Les fichiers

Master WISD

AU: 2020/2021

Fichiers en java

En java, les entrées/sorties (fichiers en particulier) sont représentés par des objets particuliers appelés flots (**Stream** en anglais).

Le principe de traitement est simple et se résume comme suit :

- Ouverture d'un canal de communication.
- Écriture ou lecture des données.
- Fermeture du canal.

Entrées: Fichier, clavier, modem, capteur, …

Sorties: Fichier, écran, imprimante, …

Fichiers en java

Il existe de nombreuses sortes de stream (flot), qui peuvent être classés selon plusieurs critères (package java.io):

- Les streams d'entrées (lecture) et les streams de sortie (écriture).
- Les streams de caractères (texte) et les streams de données binaires.
- Les streams de traitement des données et les streams de communication de données.
- Les streams à accès séquentiel et les streams à accès direct.
- Les streams avec et sans tampon de données.

Classes de gestion des fichiers

- **InputStream**
- **OutputStream**

} Lecture et écriture de flots d'octets

- **Reader**
- **Writer**

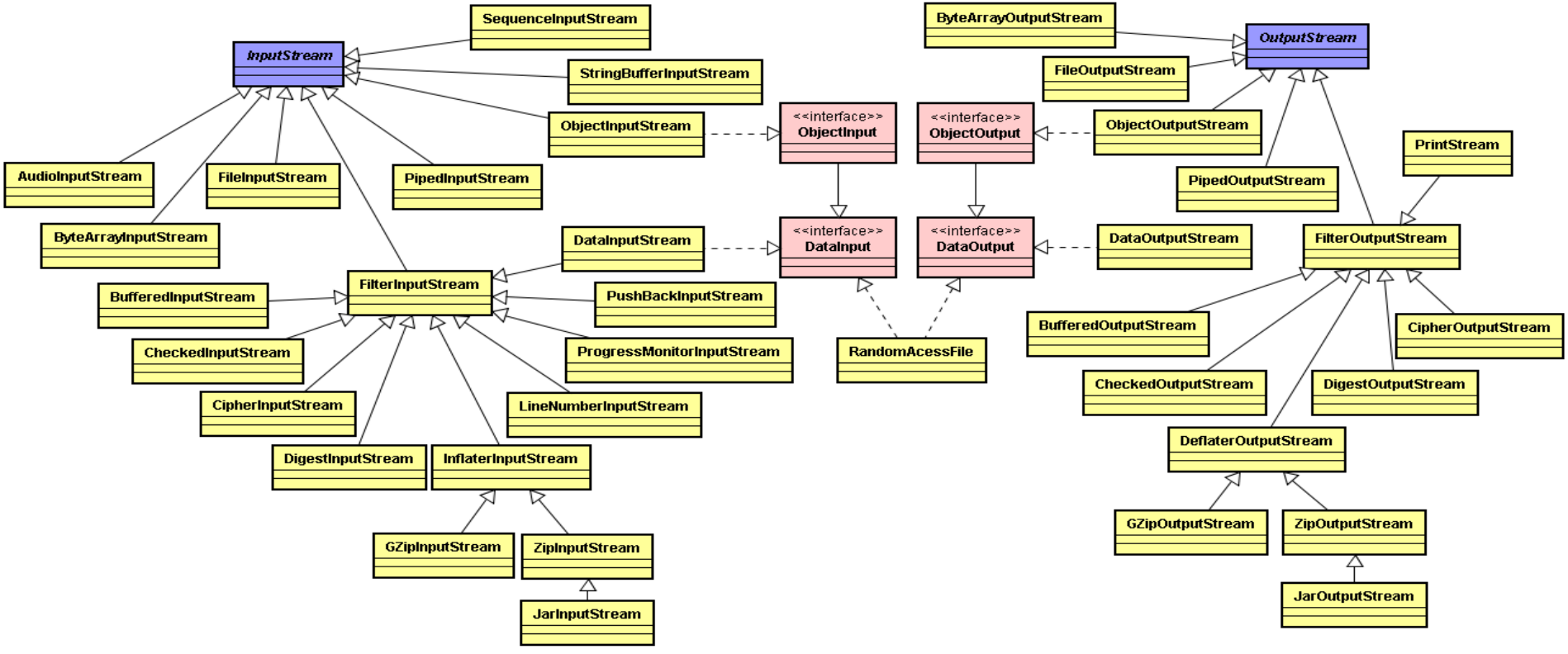
} Lecture et écriture de flots de caractères Unicode

- **StreamTokenizer** Analyse lexicale d'un flot d'entrée
- **File** Pour représenter fichiers et répertoires

Fichiers binaires

- Les flots de données binaires (ou flots d'octets) peuvent contenir d'autre type de données que des caractères (des int par exemple).
- En java, les streams de données binaires dérivent des deux classes (du package java.io): **InputStream** pour les entrées et **OutputStream** pour les sorties.
- Un flot peut être :
  - Soit une source d'octets à partir de laquelle il est possible de lire de l'information. → flot d'entrée.
  - Soit une destination d'octets dans laquelle il est possible d'écrire de l'information. → flot de sortie.

Fichiers binaires



Ecriture dans un fichier binaires

L'écriture dans un fichier binaire se fait grâce aux streams d'écriture pour les données binaires.

En java, les streams d'écriture pour les données binaires sont des sous classes de la classe java.io.**OutputStream**.

- **DataOutputStream**: écriture séquentielle dans un fichier binaire.
- **BufferedOuputStream**: écriture des données à l'aide d'un tampon.
- **PrintStream**: écriture de données avec conversion en octets.
- **ZipOutputStream**: permet d'écrire des fichiers dans le format de compression zip.
- **ObjectOutputStream**: écrire des objets dans un flux binaire (sérialisation).

Exemple

```
class CreerFichierBinaire {
    public static void main(String [] arg) throws IOException {
        FileOutputStream fos= new FileOutputStream("fichier.dat");
        DataOutputStream stream= new DataOutputStream(fos);
        stream.writeUTF("Bonjour monde java");
        stream.writeInt(100);
        stream.writeFloat(123,456);
        stream.writeBoolean(true);
        System.out.println(stream.size());
        stream.close();
    }
}
```

Lecture d'un fichier binaires

La lecture d'un fichier de données binaires se fait par des flux d'entrées pour les données binaires.

Les streams d'entrées pour les données binaires sont des sous classes de la classe java.io.**InputStream**.

- **DataInputStream:** lecture séquentielle dans un fichier binaire.
- **BufferedInputStream:** lecture de données à l'aide d'un tampon.
- **ZipInputStream**: lire des fichiers dans le format de compression zip.
- **ObjectInputStream**: lire des objets dans un stream binaire (désérialiser un objet précédemment sérialisé)

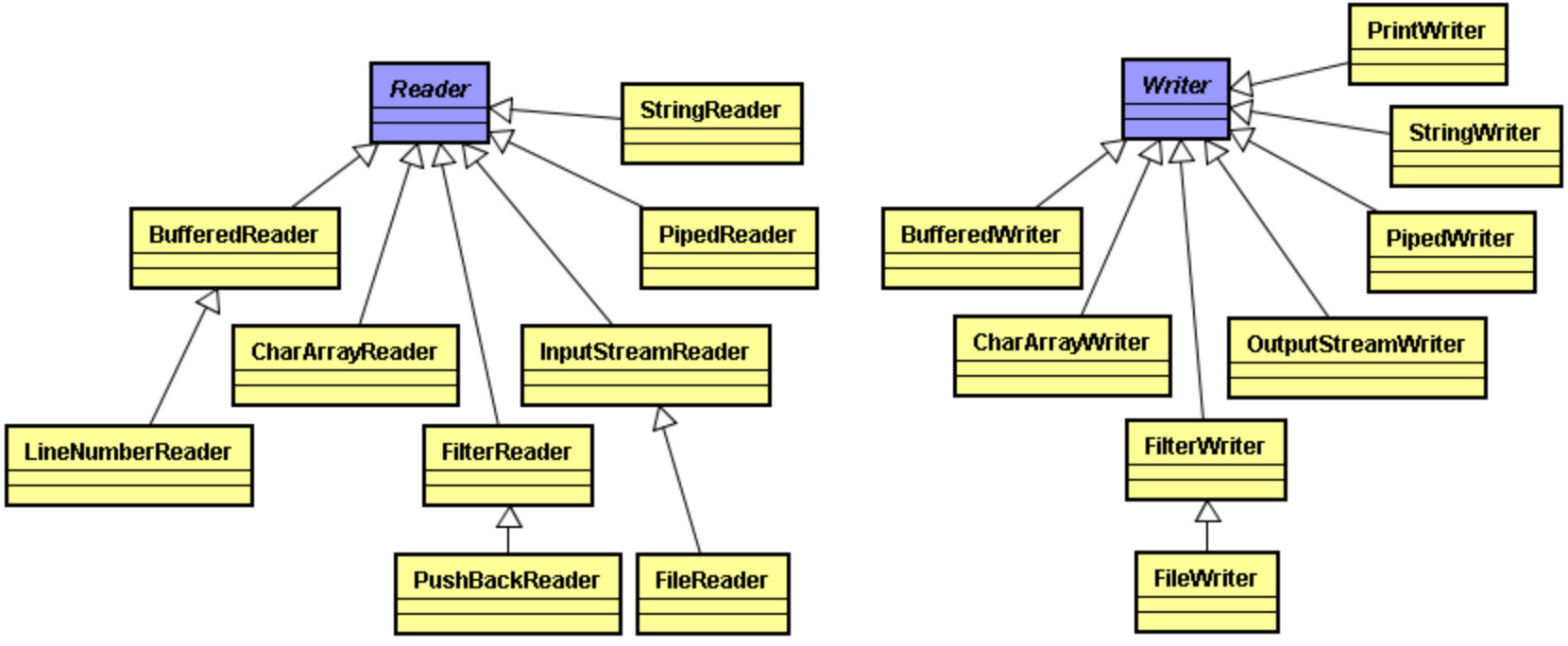
Exemple

```
class LectureFichierBinaire {
    public static void main(String [] arg) throws IOException {
        DataInputStream stream;
        stream= new DataInputStream(new FileInputStream ("fichier.dat"));
        //Supposons que fichier.dat se trouve à la racine du projet.
        System.out.println("La taille du fichier est: "+stream.available()+" octets");
        System.out.println(stream.readUTF());
        System.out.println(stream.readInt());
        System.out.println(stream.readFloat());
        System.out.println(stream.readBoolean());  stream.close();
    }
}
```

Fichiers textes

- Les flots (streams) de caractères sont conçus pour la lecture et l'écriture de texte.
- Les flots de caractères dérivent des deux classes abstraites: **Reader** pour les entrées et **Writer** pour les sorties.

Fichiers textes



Ecriture dans un fichier texte

Pour écrire dans un fichier texte, il faut utiliser les streams d'écriture qui sont des sous-classes de la classe **Writer**, qui contient les méthodes abstraites suivantes:

- write(char[] t, int id, int n), qui permet d'écrire *n* caractères à partir du tableau *t*, en commençant à l'indice *id*.
- write(String, int, int): écrire une portion d'une chaîne de caractères.
- close(), qui ferme le stream.
- …

Toutes les classes dérivées de **Writer** redéfinissent donc obligatoirement ces méthodes.

Ecriture dans un fichier texte

- **FileWriter**: sous classe particulière de OutputStreamWriter utilisant la taille du tampon par défaut. Cette classe convient dans la plupart des cas d'écrire dans un fichier de caractères.
- **OutputStreamWriter**: permet la conversion d'un stream d'octets (de données binaires) en un stream de caractères.
- **BufferedWriter**: écriture de caractères à l'aide d'un tampon.
- **PrintWriter**: permet d'écrire des caractères. Elle est particulièrement utilisée pour l'affichage en mode texte.

La méthode println() de la classe PrintWriter permet d'écrire sur le fichier texte. Cette méthode est surchargée et peut avoir un argument de type String, char, int, …

Exemple

```
class CreerFichierTexte {
    public static void main(String [] arg) throws IOException {
            PrintWriter ecrire;
            ecrire= new PrintWriter (new FileOutputStream("fichier.txt"));
            ecrire.println("Bonjour");
            ecrire.println(100);
            ecrire.println(true);
            ecrire.println(10.55);
            ecrire.close();
    }
}
```

Lecture d'un fichier texte

Pour lire dans un fichier texte, on doit utiliser des sous classes de la classe abstraite Reader.

Quelques méthodes de cette classe:

- read(): renvoie le caractère lu ou -1 si la fin du flux est atteinte.
- read(char []t, int id, int n): permet de lire n caractères et de les placer dans le tableau t, à partir de l'indice id.
- close(): ferme le stream.

Lecture d'un fichier texte

- **FileReader**: sous classe particulière de InputStreamReader utilisant une taille de tampon par défaut. Cette classe convient dans la plupart des cas de lecture d'un fichier de caractères.
- **BufferedReader**: lecture de caractères à l'aide d'un tampon. Les caractères peuvent ainsi être lus en bloc.

Exemple 1

```
BufferedReader stream=null;
String str;
try{
        stream=new BufferedReader(new FileReader("fichier.txt"));
        while((str=stream.readLine())!=null) // retourne null si rien a lire
        System.out.println(str);
}
catch(FileNotFoundException exc){
        System.out.println("Erreur d'ouverture du fichier!!");
}
finally{
        stream.close();
}
```

Exemple 2

```
FileReader fiche=new FileReader("fichier.txt");
BufferedReader flux=new BufferedReader(ficALire);
int car = flux.read(); //lecture du 1er caractère
while (car != -1) { //-1 si fin du fichier
        System.out.print((char) car);
        car = flux.read();
}
flux.close();
fiche.close();
}
```

```
while (flux.ready()) {
    car = flux.read();
    System.out.print((char) car);
}
```

Exemple 3

```
FileWriter ecrire;
char tmp[]={'u','n','e',' ','c','h','a','i','n','e',' ','d','e',' ','c','a','r','a','c','t','è','r','e','s'};
ecrire= new FileWriter ("fichier.txt");
ecrire.write(tmp);
ecrire.write(tmp,5,8);
ecrire.close();
```

Exemple 4

```
FileReader lire;
char tmp[]=new char[1024];
int nbcar;
lire= new FileReader ("fichier.txt");
nbcar=lire.read(tmp);
for(int i=0;i<nbcar;i++)
        System.out.print(tmp[i]);
lire.close();
```

Classe File

Cette classe donne une représentation objet d'un fichier ou d'un répertoire. Ses méthodes donnent beaucoup d'informations sur les fichiers (la taille, date de modification, fichier ou répertoire, …).

Constructeurs:

- *File(String ch): ch* est le chemin d'accès
- *File(String ch, String nom): ch* est le chemin complet d'accès, *nom* est le nom du fichier.

Classe File

Quelques méthodes:

canRead(): renvoie true si le fichier est accessible en lecture

canWrite(): renvoie true si le fichier est accessible en écriture

isDirectory(): retourne true si le chemin indiqué existe

isFile(): renvoie true si le fichier indiqué existe

list(): retourne le contenu d'un répertoire.

length(): la taille d'un fichier

lastModified(): renvoie la date et l'heure de la dernière modification

renameTo(File): renomme un fichier ou un répertoire

getParent(): retourne le nom du répertoire parent.

Classe File

Autres méthodes de cette classe :

- boolean delete()
- boolean exists()
- boolean mkdir()
- boolean mkdirs()
- String getName( )
- boolean isHidden()
- boolean setLastModified()

Classe File

Création de fichiers en utilisant la classe File

```
File file = new File (CheminDuFichier);
```

- Création de FileReader avec un objet File

```
FileReader fichier = new FileReader(file);
```

- Création de FileWriter:

```
FileWriter fichier = new FileWriter(file);
```

Exemple

```
String nomFichier;
File repertoire=new File(".");
if(repertoire.isDirectory()){
    String [] tab=repertoire.list();
    for(int i=0;i<tab.length;i++){
        nomFichier=tab[i];
        File fiche=new File(nomFichier);
        if(fiche.isFile()){
            Date dt=new Date(fiche.lastModified());
            long taille=fiche.length();
            System.out.println(nomFichier+ "\t "+ taille+ "\t"+dt);
        }
    }
}
```

Classe StreamTokenizer

- La classe StreamTokenizer permet de découper un flux en entrée en unités syntaxiques telles que des mots et des nombres.
- Attributs de cette classe:
  - *double nval*: valeur d'une unité numérique
  - *String sval*: chaîne contenant les caractères d'un mot
  - *int ttype*: contient le type de l'unité syntaxique lue.
  - *int TT_EOF*: constante indiquant de la fin du flux est atteinte
  - *int TT_EOL*: constante indiquant de la fin d'une ligne est atteinte
  - *int TT_NUMBER*: constante indiquant qu'un nombre a été lu
  - *int TT_WORD*: constante indiquant qu'un mot a été lu

StreamTokenizer

Quelques méthodes:

- *StreamTokenizer(Reader)*
- *StreamTokenizer(InputStream)*
- *int lineno():* renvoie le numéro de la ligne en cours
- *nextToken():* recherche l'unité suivante
- *ordinaryChars(int):* le caractère passé en paramètre devient un caractère ordinaire.

Exemple

```
FileReader fiche=new FileReader ("fichier.txt");
StreamTokenizer lire=new StreamTokenizer(fiche);
while(lire.nextToken()!=StreamTokenizer.TT_EOF){
        if(lire.ttype==StreamTokenizer.TT_WORD)
                System.out.print(lire.sval);
}
fiche.close();
```

Classe RandomAccessFile

- La classe RandomAccessFile permet d'accéder à un fichier en mode écriture et lecture à la fois, comme elle permet de se déplacer dans un fichier avec la méthode seek.

- La classe RandomAccessFile admet les constructeurs:
  - RandomAccessFile(File fichier, String mode)
  - RandomAccessFile(String nom, String mode)

Classe RandomAccessFile

Quelques méthodes de la classe RandomAccessFile:

int readInt(), float readFloat(), char readChar(), …

int read(byte [] t, int id, int n): lit jusqu'à n octets vers le tableau t.

long getFilePointer(): renvoie la position du pointeur.

seek(long pos): positionne le pointeur à la position indiquée.

void close(): ferme le flux de fichier.

void writeInt(int), writeDouble(double), …

void write(byte[] t, int id, int n): écrit n octets depuis le tableau t vers le fichier en commençant de la position id.

Exemple

```
File ch=new File(".","fiche.txt");
RandomAccessFile fiche=new RandomAccessFile(ch,"rw");
int nb;
for(int i=0;i<10;i++)
        fiche.writeInt(i*i);
fiche.seek(8);
fiche.writeInt(100);
fiche.seek(0);
for(int i=0;i<10;i++){
        nb=fiche.readInt();
        System.out.println(nb);
}
```

Compression ZIP

- La classe java.util.zip.**ZipOutputStream** permet d'écrire un fichier ZIP
- Un objet java.util.zip.**ZipEntry** doit être créé pour chacune des entrées du fichier ZIP.
- la méthode ***putNextEntry***() du flot **ZipOutputStream** consiste à commencer à écrire dans un nouveau fichier.
- Les données du fichier doivent être envoyer dans le flux ZIP.
- répéter la même chose pour chacun des fichiers à archiver.

Exemple 1

```
ZipOutputStream archive = new ZipOutputStream(new
                                                FileOutputStream("archive.zip"));
PrintWriter fichier= new PrintWriter(archive, true);
archive.putNextEntry(new ZipEntry("Premier.txt"));
fichier.println("Il s'agit juste")
fichier.println("d'un premier texte");
fichier.println("qui va être compressé.");
archive.putNextEntry(new ZipEntry("Deuxieme.txt"));
fichier.println("Le deuxième texte");
fichier.println("est également compressé.");
archive.close();
```

Exemple 2

```
File f1=new File(".","Fichier.doc");
FileInputStream fis=new FileInputStream(f1);
BufferedInputStream bis=new BufferedInputStream(fis);
File f2=new File("Fichier.zip");
FileOutputStream fos=new FileOutputStream(f2);
BufferedOutputStream bos=new BufferedOutputStream(fos);
ZipOutputStream zos=new ZipOutputStream(bos);
zos.setMethod(ZipOutputStream.DEFLATED);
zos.putNextEntry(new ZipEntry(f1.getName()));
int c;
while ((c=bis.read())!=-1)
        zos.write(c);
zos.close();bis.close();
```

Décompression ZIP

- La classe java.util.zip.**ZipInputStream** permet de lire en Java un fichier ZIP.
- Pour lire dans un java.util.zip.**ZiptInputStream**, on utilise la méthode *getNextEntry()* qui renvoie un objet java.util.zip.**ZipEntry** qui représente le fichier compressé. Cette méthode retourne null si il ne reste plus d'élément à lire.
- La méthode *read()* de **ZipInputStream** a été modifiée pour renvoyer -1, non pas à la fin du fichier ZIP, mais à la fin de l'entrée courante. Il faut appeler la méthode *closeEntry()* pour pouvoir passer à la prochaine entrée.
- La classe java.util.**Scanner** peut être utilisée pour lire un fichier texte se trouvant dans un fichier ZIP.

Exemple 1

```
ZipInputStream archive = new ZipInputStream(new
                                        FileInputStream("archive.zip"));
ZipEntry fichier;
while ((fichier = archive.getNextEntry()) != null) {
    Scanner lire = new Scanner(archive);
    out.println("Fichier : "+fichier.getName());
    while (lire.hasNextLine())
        out.println(lire.nextLine());
    archive.closeEntry();
}
archive.close();
```

Exemple 2

```
File f=new File("Fichier.zip");
FileInputStream fis=new FileInputStream(f);
BufferedInputStream bis=new BufferedInputStream(fis);
ZipInputStream zis=new ZipInputStream(bis);
ZipEntry ze=zis.getNextEntry();
File f2=new File(ze.getName());
FileOutputStream fos=new FileOutputStream(f2);
BufferedOutputStream bos=new BufferedOutputStream(fos);
int c;
while ((c=zis.read())!=-1)
            bos.write(c);
zis.close();  bos.close();
```

Flots de données prédéfinis

Il existe 3 flots prédéfinis :

- l'entrée standard **System.in** (instance de la classe InputStream)
- la sortie standard **System.out** (instance de PrintStream)
- la sortie standard d'erreurs **System.err**(instance de PrintStream)

```
try {
    int c;
    while((c = System.in.read()) != -1)
        System.out.print(c);
}
catch(IOException e) {
    System.out.print(e);
}
```

Exemple de lecture au clavier

```
class DemoLireClavier{
    public static void main(String [] arg) throws IOException
    {
        BufferedReader lire=new BufferedReader(new
                                        InputStreamReader(System.in));
        String ligne;
        System.out.println("Entrer une chaîne de caractère");
        ligne=lire.readLine();
        System.out.println("Votre chaîne est:");
        System.out.println(ligne);
        lire.close();
    }
}
```

Sérialisation

- La sérialisation consiste à prendre un objet en mémoire et à en sauvegarder l'état sur un flux de données (vers un fichier, par exemple).
- Ce concept permet aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement.
- La sérialisation peut donc être considérée comme une forme de persistance des données.

Sérialisation

- Les deux classes ObjectInputStream et ObjectOutputStream proposent, respectivement, les méthodes readObject et writeObject.
- Par défaut, les classes ne permettent pas de sauvegarder l'état d'un objet sur un flux de données. Il faut implémenter l'interface java.io.Serializable.
- L'interface Serializable ne possède aucun attribut et aucune méthode. Mais elle sert uniquement à identifier une classe sérialisable.
- Tous les attributs de l'objet sont sérialisés mais à certaines conditions :
  - être lui-même sérialisable ou être un type primitif
  - ne pas être déclaré static ni transient
- Il faut que la classe n'ait pas supprimé le constructeur par défaut.

Exemple

```
void sauvegarde(String s) {
   try {
      FileOutputStream fos = new
         FileOutputStream(new File(s));
      ObjectOutputStream oos = new
         ObjectOutputStream(fos);
      oos.writeObject(this);
      oos.close();
   }
   catch (Exception e){
      System.out.println("Erreur : "+e);
   }
}
```

```
static Object lecture(String s) {
   try  {
      FileInputStream fis = new
         FileInputStream(new File(s));
      ObjectInputStream ois = new
         ObjectInputStream(fis);
      Object obj=ois.readObject();
      ois.close();
      return obj;
   }
   catch (Exception e){
      System.out.println("Erreur: "+e);
      return null;}
}
```

Gestion d'exception

Utilisation de try-with-resources (à partir de java SE 7)

Exemple:

```
try ( FileReader lire= new FileReader("fich1.txt" ) ;
      FileWriter ecrire= new FileWriter ("fich2.txt" )  ){
            int car= lire.read() ;
            while ( car != −1) {
                ecrire.write ( car ) ;
                car= r.read( ) ;
            }
}catch(Exception e){ System.err.println(e.getMessage()); }
```

- Instructions d'initialisation des ressources séparées par des ';'
- La méthode close() est automatiquement appelée à la fin du bloc.

→ Pas besoin du bloc *finally* … → moins du code

Gestion de plusieurs exceptions

Avec un seul bloc catch, on peut faire le traitement de plusieurs types d'exceptions (séparés par un pipe '|').

```
try {
            //…..
}
catch (ClassNotFoundException | IOException| IllegalAccessException e) {
        System.err.println("Erreur: ", e.getMessage());
}
```

Package java.nio

Les objets du package java.io traitaient les données par octets. Par contre, les objets du package java.nio (signifie New I/O) les traitent par blocs de données (traitement accéléré).

Java 7 propose une nouvelle API NIO.2 pour la gestion et l'accès au système de fichier.

Quelques limites de la classe File de java.io:

- gestion des exceptions : impossibilité d'obtenir la moindre information sur l'origine de quelques problèmes
- accès limité aux propriétés des fichiers (propriétaire, droits, …)
- absence de certaines fonctionnalités (copie et déplacement de fichier, gestion des liens, …)

➔ La classe File est remplacée par une nouvelle classe Path (package java.nio.file)

Exemple

```
FileInputStream fis;
FileChannel fc;
try {
        fis = new FileInputStream(new File("text1.txt")); //Ouvrir le fichier
        fc = fis.getChannel();//Récupérer le canal
        int size = (int)fc.size();//la taille du fichier en octet
        ByteBuffer bBuff = ByteBuffer.allocate(size);//le tampon correspondant à cette taille
        fc.read(bBuff);//Lire les octets dans le tampon
        byte [] tab=bBuff.array(); //convertir le tampon en un tableau
        for(byte bit : tab)   System.out.print((char)bit ); //Affichage
        System.out.println("La taille du fichier est: "+size+" octets");
        }
catch (…) {
}
```

Gestion des chemins

- Un chemin d'accès à une ressource fichier ou répertoire peut être absolu ou relatif.
- À partir de java 7, l'interface java.nio.file.**Path** représente le chemin.
- Avantages de Path
  - accès facile aux éléments d'un chemin;
  - comparaison de chemins;
  - possibilité d'être averti lorsque un fichier est modifié (surveillance).
- Quelques méthodes:
  - equals, compareTo, startsWith et endsWith : comparaison
  - normalize: remplacer « . » et « .. » dans les chemins
  - toFile: créer un File depuis un Path

Gestion des chemins

- La classe java.nio.file.**Paths** est souvent utilisée pour créer des instances **Path**, à partir d'une URI ou d'une chaîne de caractères.

Exemples:

```
Path path1 = Paths.get("d:");
Path path2 = Paths.get(System.getProperty("user.home"));
Path path3 = Paths.get(System.getProperty("user.dir", "fichier.txt"));
```

- La classe java.nio.file.**Files** contient des méthodes statiques pour manipuler des objets de type **Path**.

Accès aux fichiers

La classe **java.nio.file.Files** contient des méthodes permettant:

- la manipulation des chemins
- création et suppression des fichiers et des dossiers
- copie et déplacement des fichiers et dossiers
- vérification et gestion des droits d'accès
- existence des fichiers
- création d'objets flots
- parcours d'arborescence
- méthodes simplifiées pour la lecture et l'écriture.

Parcourir les répertoires

```
Path path = Paths.get("C:/Windows/");
DirectoryStream<Path> flut = Files.newDirectoryStream(path);
try {
    Iterator<Path> iterator = flut.iterator();
    while(iterator.hasNext()) {
        Path pt = iterator.next();
        System.out.println(pt);
    }
}
finally {
    flut.close();
}
```

Parcourir les répertoires

```
Path path = Paths.get(System.getProperty("user.home"));
 //avoir  et afficher tous les fichiers du répertoire
try (DirectoryStream<Path> flut= Files.newDirectoryStream(path)) {
                    for (Path elem : flut)
                                System.out.println(elem);
  }
//appliquer un filtre (pas tous les fichiers en utilisant la notation  des Exp Reg)
try (DirectoryStream<Path> flut= Files.newDirectoryStream(path, "*.{java,jsp,xml}")) {
     for (Path elem : flut)
          System.out.println(elem);              //ou System.out.println(fich.getFileName());
}
```

Lecture d'un fichier

- Lecture séquentielle du contenu d'un fichier: instancier FileReader ou FileInputStream.
- Files fournit des méthodes pour simplifier les opérations de manipulation de fichiers.

Exemple:

Parcourir le contenu d'un fichier à l'aide de la méthode **newBufferedReader.**

```
Path pt= Paths.get("d:\\java\\src\\DemoFile.java");
try (BufferedReader br = Files.newBufferedReader(pt)) {
    String ligne = null;
    while ((ligne = br.readLine()) != null) {
        System.out.println(ligne);
    }
}
```

Lecture d'un fichier

- Lecture des petits fichiers:
  - Binaires: readAllBytes
  - Textuels: readAllLines

Exemple:

```
Path pt= Paths.get("d:\\java\\src\\DemoFile.java");
List<String> lignes = Files.readAllLines(pt);
for (String elem : lignes) {
      System.out.println(elem);
}
```

Ecriture dans un fichier

- La classe Files offre des méthodes : newOutputStream, newInputStream, newBufferedWriter, …

- **Exemple:**

```
Charset utf8 = Charset.forName("UTF-8");
//ou utiliser StandardCharsets.UTF_8
Path pt= Paths.get("d:\\fiche.txt");
String contenu= "Nom et prénom: Toto Titi\nÂge: 24\nDiplôme: BAC\n";
try (BufferedWriter bw= Files.newBufferedWriter(pt,utf8,
                                    StandardOpenOption.CREATE) ) {
        bw.write(contenu);
}
```

Afficher les partitions

Exemple 1: afficher les partitions de sa machine

```
Iterable<Path> lecteurs = FileSystems.getDefault().getRootDirectories();
for (Path nom: lecteurs ) {
        System.out.println(nom);
}
```

Afficher les partitions

Exemple 2: avec plus de détails

```
try{  FileSystem defaultFS = FileSystems.getDefault();
      Iterable<Path> lecteurs = defaultFS.getRootDirectories();
      for (Path lecteur : lecteurs ) {
         System.out.println(lecteur);
         FileStore fs= Files.getFileStore(lecteur);
         long et=fs.getTotalSpace(),  ed=fs.getUsableSpace() ;
         System.out.printf("Taille (Go) : %.2f\n", et/1073741824.0);
         System.out.printf("Espace disponible (Go) : %.2f\n", ed/1073741824.0);
         System.out.println("_____________________________");
      }
}
catch(Exception e){   }
```