Entrées/Sorties

Introduction

- Les E/S de Java passent par des *flux* (streams).
- Un flux est une abstraction qui *produit* ou *consomme* des données.
- Il est lié à un dispositif physique par le système d'E/S de Java.
- Quel que soit ce dispositif physique, le flux fonctionne de la même façon : on peut donc utiliser les mêmes classes et les mêmes méthodes indépendamment du support (disque, socket, console, etc.)
- Les flux sont implémentés par des classes du paquetage `java.io`.
- Il existe également un paquetage `java.nio` (`new IO`) pour les E/S par canal.

Flux d'entrée-sortie

- Depuis sa version 1.1, Java distingue deux sortes de flux : les *flux d'octets* et les *flux de texte* (ces derniers ne sont qu'un moyen commode de gérer les caractères – le mécanisme de base reste le flux d'octets).
- Le paquetage *java.io* fournit 4 classes *abstraites* :
  - *InputStream* pour la lecture de fichiers ou de tableaux d'*octets*
  - *OutputStream* pour l'écriture de fichiers ou de tableaux d'*octets*.
  - *Reader* pour la lecture de fichiers ou de tableaux de caractères Unicode (flux texte).
  - *Writer* pour l'écriture de fichiers ou de tableaux de caractères Unicode (flux texte).
- Il fournit également la classe *File*, qui permet de gérer les *noms* de fichiers ou de répertoires en offrant des méthodes de tests, de suppression, de renommage, de listing de répertoires, etc. Cette classe n'est pas abstraite : elle permet de manipuler les fichiers/répertoires en faisant abstraction du SGF sous-jacent.

Flux d'entrée-sortie

- Le paquetage *java.io* définit également une hiérarchie des exceptions liées aux opérations d'entrées-sortie à l'aide d'un ensemble de classes dérivant de *IOException*. Toutes ces exceptions sont *contrôlées* et doivent donc être annoncées avec `throws`.
- Java 7 introduit la notion de *flux à fermeture automatique* afin de simplifier le processus de fermeture des fichiers ouverts par un programme (voir l'interface *java.io.Closeable* et *java.lang.AutoCloseable*).

La classe File : exemple de suppression de fichier ou répertoire

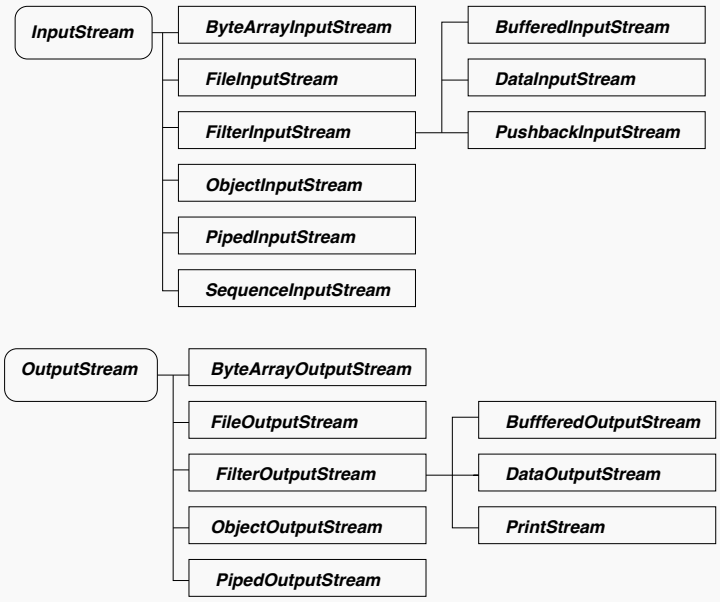
```
// Contenu du fichier Delete.java
import java.io.*;
public class Delete {
  public static void main(String[] args) {
     if (args.length != 1) {
       System.err.println("Usage: java Delete <fichier ou répertoire>");
       System.exit(1);
     }
     try {
       delete(args[0]);
     } catch (IllegalArgumentException e) {
       System.err.println(e.getMessage()); }
  } // main

  private static void delete(String nom) {
    File f = new File(nom);
    if (!f.exists())    echec("Delete: fichier ou répertoire inexistant: " + nom);
    if (!f.canWrite())  echec("Delete: fichier ou répertoire protégé en écriture: "  + nom);
    if (f.isDirectory()) {
       String[] fichiers = f.list();  // Contenu du répertoire
       if (fichiers.length > 0)  echec("Delete:  répertoire non vide: " + nom);
    }
    // On peut donc y aller...
    boolean ok = f.delete();
    if (!ok) echec("Delete: échec de la suppression");
  } // delete

  private static void echec(String msg) throws IllegalArgumentException {
    throw new IllegalArgumentException(msg);
  }
} // class Delete
```


Les classes de java.nio.file : exemple

La même chose avec la nouvelle API `java.nio` :

```
// Contenu du fichier Delete2.java
import java.io.*;
import java.nio.file.*;

public class Delete2 {
  public static void main(String[] args) {
     if (args.length != 1) {
       System.err.println("Usage: java Delete2 <fichier ou répertoire>");
       System.exit(1);
     }
     try {
       Files.delete(Paths.get(args[0]));
     } catch (NoSuchFileException e) {
       System.err.println("Delete2: fichier ou répertoire inexistant: " + args[0]);
       System.exit(1);
     } catch (DirectoryNotEmptyException e) {
       System.err.println("Delete2: répertoire non vide : " + args[0]);
       System.exit(2);
     } catch (IOException e) {
       System.err.println("Delete2: impossible de supprimer " + args[0]);
       System.exit(3);
     }
  } // main
} // class Delete2
```

Flux d'entrée-sortie : hiérarchie des flux d'octets



Flux d'entrée-sortie : hiérarchie des flux d'octets

- Toutes les méthodes de lecture s'appellent *read(...)*, toutes les méthodes d'écriture s'appellent *write(...)*. Ces méthodes peuvent lancer une *IOException*.
- `FileInputStream` et `FileOutputStream` permettent de lire et écrire des octets ou des tableaux d'octets dans des fichiers.
- `ByteArrayInputStream` et `ByteArrayOutputStream` permettent de lire et écrire des octets ou des tableaux d'octets en mémoire.
- `PipedInputStream` et `PipedOutputStream` permettent de lire et écrire des octets ou des tableaux d'octets dans un `PipedOutputStream` ou un `PipedInputStream`. Ils implémentent une communication par tubes entre des threads.

Flux d'entrée-sortie : hiérarchie des flux d'octets

- `FilterInputStream` et `FilterOutputStream` permettent de filtrer les octets lus ou écrits. Ils sont créés à partir d'un `InputStream` ou d'un `OutputStream`. La méthode `read(...)` appelle celle de l'`InputStream`, traite les octets lus et renvoie le résultat. La méthode `write(...)` traite les octets puis appelle celle de l'`OutputStream`.
- Ces deux classes ne réalisent aucun filtrage (et on ne peut pas les utiiser directement car leurs constructeurs sont *protected*) : ce sont leurs sous-classes qui s'en occupent (`BufferedInputStream`, `DataInputStream`, `PushbackInputStream` et `BufferedOutputStream`, `DataOutputStream`).
- `BufferedInputStream` et `BufferedOutputStream` effectuent des entrées-sorties *tamponnées*, donc plus efficaces.
- `DataInputStream` et `DataOutputStream` lisent ou écrivent des octets bruts et les interprètent dans différents formats. Ces objets permettent de lire et écrire des types primitifs (numériques) de Java (`readFloat()`, `writeBoolean()`, etc.)

Flux d'entrée-sortie : hiérarchie des flux d'octets

- Il ne faut pas utiliser les flux d'octets pour les fichiers textes (donc notamment pour les E/S standard) car ils ne gèrent pas correctement les caractères Unicode. Pour cela, on utilise `Reader` et `Writer` (ajoutées en Java 1.1).
- Pour des raisons « historiques », les attributs statiques `System.in` et `System.out` sont, respectivement, de type `InputStream` et `PrintStream`. Mais il ne faut plus utiliser la classe `PrintStream` pour écrire des chaînes de caractères.

Flux d'entrée-sortie : Exemple typique d'utilisation

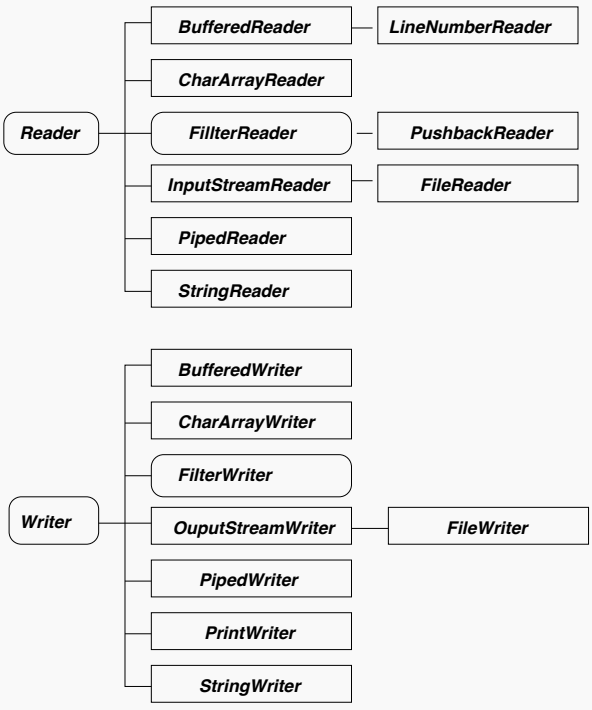
Copie d'un fichier bloc par bloc avant Java 7

```
BufferedInputStream source;
BufferedOutputStream dest;
try {
  source = new BufferedInputStream(new FileInputStream("toto.dat"), 4096);
  dest = new BufferedOutputStream(new FileOutputStream("toto.dat.bak"), 4096);
  int nb_lus;
  byte[] tampon = new byte[4096];

  while ( (nb_lus = source.read(tampon)) != -1) dest.write(tampon, 0, nb_lus);
}
finally {
  if (source != null) try { source.close(); } catch(IOException e) {}
  if (dest   != null) try { dest.close(); }   catch(IOException e) {}
}
```

Copie d'un fichier bloc par bloc à partir de Java 7

```
try (BufferedInputStream src = new BufferedInputStream(new FileInputStream("toto.dat"), 4096);
     BufferedOutputStream dest = new BufferedOutputStream(new FileOutputStream("toto.dat.bak"), 4096)) {
  int nb_lus;
  byte[] tampon = new byte[4096];

  while ( (nb_lus = src.read(tampon)) != -1) dest.write(tampon, 0, nb_lus);
} // fermeture automatique de src et dest
```


Flux d'entrée-sortie : hiérarchie des flux de caractères



Flux d'entrée-sortie : hiérarchie des flux de caractères

- Toutes ces classes sont spécialisées dans le traitement du texte et gèrent correctement les conversions entre les encodages locaux (ISO-8859-1, par exemple) et l'Unicode.
- La classe la plus utilisée pour la lecture est `BufferedReader` qui dispose des méthodes *read()* pour lire un caractère ou un tableau de caractères et *readLine(...)* pour lire un `String`.
- On crée généralement un `BufferedReader` à partir d'un `FileReader` ou d'un `InputStreamReader`. C'est l'équivalent de la classe `BufferedInputStream` des flux d'octets.
- Toutes ces méthodes peuvent lever une `IOException`.

Flux d'entrée-sortie : hiérarchie des flux de caractères

- La classe la plus utilisée pour l'écriture est `PrintWriter`, qui utilise les méthodes *toString()* des objets pour les convertir en `String`, ou effectue cette conversion pour les types primitifs, avant de les écrire avec *print(...)* ou *println(...)*. Elle fournit aussi la méthode *write(...)* surchargée pour écrire des tableaux de caractères ou des `String`.
- Toutes ces opérations sont automatiquement tamponnées (sauf si on a demandé le contraire à la création de l'objet).
- Les méthodes de `PrintWriter` ne lèvent jamais d'exception mais positionnent un flag que l'on peut tester avec `checkError()`.

Exemple sur les fichiers texte

Exemple typique avant Java 7

```
BufferedReader src;
PrintWriter dest;
try {
  src = new BufferedReader(new FileReader("toto.txt"));
  dest = new PrintWriter(new FileWriter("toto.txt.bak"));
  String s;
  while ( (s = src.readline()) != null) {
    System.out.println(s);
    dest.println(s);
 }
} finally {
  if (src != null) try { src.close(); } catch(IOException e) {}
  if (dest != null) try { dest.close(); }   catch(IOException e) {}
}
```

Exemple typique à partir de Java 7

```
try (BufferedReader src = new BufferedReader(new FileReader("toto.txt"));
     PrintWriter dest = new PrintWriter(new FileWriter("toto.txt.bak"))) {
  String s;
  while ( (s = src.readline()) != null) {
    System.out.println(s);
    dest.println(s);
 }
} // Fermeture automatique des fichiers
```


La classe Console

- La classe *Console* a été ajoutée à `java.io` par Java 6 pour faciliter la lecture et l'écriture sur la console (en réalité, elle simplifie les accès à `System.in` et `System.out`).
- La classe `Console` n'a pas de constructeur. Pour y accéder, on utilise l'appel *System.console()*.
- Une console dispose notamment des méthodes *printf(...)*, *readLine(...)*, *readPassword(...)* (voir la documentation de cette classe).
- On rappelle que la classe *Scanner* permet de saisir des valeurs de types primitifs (avec *nextInt()*, *nextFloat()*, etc.)

```
// Exemple d'utilisation de la classe Console
import java.io.Console;

class TestConsole {
  public static void main(String[] args) {
     String nom;
     Console console = System.console();

     if (console == null) return;    // pas de console disponible

     nom = console.readLine("Entrez votre nom : ");
     console.printf("Bonjour %s%n", nom)
  }
}
```

Sérialisation

- La *sérialisation* consiste à écrire l'état d'un objet dans un flux d'octets. La *désérialisation* consiste à restaurer l'état d'un objet à partir d'un flux d'octets produit par une sérialisation précédente.
- Seuls les objets qui implémentent l'interface *java.io.Serializable* peuvent être sérialisés.
- La sérialisation d'un objet entraîne la sérialisation de tous les objets qu'ils contient : ceux-ci doivent donc également être sérialisables.
- L'interface `Serializable` est une *interface marqueur* : elle ne définit aucune méthode.
- Si une classe est sérialisable, toutes ses classes filles le sont aussi.
- Les variables *transient* et *static* ne sont pas sauvegardées au cours d'une sérialisation.

Sérialisation

- Typiquement, l'objet sérialisé est sauvegardé dans un flux *ObjectOutputStream* à l'aide de la méthode *writeObject()*.
- Pour désérialiser un objet, on utilise la méthode *readObject()* d'un flux *ObjectInputStream*.
- Exemple de classe sérialisable :

```
class MaClasse implements java.io.Serializable {

  private static final long serialVersionUID = 2012L;   // recommandé...

  private String s;
  private int i;
  private double d;

  public MaClasse(String s, int i, double d) {
    this.s = s; this.i = i; this.d = d;
  }

  @Override
  public String toString() {
    return "s = " + s + ", i = " + i + ", d = " + d;
  }
}
```

Sérialisation : exemple d'utilisation

```
import java.io.*;

class TestSerialisation {
  public static void main(String[] args) {

    // Sérialisation
    try ( ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("MaClasse.dat")) ) {
      MaClasse obj1 = new MaClasse("Hello", 42, 3.14);
      System.out.println("obj1 = " + obj1);
      os.writeObject(obj1);
    } catch (IOException e) {
      System.err.println("Erreur pendant la sérialisation : " + e);
      System.exit(1);
    }

    // Désérialisation
    try ( ObjectInputStream is = new ObjectInputStream(new FileInputStream("MaClasse.dat")) ) {
      MaClass obj2 = (MaClasse)is.readObject();
      System.out.println("obj2 = " + obj2);
    } catch (IOException e) {
      System.err.println("Erreur pendant la désérialisation : " + e);
      System.exit(2);
    }
  }
}
```


Fichiers de propriétés

- La classe *java.util.Properties* permet de gérer simplement les fichiers de configuration d'un programme ou peut servir à créer une base de données très simple.
- Un fichier de propriétés est un fichier texte formé de lignes de la forme `propriété=valeur`.
- En réalité, `Properties` hérite de `Hashtable` : c'est donc une table de hachage (les clés sont les propriétés et les valeurs sont les valeurs associées à ces propriétés).
- La méthode *System.getProperties()* renvoie un objet `Properties` décrivant les variables d'environnement du programme :

```
import java.util.Properties;

class TestProperties {
    public static void main(String[] args){
      Properties sysProps = System.getProperties();
      // sysProps.list(System.out);
      System.out.println("Exécuté sur " + sysProps.get("os.name") +
                         " par " + sysProps.get("user.name"));
    }
}
```

Fichiers de propriétés

- Exemple de fichier de configuration : `TestConfig.conf`

```
nom=jaco
repertoire=/home/jaco/Travail
taille=42
```

- Exemple d'utilisation :

```
import java.util.Properties;
import java.io.*;

class TestConfig {
  public static void main(String[] args) {
    Properties config = new Properties();
    String nom = "", rep = "";
    int valeur = 0;

    try (FileInputStream in = new FileInputStream("TestConfig.conf")) {
      config.load(in);
      nom = config.getProperty("nom");
      rep = config.getProperty("repertoire", (String)System.getProperties().get("user.home"));
      valeur = Integer.parseInt(config.getProperty("taille"));
    } catch (IOException e) { // il n'y a pas de fichier de config... tant pis
    } catch (NumberFormatException e) {
      System.err.println("format de configuration incorrect : " + e.getMessage());
      System.exit(1);
    }

    if (nom != "") {
      System.out.printf("nom = %s, rep = %s, val = %d%n", nom, rep, valeur);
    }
  }
}
```