java.io.InputStream
{// Construction d'un flux d'entrée en provenance d'un fichier
    public FileInputStream(File  file);
...
// Deux méthodes … qui font appel à des fonctions natives systèmes
   public int read( );
   public int read(byte b[]) throws IOException
…….
}
```

Les données demandées à un FileInputStream sont lues au fur et a mesure dans le fichier

Exemple de lecture à partir d'un fichier

```
// on lit le contenu du fichier de l'exemple précédent : il contient 1002 bytes
// création du File puis du FileInputStream
File file = new File("monFichier");
FileInputStream fis = new FileInputStream(file);

// lecture du premier byte contenu dans le fichier
int n1 = fis.read( );
// lecture du second byte
int n2 = fis.read( );

// on crée un tableau de 1000 bytes
byte [ ] tb = new byte[1000];

// on lit les 1000 bytes suivant du fichier
fis.read(tb);

// et on ferme le flux
fis.close( );
```

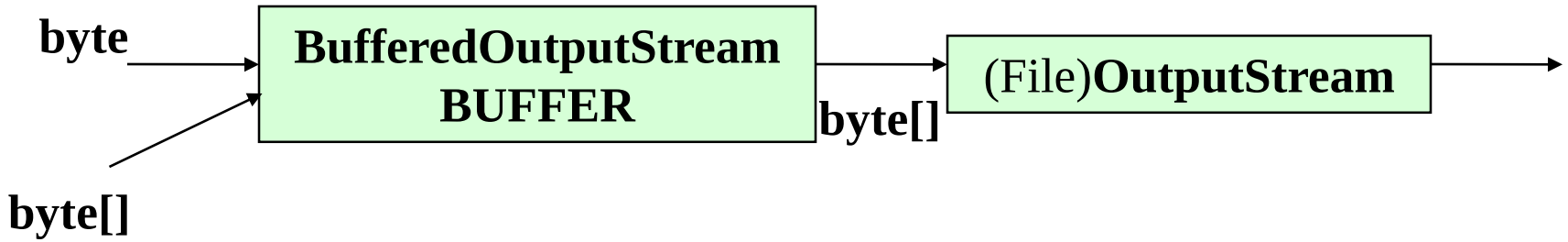
Un autre exemple de lecture

On aurait pu récupérer le contenu du fichier précédent de façon différente
Le fichier contient 1002 octets sans autre structure.
On peut récupérer le contenu d'un coup dans un tableau de 1200 octets par exemple

```
File file = new File("monFichier");
FileInputStream fis = new FileInputStream(file);

// on crée un tableau de 1200 bytes
byte [ ] tb = new byte[1200];

// on lit les 1002 bytes du fichier
int nbLu = fis.read(tb);    // nbLu doit valoir 1002 : le nombre d'octets du fichier,
            // seuls les 1002 premiers éléments du tableau sont affectés

// et on ferme le flux
fis.close( );
```

BufferedOutputStream

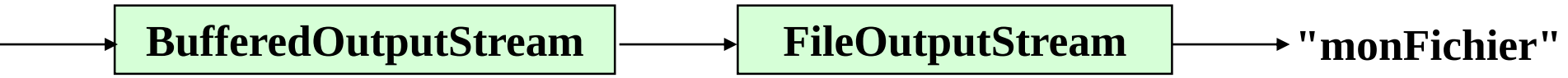
Pour écrire des octets et des tableaux d'octets
en évitant des accès disques ou réseaux trop nombreux



La taille du buffer est donnée à la création du flux.
Le buffer n'est vidé que lorqu'il est plein ou lorsqu'un appel a flush( ) est effectué.

```
public  class  java.io.BufferedOutputStream extends  java.io.FilterOutputStream {
// Constructors
      public BufferedOutputStream(OutputStream  out);/*size=8192*/
      public BufferedOutputStream(OutputStream  out, int size);
// Methods
     public void flush( );
……}
```

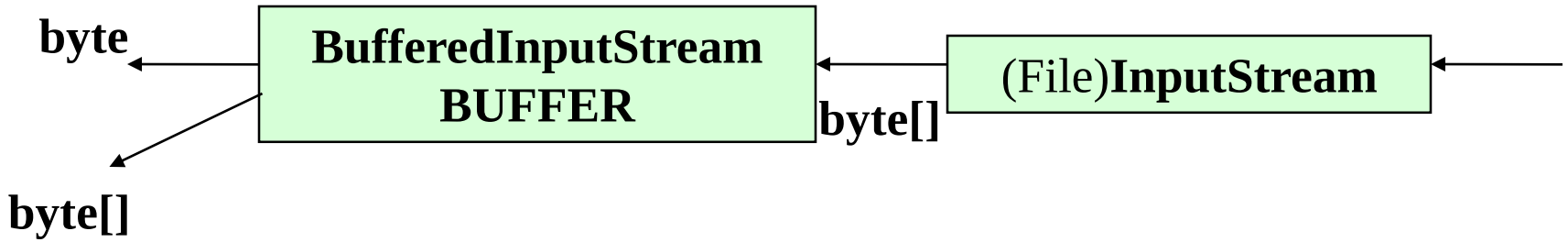
BufferedOutputStream




```
// création du File puis du FileOutputStream
File file = new File("monFichier");
FileOutputStream fos = new FileOutputStream(file);
// création du BufferedOutputStream lié à ce FileOutputStream
BufferedOutputStream bos = new BufferedOutputStream(fos,512);

// l'instruction suivante écrit 1000 octets dans le BufferedOutputStream
// Le buffer étant de taille 512, au moment de l'écriture du 512 ème,
// les 512 premiers éléments vont être transmis en bloc au FileOutputStream
// qui va les écrire dans le fichier. Les 488 vont être bufferisés
for (int i=0;i < 1000 ; i++ ) bos.write(i%100);
// L'appel à close se traduit par le vidage du buffer
bos.close( );
// maintenant le fichier contient 1000 octets
```

BufferedInputStream

Pour lire des octets et des tableaux d'octets dans un fichier en évitant des accès disques ou réseaux trop nombreux

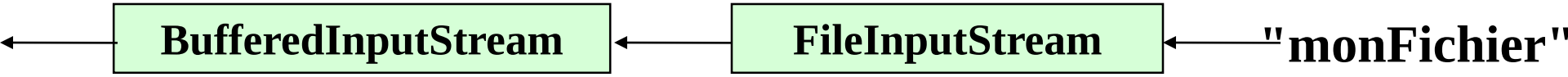


Le BufferedInputStream rempli son buffer lorsque une demande en lecture intervient alors que son buffer est vide

La taille du buffer est donnée à la création du flux.

```
public  class  java.io.BufferedInputStream extends  java.io.FilterInputStream {
// Constructors
     public BufferedInputStream(InputStream  in);/*size=8192*/
     public BufferedInputStream(InputStream  in, int size);
// Methods
……}
```

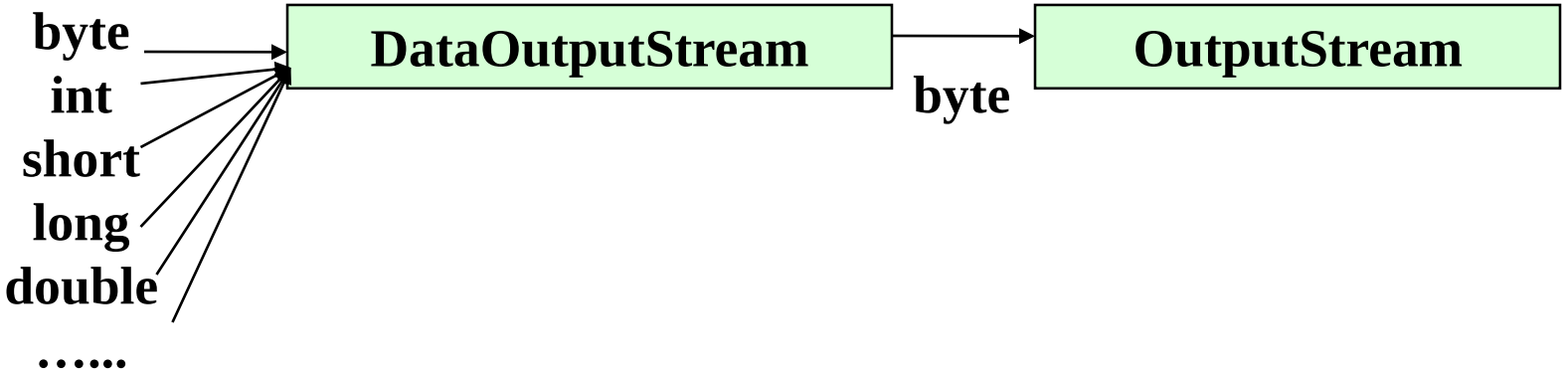
BufferedInputStream




```
// on considère notre fichier "monFichier" avec ses 1000 octets
// création du File puis du FileInputStream
File file = new File("monFichier");
FileInputStream fis = new FileInputStream(file);
// création du BufferedInputStream lié à ce FileInputStream
BufferedInputStream bis = new BufferedInputStream(fis,512);

// l'instruction suivante lit 1000 octets à partir du BufferedInputStream
// Seuls deux accès disques seront effectués : le premier lors du premier
appel à read (le buffer sera alors rempli avec 512 octets), et le second au 513
ème appel (le buffer sera alors rempli avec  les 488 octets restants)
for (int i=0;i < 1000 ; i++ ) System.out.println(bis.read( ));
// fermeture du flux
bis.close( );
```

DataOutputStream

Pour écrire des valeurs de types intrinsèques et des Strings vers un flux de sortie



Les écritures de types int , double ,… sont traduites en écriture d'octets et sont transmis au flux suivant.

Par exemple, si on demande à un dataOutputStream d'écrire un entier court, il traduira cette demande par 2 demandes d'écriture au flux suivant (un entier court occupe 2 octets)

java.io.DataOutputStream

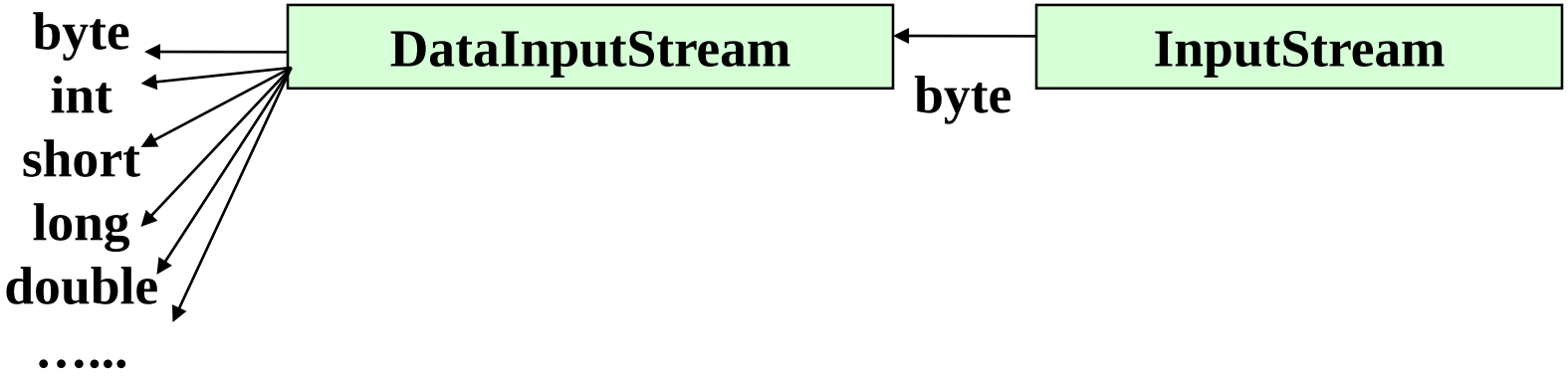
```
public  class  DataOutputStream extends  java.io.FilterOutputStream
                implements java.io.DataOutput
{
   public DataOutputStream(OutputStream  out);
   public void write(byte  b[]) throws IOException {…}
   public void write(byte  b[], int  off, int  len) throws IOException {…}
   public void writeBoolean(boolean  v) throws IOException {…}
   public void writeByte(int  v) throws IOException {…}
   public void writeBytes(String  s) throws IOException {…}
   public void writeChar(int  v) throws IOException {…}
   public void writeDouble(double  v) throws IOException {…}
   public void writeFloat(float  v) throws IOException {…}
   public void writeInt(int  v) throws IOException {…}
   public void writeLong(long  v) throws IOException {…}
   public void writeShort(int  v) throws IOException {…}
   public void writeUTF(String  str) throws IOException {…}
……
}
```

DataOutputStream : writeShort

```
// class java.io.DataOutputStream

/** écriture de l'entier court 'v' dans ce DataOutputStream */
public final void writeShort(int v) throws IOException {
// écriture de l'octet de poids fort
this.out.write((v >>> 8) & 0xFF);
// écriture de l'octet de poids faible
this.out.write( v & 0xFF);
// le nombre d'octets écrits dans ce dataOuputStream est incrémenté de deux
this.written += 2;
    }
```

DataInputStream

Pour lire des valeurs de types intrinsèques et des Strings en provenance d'un flux d'entrée



Les lectures de types int , double ,… sont traduites en lecture de byte au flux suivant puis les bytes obtenus permettent d'obtenir la valeur cherchée

java.io.DataInputStream

```
public  class  DataInputStream extends  java.io.FilterInputStream
                        implements java.io.DataInput
{
   public DataInputStream(InputStream  in){…}
   public boolean readBoolean( ) throws IOException {…}
   public byte readByte( ) throws IOException {…}
   public char readChar( ) throws IOException {…}
   public double readDouble( ) throws IOException {…}
   public float readFloat( ) throws IOException {…}
   public int readInt( ) throws IOException {…}
   public long readLong( ) throws IOException {…}
   public short readShort( ) throws IOException {…}
   public String readUTF( ) throws IOException {…}
….
}
```

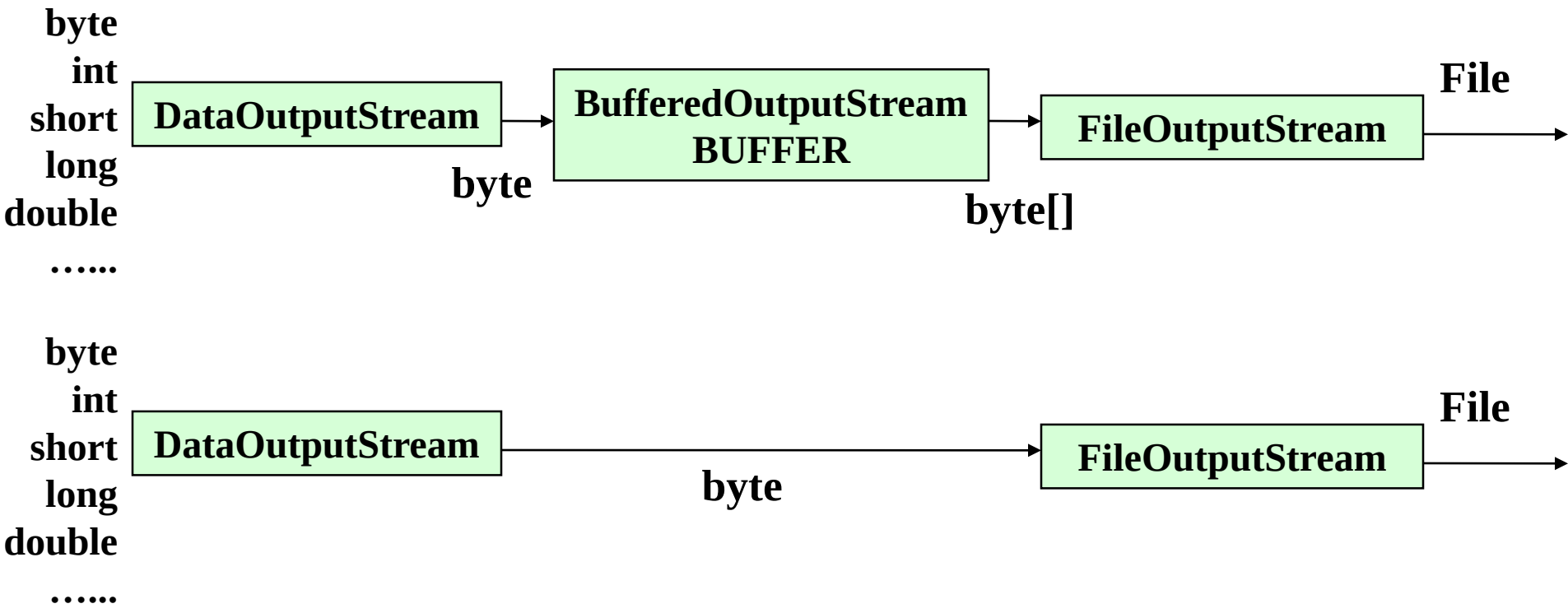
UTF-8 : code des caractères unicode sur 1 à trois octets

DataInputStream : readShort

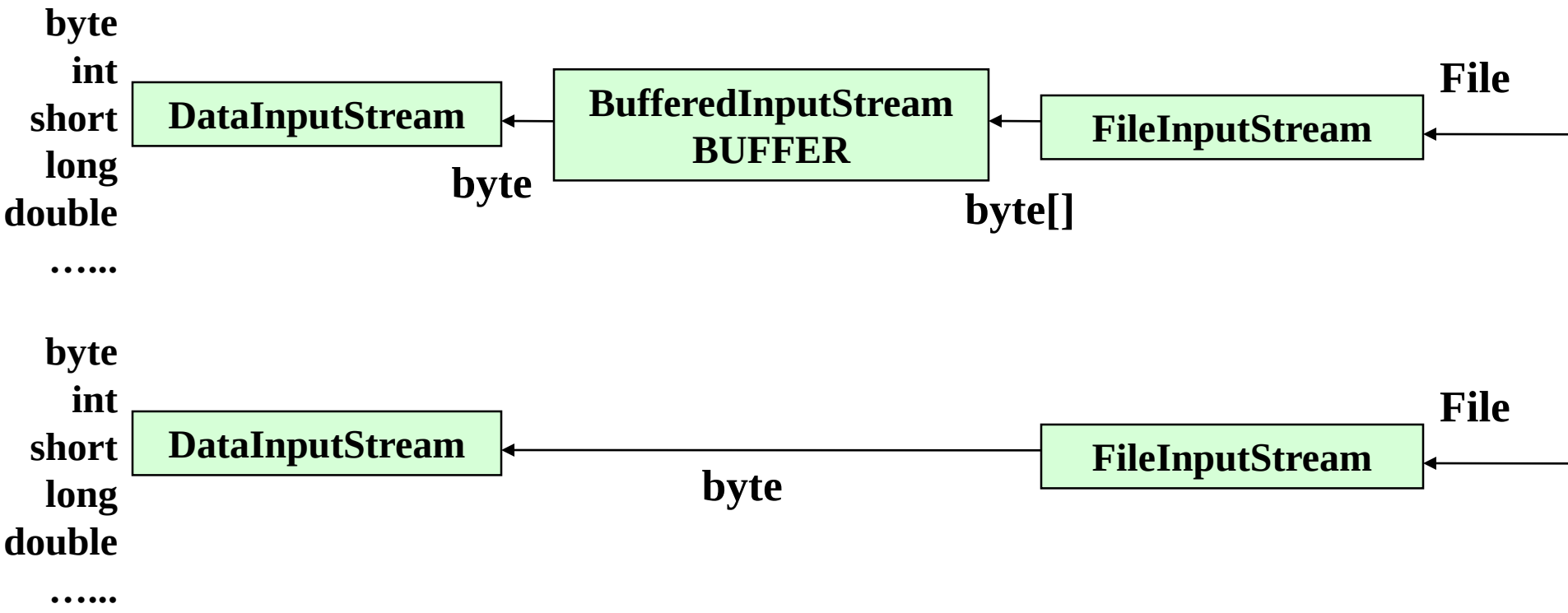
Traduction de deux octets en un entier de type 'short'

```
public final short readShort( ) throws IOException {
InputStream in = this.in;
int ch1 = in.read( );
int ch2 = in.read( );
if ((ch1 | ch2) < 0) // ((ch1==-1)||(ch2==-1)
    throw new EOFException( );
return (short)((ch1 << 8) + (ch2 << 0)); }
```

exemples de suites de flux en sortie



exemple de suites de flux en entrée



deux exemples classiques de suite de flux InputStream

Exercices : combien d'accès disque en écriture?

```
byte [ ] t = new byte[500];
File f = new File("donnees");
FileOutputStream fos = new FileOutputStream(f);
BufferedOutputStream bos= new BufferedOutputStream(fos,1024);
…...
for (int i=0;i<500;i++)bos.write(t[i]);
bos.close( );
```

```
double [ ] td = new double[500];
File f = new File("donnees");
FileOutputStream fos = new FileOutputStream(f);
DataOutputStream dos = new DataOutputStream(fos);
…...
for (int i=0;i<500;i++)dos.writeDouble(td[i]);
dos.close( );
```

```
File f = new File("donnees");
FileOutputStream fos = new FileOutputStream(f);
BufferedOutputStream bos= new BufferedOutputStream(fos,1024);
DataOutputStream dos = new DataOutputStream(bos);
…...
for (int i=0;i<500;i++)dos.writeDouble(td[i]);
dos.close( );
```

Arborescence de dérivation des classes de flux d'octets

InputStream (read, close) ***abstraite***

- ObjectInputStream : ***serialisation*** **(readObject)**
- FileInputStream ***le flux d'entrée à partir d'un fichier***
- FilterInputStream ***ajout de fonctionnalités***
  - BufferedInputStream ***bufférisation***
  - DataInputStream ***nombreuses méthodes de lecture*** **(readInt, readDouble …)**
- ByteArrayInputStream ***lecture dans un tableau de Byte***

OutputStream (write, flush, close)

- ObjectOutputStream : ***serialisation*** **(writeObject)**
- FileOutputStream ***le flux de sortie vers un fichier***
- FilterOutputStream ***ajout de fonctionnalités***
  - BufferedOutputStream ***bufférisation***
  - DataOutputStream ***nombreuses méthodes d'écriture*** **(writeInt, writeDouble)**
- PrintStream ***Pour l'affichage en mode texte***

Arborescence de dérivation des classes de flux de caractères

Reader (read)
InputStreamReader
    FileReader
BufferedReader
CharArrayReader **(lecture dans un tableau de caractères)**
StringReader **(lecture dans une chaîne)**

Writer
OutputStreamWriter
    FileWriter
PrintWriter
BufferedWriter
CharArrayWriter
StringWriter

E/S sur fichier: exemple

```
public class TestSauvegarde{
  public static void main(String [ ] args){
   Terminal term = new Terminal("Lecture et écriture sur fichier",400,400);
   File file = new File(term.readString("donner un nom"));
   try {
         if (file.exists( ))
         try (DataInputStream dis = new DataInputStream(new FileInputStream(file))) {
             term.println("1:"+dis.readUTF( ));
             term.println("2:"+dis.readUTF( ));
          }
        else
         try (DataOutputStream dos = new DataOutputStream(new FileOutputStream(file))){
             dos.writeUTF(term .readString("chaine 1?"));
             dos.writeUTF(term .readString("chaine 2?"));
          }
   }
   catch (IOException e){term .println("Erreur:"+e); }
   }
}
```

PrintStream

```
public  class  java.io.PrintStream
   extends  java.io.FilterOutputStream
{       // Constructors
   public PrintStream(OutputStream  out);
   public PrintStream(OutputStream  out, boolean  autoflush);
     // Methods
   public boolean checkError( );
   public void close( );
   public void flush( );
   public void write(byte  b[], int  off, int  len);
   public void write(int  b);              public void println( );

   public void print(boolean  b);       public void println(boolean  b);
   public void print(char  c);          public void println(char  c);
   public void print(char  s[]);        public void println(char  s[]);
   public void print(double  d);        public void println(double  d);
   public void print(float  f);         public void println(float  f);
   public void print(int  i);           public void println(int  i);
   public void print(long  l);          public void println(long  l);
   public void print(Object  obj);      public void println(Object  obj);
   public void print(String  s);        public void println(String  s);
```

System.out est un PrintStream

SERIALISATION

La sérialisation désigne l'opération qui transforme un objet en un flux d'octets permettant ainsi la sauvegarde et la transmission sur réseaux d'un objet.

L'interface java.io.Serializable

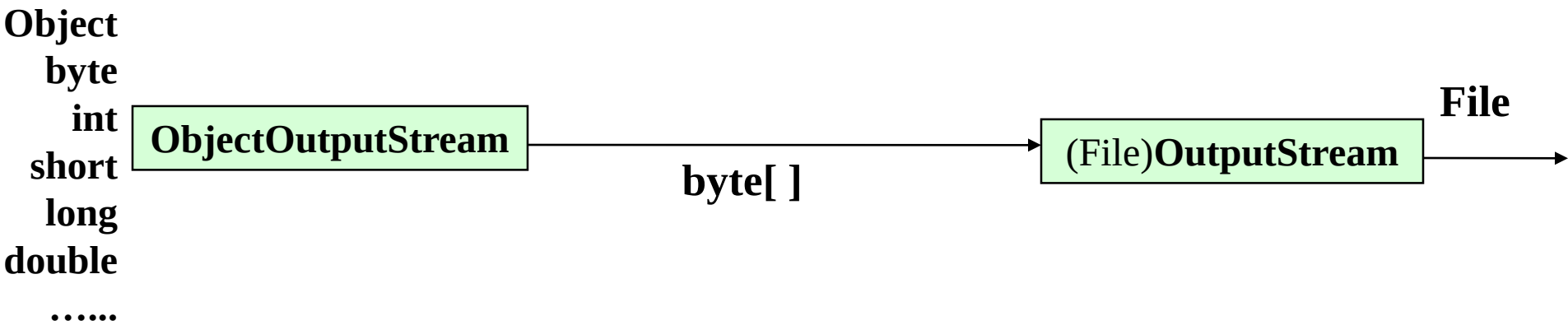
Tous les objets pouvant être sérialisés doivent être instance d'une classe implémentant l'interface java.io.Serializable.

public Compte implements Serializable {...

L' interface Serializable ne possède aucun membre et joue en fait le rôle d'un marqueur pour les classes sérialisable.

La sérialisation "standard"
- concerne uniquement les attributs d'instances
- ignore les attributs d'instance déclarés 'transient'

ObjectInputStream et ObjectOutputStream



Exemple de sérialisation d'instances de la classe Compte

```
Compte c = new Compte("Dupond);
// serialisation
File file = new File("donnees");
FileOutputStream fos = new FileOutputStream(file);
ObjectOutputStream os = new ObjectOutputStream(fos);
os.writeObject(c);
os.close( );

//……..
// deserialisation
FileInputStream fis = new FileInputStream(file);
ObjectInputStream is = new ObjectInputStream(fis);
Compte c = (Compte)is.readObject( );
is.close( );
```

ObjectInputStream

```
/**
  * @throws java.lang.ClassNotFoundException  Class of a serialized object cannot be found.
  * @throws InvalidClassException  Something is wrong with a class used by serialization.
  * @throws StreamCorruptedException  Control information in the stream is inconsistent.
  * @throws OptionalDataException  Primitive data was found in the stream instead of objects.
  * @throws IOException  Any of the usual Input/Output related exceptions.
**/
public final Object readObject( )
            throws OptionalDataException, ClassNotFoundException, IOException

// autres méthodes
public byte readByte( ) throws IOException
public short readShort( ) throws IOException
public int readInt( ) throws IOException
public long readLong( ) throws IOException
public float readFloat( ) throws IOException
public double readDouble( ) throws IOException
public char readChar( ) throws IOException
public boolean readBoolean( ) throws IOException
public String readUTF( ) throws IOException
```

ObjectOutputStream

```
/** @throws InvalidClassException   Something is wrong with a class used by serialization.
  * @throws NotSerializableException
  *          Some object to be serialized does not implement the java.io.Serializable interface.
  * @ throws: IOException   Any exception thrown by the underlying OutputStream.
public final void writeObject(Object obj) throws IOException


// autres méthodes
public void writeByte(byte b) throws IOException
public void writeShort(short s) throws IOException
public void writeInt(int i) throws IOException
public void writeLong(long l) throws IOException
public void writeFloat(float f) throws IOException
public void writeDouble(double d) throws IOException
public void writeChar(char c) throws IOException
public void writeBoolean(boolean b) throws IOException
public void writeUTF(String s) throws IOException
```

Sérialisation : exemple de sauvegarde sur Fichier

sauvegarde d'un objet

```
/** sérialise et sauvegarde un objet dans un fichier */
public void saveToFile(Object object,File file) throws IOException{
      try (FileOutputStream fos = new FileOutputStream(file);
           ObjectOutputStream oos = new ObjectOutputStream(fos);)
          {
            oos.writeObject(object);
           }
}
```

récupération d'un objet

```
/** récupère un objet qui a été sérialisé dans un fichier */
public Object loadFromFile(File file) throws IOException,
                                    ClassNotFoundException{
      try (FileInputStream fis = new FileInputStream(file);
           ObjectInputStream ois = new ObjectInputStream(fis);)
       {
           return ois.readObject( );
       }
 }
```

Exemple (suite)

Déterminer le nom du fichier

```
public static void main(String[ ] args) throws IOException{
      Terminal term = new Terminal("Sérialisation",400,400);
String nomTit = term .readString("nom du titulaire du compte ?");
File fichierSauvegarde=new File(nomTit);
```

restauration (ou création) de l'objet (ou des objets)

```
Compte compte;
try {  compte = loadFromFile(fichierSauvegarde);}
catch (Exception e){compte = new Compte(nomObj);};
```

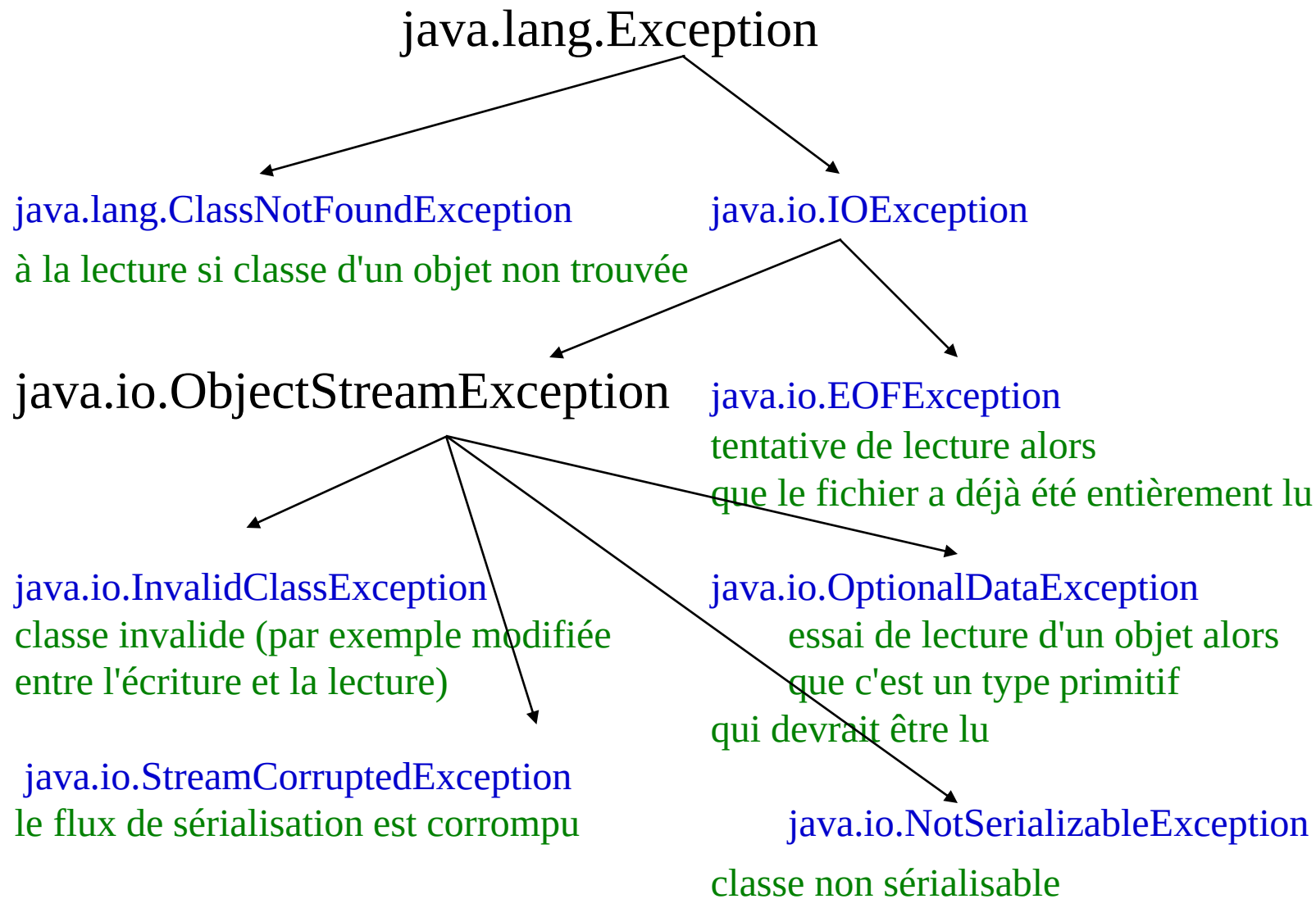
le programme

```
// ICI se trouvent les instructions centrales du programme
// par exemple un menu pour consulter le solde, ajouter des opérations
```

sauvegarde de l'objet ou des objets

```
try {saveToFile(compte, fichierSauvegarde);}
catch (Exception e){term.println("Erreur : "+e);}
}
```

Les classes d'exceptions pour la sérialisation



Sérialisation "manuelle"

Les données membres sont toutes sauvegardées sauf:
- les données membres de classe (modificateur 'static')
- les données membres d'instances déclarées avec le modificateur 'transient'

Si pour une classe donnée, on désire indiquer comment la sérialisation doit se faire, on doit définir dans cette classe les deux méthodes suivantes:

```
private void writeObject(java.io.ObjectOutputStream sortie) throws IOException
// appel éventuel dans cette méthode de sortie.defaultWriteObject( )
// pour appeler le mécanisme de sérialisation par défaut
```

```
private void readObject(java.io.ObjectInputStream entree) throws IOException
// appel éventuel dans cette méthode de entree.defaultReadObject( )
// pour appeler le mécanisme de sérialisation par défaut
```