Le langage Java

Apprentissage en lien avec le langage UML

Le langage Java

Apprentissage en lien avec le langage UML

Utilisation des Flots (streams)

- En Java, la communication entre le programme et les échanges de données entre un programme et l'extérieur (autre programme, fichier ou application réseau, ...) sont réalisées à travers un " flot " de données
- Un flot permet de transporter **séquentiellement** des données. Les données sont transportées une par une (ou bloc) et dans l'ordre
- Le cycle d'utilisation d'un flot de données est le suivant :
  1. Ouverture du flot
  2. Tant qu'il y a des données à lire (ou à écrire), on lit (ou on écrit) la donnée suivante dans le flot
  3. Fermeture du flot

Sources ou destinations de flots

- Fichier
- Socket pour échanger des données sur un réseau
- Données de grandes tailles dans une base de données (blob) (images, par exemple)
- Pipe entre 2 processus
- Tableau d'octets (en mémoire)
- Chaîne de caractères
- URL (adresse Web)
- ...

Le paquetage `java.io`

- La plupart des classes liées au E/S sont définies dans le paquetage `java.io`
- Il prend en compte un grand nombre de flots :
  - On distingue deux grands types de flots :
    - Les octets
    - Les caractères
  - différentes sources et destinations
  - Il est possible de *décorer* les flots :
    - le modifier au fur et à mesure
- Le nombre de classes de ce paquetage est important, il faut regarder la javadoc.

Les grands types de flots

- Les *flots d'octets* servent à lire ou écrire des octets " bruts ", qui représentent des données codées, manipulées par un programme
- Les *flots de caractères* servent à lire ou écrire des données qui représentent des caractères lisibles par un homme (codés en Unicode)

Les types de classes

- Pour les deux grands type de flots (octets et caractères), on distingue :
  - Les classes associées à une source ou une destination “ concrète ”
    - `FileReader` pour lire un fichier
  - Les classes qui permettent de “ décorer ” un autre flot
    - `BufferedReader` qui ajoute un tampon (buffer) pour lire un flot de caractères

La décoration des flots

- Les fonctionnalités de base d'un flot sont la lecture ou l'écriture (méthodes `read` ou `write`)
- Selon les besoins, on peut lui ajouter d'autres fonctionnalités (appelées décorations) :
  - utilisation d'un buffer pour réduire les lectures ou écritures réelles
  - Codage ou décodage des données manipulées
  - Compression ou décompression de ces données
  - etc...

Les classes de base du paquetage `java.io`

- `InputStream` (Lecture d'octets)
- `OutputStream` (Ecriture d'octets)
- `Reader` (Lecture de caractères Unicode)
- `Writer` (Ecriture de caractères Unicode)
- `File` (Manipulation de noms de fichiers et de répertoires)
- `StreamTokenizer` (Segmentation lexicale d'un flot d'entrée)

Les sources et destinations concrètes

- Fichiers :
  - `File{In|Out}putStream`
  - `File{Reader|Writer}`
- Tableaux
  - `ByteArray{In|Out}putStream`
  - `CharArray{Reader|Writer}`
- Chaînes de caractères
  - `String{Reader|Writer}`

Les décorateurs

- Pour ajouter un tampon sur une entrée/sortie :
  - `Buffered{In|Out}putStream`
  - Buffered{Reader—Writer}
- Pour lire et écrire des types primitifs sous une forme binaire :
  - `Data{In|Out}putStream`
- Pour compter les lignes lues :
  - `LineNumberReader`
- Pour écrire sous forme de chaînes de caractères :
  - `PrintStream`
  - `PrintWriter`
- Pour permettre de remettre un caractère lu dans le flot :
  - `PushbackInputStream`
  - `PushbackReader`

La lecture et l'écriture de flots d'octets

Classe d'entrée	**Classe de sortie**	**Utilisation**
`InputStream`	`OutputStream`	Classes abstraites de base pour les lecture et écriture d'un flot de données
`FilterInputStream`	`FilterOutputStream`	Classe mère des classes qui ajoutent des fonctionnalités à Input/OutputStream
`BufferedInputStream`	`BufferedOutputStream`	Lecture et écriture avec buffer
`DataInputStream`	`DataOutputStream`	Lecture et écriture des types primitifs
`FileInputStream`	`FileOutputStream`	Lecture et écriture d'un fichier
	`PrintStream`	Possède les méthodes `print()`, `println()` utilisées par `System.out`

La classe `InputStream`

- Classe abstraite
- C'est la racine des classes qui concernent la lecture d'octets depuis un flot de données
- Elle fixe les méthodes de base
- Elle possède un constructeur sans paramètre
- La lecture d'un flot d'octets avec `InputStream` :
  - `FilterInputStream` (Décorateur - Doit être sousclassée)
    - `BufferedInputStream` (Entrées bufférisées)
    - `DataInputStream` (Lecture des types primitifs)
  - `FileInputStream` (Lecture des octets d'un fichier)
  - `ObjectInputStream` (Lecture d'un objet sérialisé)

Méthodes de la classe InputStream

- Interface publique de cette classe :
  - `abstract int read() throws IOException`
  - `int read(byte[] b) throws IOException`
  - `int read(byte[] b, int début, int nb) throws IOException`
  - `long skip(long n) throws IOException`
  - `int available() throws IOException`
  - `void close() throws IOException`
  - `synchronized void mark(int nbOctetsLimite)`
  - `synchronized void reset() throws IOException`
  - `public boolean markSupported()`
  - La lecture est bloquante

Les sous-classes de InputStream

- InputStream
  - FileInputStream
  - PipedInputStream
  - FilterInputStream
    - LineNumberInputStream
    - DataInputStream
    - BufferedInputStream
    - PushbackInputStream
  - ByteArrayInputStream
  - SequenceInputStream
  - StringBufferInputStream
  - ObjectInputStream

Un exemple : lire des octets dans un fichier

- Cet exemple utilise
  - `FileInputStream` : fichier source
  - `BufferedInputStream` : décorateur d'ajout d'un buffer
  - `DataInputStream` : décorateur de lecture des primitifs
- En général, on n'utilise que le flot décoré (pas besoin des variables intermédiaires)

```
package coursSSI3.exemples.es;
import java.io.*;
public class Decorateur {
public static void main(String[] args) throws IOException {
  FileInputStream fis = new FileInputStream("fichier");
  BufferedInputStream bis = new BufferedInputStream(fis);
  DataInputStream dis = new DataInputStream(bis);
  double d = dis.readDouble(); String s = dis.readUTF(); //UTF
  int i = dis.readInt();
  dis.close();
  DataInputStream disBis = new DataInputStream(
    new BufferedInputStream(new FileInputStream("fichier2")));
  i = dis.readInt();
  dis.close();  } }
```

Listing 1 – coursSSI3/exemples/es/Decorateur.java

La lecture des octets d'un fichier

- Pour lire un fichier qui contient des octets qu'on ne peut lire sous forme de types Java particuliers (images, vidéo, etc, ...)
- La chaîne de caractères passé au constructeur de File peut être un chemin relatif ou absolu
- Les chemins relatifs sont relatifs au répertoire dans lequel on lance la commande java, et pas par rapport au répertoire qui contient la classe

```
package coursSSI3.exemples.es;
import java.io.*;
public class LectureOctets {
        public static void main(String[] args)
          throws IOException {
                File f = new File("fichier");
                int tailleFichier = (int)f.length();
                byte[] donnees = new byte[tailleFichier];
                DataInputStream dis =
                  new DataInputStream(
                   new FileInputStream(f));
                dis.readFully(donnees);
                dis.close();
        }
}
```

Listing 2 – coursSSI3/exemples/es/LectureOctets.java

L'écriture d'un flot d'octets

- `OutputStream` (Classe abstraite de base)
  - `FilterOutputStream` (Décorateur)
    - `BufferedOutputStream` (Sorties bufférisées)
    - `DataOutputStream` (Ecriture de types primitifs)
    - `PrintStream` (Utilisé par System.out - Ne pas utiliser autrement)
  - `FileOutputStream` (Ecriture des octets d'un fichier)
  - `ObjectOutputStream` (Ecriture d'un objet sérialisé)

La classe `OutputStream`

- Interface publique de cette classe (ajouter `throws IOException` à toutes les méthodes) :
  abstract void write(int b)
  void write(byte[] b)
  void write(byte[] b,
  int debut, int nb)
  void flush()
  void close()
- Remarque : avec la méthode `write(int b)`, seul l'octet de poids faible de b est écrit dans le flot

Les particularités de `PrintStream`

- Cette classe possède les deux méthodes `print()` et `println()` qui écrivent tous les types de données sous forme de chaînes de caractères
- Aucune des méthodes de `PrintStream` ne lève d'exception ; on peut savoir s il y a eu une erreur en appelant la méthode `checkError()`
- Attention, `println()` n'effectue un `flush()` (vidage des tampons) que si le `PrintStream` a été créé avec le paramètre " autoflush "

Un exemple de ByteArrayOutputStream

- Ecrire un table d'octet en mémoire sans connaire la taille à l'avance

```
package coursSSI3.exemples.es;
import java.io.ByteArrayOutputStream;
public class EcritureOctet {
        private static int p=4;
        private static int[] T={4,5,-2,5};

        public static int getValeur() {return T[p++];}

        public static void main(String[] args) {
                ByteArrayOutputStream out =
                 new ByteArrayOutputStream();
                int b;
                while ((b = getValeur()) > 0) {
                 out.write(b); }
                byte[] octets = out.toByteArray();
        }
}
```

Listing 3 – coursSSI3/exemples/es/EcritureOctet.java

L'écriture de types primitifs dans un fichier

```
package coursSSI3.exemples.es;

import java.io.*;

public class EcriturePrimitifs {
        public static void main(String[] args)
          throws IOException {
                DataOutputStream dos =
                        new DataOutputStream(
                          new BufferedOutputStream(
                          new FileOutputStream("fichier")));
                dos.writeDouble(27.7);
                dos.writeUTF("Pierre");
                dos.writeInt(56);
                dos.close();
        }

}
```

Listing 4 – coursSSI3/exemples/es/EcriturePrimitifs.java

- Le constructeur `FileOutputStream(String nom, boolean append)` permet d'ajouter à la fin du fichier ; sinon, le contenu du fichier est effacé à la création

Les exceptions liées aux entrées-sorties

- `Exceptions`
  - `IOException` (Exception durant une entrée-sortie)
    - `EOFException` (Lecture d'une fin de fichier)
    - `FileNotFoundException` (Fichier n'existe pas)
    - `ObjectStreamException` (Problème lié à la sérialisation)

Le traitement des exceptions

```
package coursSSI3.exemples.es;

import java.io.*;

public class ErreurES {
   public static void main(String[] args) {
                try {
                        DataInputStream dis =
                         new DataInputStream(
                           new FileInputStream("fichier"));
                        try { while (true) {
                          double d= dis.readDouble();
                          }
                        }catch(EOFException e) { }
                         catch(IOException e) { }
                         finally {
                           try {dis.close();}
                           catch (IOException e) {}
                           }
                } catch(FileNotFoundException e) { }
        }
}
```

Listing 5 – coursSSI3/exemples/es/ErreurES.java

Classes de lecture d'un flot de caractères

- `Reader` (Classe abstraite de base)
  - `FilterReader` (Décorateur)
  - `InputStreamReader` (Lecture de caractères Unicode à partir d'un flot d'octets)
    - `FileReader` (Lecture de caractères Unicode à partir des octets d'un fichier)
  - `BufferedReader` (Entrées bufférisées)

Classes d'écriture d'un flot de caractères

- `Writer` (Classe abstraite de base)
  - `FilterWriter` (Décorateur)
  - `OutputStreamWriter` (Ecriture de caractères Unicode sous forme d'octets)
    - `FileWriter` (Ecriture de caractères Unicode dans un fichier, sous forme d'octets)
  - `BufferedWriter` (Ecriture bufférisée)
  - `PrintWriter` (Fournit des méthodes print() et println())

Lecture et écriture de flots de caractères

Classe pour entrée	**Classe pour sortie**	**Fonctions fournies**
`Reader`	`Writer`	Classes abstraites de base
`InputStreamReader`	`OutputStreamReader`	Ponts entre les flots d'octets et les flots de caractères
`FileReader`	`FileWriter`	Lecture et écriture de caractères à partir des octets d'un fichier (codage par défaut)
`BufferedReader`	`BufferedWriter`	Lecture et écriture avec buffer
	`PrintWriter`	Possède les méthodes `print()` et `println()`

print et println de PrintWriter

- Ces méthodes sont surchargées pour tous les types primitifs, les tableaux de caractères et les classes `String` et `Object`
- Elles ne lancent jamais d'exceptions (comme toutes les autres méthodes de `PrintWriter`) ; `boolean checkError()` permet de savoir s'il y a eu une erreur avec le flot sous-jacent
- Attention, `println()` n'effectue un `flush()` (vidage des buffers) que si le `PrintWriter` a été créé avec le paramètre `autoflush`

Séparateur de lignes

- La façon de séparer les lignes dépend du système d'exploitation
- Pour être portable utiliser
  - `println` de `PrintWriter` (le plus simple)
  - `writeLine` ou `newLine` de `BufferedWriter`
  - ou la propriété système `line.separator` (`System.getProperty("line.separator")`)
- Ne pas utiliser le caractère \n qui ne convient pas, par exemple, pour Windows

Codage

- En Java les caractères sont codés en Unicode
- Ce n'est souvent pas le cas sur les périphériques source ou destination des flots (le plus souvent ASCII étendu ISO 8859-1 pour les français)
- Des classes spéciales permettent de faire les traductions entre le codage Unicode et un autre codage
- Un codage par défaut est automatiquement installé par le JDK, conformément à la locale

Flots de caractères et les flots d'octets

- `InputStreamReader` et `OutputStreamWriter` sont des classes filles de Reader et Writer
- Leur constructeur prend en paramètre un flot d'octets ; par exemple, `public OutputStreamWriter(OutputStream out)`
- On peut préciser un codage particulier en paramètre de leur constructeur si on ne veut pas le codage par défaut
- Ecriture-lecture dans un fichier de texte
  - `File{Reader|Writer}` sont des classes filles de `InputStreamReader` et `OutputStreamWriter`
  - Elles permettent de lire et d'écrire des caractères Unicode dans un fichier, suivant le codage par défaut (utiliser leur classe mère si on veut un autre codage)

Fichier composé de lignes de texte

- En lecture, on utilise la classe `BufferedReader` qui comprend la méthode `readLine()`
- En écriture, on utilise la classe `PrintWriter` qui comprend les méthodes `print()` et `println()`

Lire les lignes de texte d'un fichier

```
package coursSSI3.exemples.es;

import java.io.*;

public class LireLigne {
  public static void main(String[] args) {
    try {
       String ligne;
       FileReader fr = new FileReader("fichier");
       BufferedReader br = new BufferedReader(fr);
       try {
        while ((ligne = br.readLine()) != null) {
            }
        } catch (IOException e) {  }
        finally {
          try {br.close();}
          catch(IOException e) { }
          }
       } catch (FileNotFoundException e) { }
  }
}
```

Listing 6 – coursSSI3/exemples/es/LireLigne.java

Écrire des lignes de texte dans un fichier

```
package coursSSI3.exemples.es;

import java.io.*;

public class EcrireLigne {
  public static void main(String[] args) {
    try {
       PrintWriter pw = new PrintWriter(
         new BufferedWriter(new FileWriter("fichier")),
          true);
       String ligne="le texte";
       pw.println(ligne);
       pw.close(); // vide les buffers
      } catch(IOException e) {}
  }

}
```

Listing 7 – coursSSI3/exemples/es/EcrireLigne.java

Les séparateurs des données

- Pour les flots d'octets, il suffit de relire les données dans l'ordre dans lequel elles ont été écrites, avec le décodage approprié
- Pour les flots de caractères, on doit explicitement mettre des séparateurs entre les données ; par exemple, pour distinguer un nom d'un prénom
- `StringTokenizer` et `StreamTokenizer` facilitent la relecture de données écrites avec des séparateurs

Entrées-sorties sur clavier-écran

- Lecture de caractères tapés au clavier :

```
package coursSSI3.exemples.es;

import java.io.IOException;

public class LireClavier {
        public static void main(String[] args) {
                int n;char car;
                String s = "";
                try {
                 while (true) {
                  n = System.in.read();
                  if (n == -1) break;
                  car = (char)n;  s += car;
                  }
                 }
                 catch(IOException e) {
                   System.err.println("Erreur I/O" + e);
                 }
        }
}
```

Listing 8 – coursSSI3/exemples/es/LireClavier.java

Nouveau paquetage java.nio

- Reprend toute l'architecture des classes pour les entrées/sorties
- notion de canal (channel) et de buffer
- Utilise les possibilités avancées du système d'exploitation hôte pour optimiser les entrées-sorties et offrir plus de fonctionnalités

La Classe `File`

- Cette classe représente la notion de fichier, indépendamment du système d'exploitation
- Un fichier est repéré par un chemin abstrait composé d'un préfixe optionnel (nom d'un disque par exemple) et de noms (noms des répertoires parents et du fichier lui-même)
- Attention, `File fichier = new File("/bidule/truc");` ne lève aucune exception si /bidule/truc n'existe pas dans le système de fichier
- Constructeurs
  - Les chemins passés en premier paramètre peuvent être des noms relatifs ou absolus
  - File (String chemin)
  - File (String cheminParent, String chemin)
  - File (File parent, String chemin)
- La classe File offre des facilités pour la portabilité des noms de fichiers

Les fonctionnalités de la classe `File`

- Elle permet d'effectuer des manipulations sur les fichiers et répertoires considérés comme un tout (mais pas de lire ou d'écrire le contenu) :
  - lister un répertoire,
  - supprimer, renommer un fichier
  - créer un répertoire
  - créer un fichier temporaire
  - connaître les droits que l'on a sur un fichier (lecture, écriture)
  - etc.

Méthodes de `File`

- informatives
  - `boolean exists()`
  - `boolean isDirectory()`
  - `boolean isFile()`
  - `boolean canRead()`
  - `boolean canWrite()`
  - `long lastModified()`
  - `long length()`
- actions
  - `boolean delete()` (true si suppression réussie)
  - `boolean mkdir()`
  - `boolean mkdirs()` (peut créer des répertoires intermédiaires)
  - `boolean renameTo(File nouveau)`
  - `boolean setLastModified(long temps)`
  - `boolean setReadOnly()`

Méthodes pour les noms

- `String getName()` (nom terminal)
- `String getPath()` (chemin absolu ou relatif)
- `File getAbsoluteFile()`
- `String getAbsolutePath()`
- `String getParent()`
- `File getParentFile()`
- `URL toURL()` (de la forme file:url)

Classe URL

- `protocole://machine[:port]/cheminPage`
- Cette classe du paquetage java.net fournit de nombreux constructeurs
- Elle permet d'extraire des éléments à partir d'un URL (`getPort`, `getHost`, ...)
- Les données associées à l'URL peuvent être obtenues par les méthodes `openConnection` et `openStream` ou par la méthode `getContent`

Lire le code HTML d'une page Web

- La classe `URL` fournit la méthode `InputStream openStream() throws IOException`

```
package coursSSI3.exemples.es;
import java.io.*;
import java.net.*;

public class LireURL {
  public static void main(String[] args) throws IOException {
    String ligne;
    URL url = new URL("http://www.univ-tln.fr/index.html");
    InputStream is = url.openStream();
    BufferedReader br =
      new BufferedReader( new InputStreamReader(is));
    while ((ligne = br.readLine()) != null) {
       System.out.println(ligne); }
 }
}
```

Listing 9 – coursSSI3/exemples/es/LireURL.java

Définition de la Sérialisation

- Sérialiser un objet c'est transformer l'état (les valeurs des variables d'instances) d'un objet en une suite d'octets
- On peut ainsi conserver l'état d'un objet pour le retrouver ensuite et reconstruire un autre objet avec le même état
- On utilise ObjectOutputStream et ObjectInputStream

Qu'est-ce qui est sérialisé ?

- Les valeurs des variables d'instance (pas des variables de classe) des instances sérialisées
- Des informations sur les classes des objets sérialisés ; en particulier :
  - nom de la classe
  - noms, types, modificateurs des variables à sauvegarder
  - des informations qui permettent de savoir si une classe a été modifiée entre la sérialisation et la désérialisation

Utilisation de la sérialisation

- Conserver un objet dans un fichier ou une base de données pour le récupérer plus tard
- Conserver la configuration d'un composant, pour pouvoir le réutiliser plus tard dans une application (JavaBean)
- Transmettre les paramètres (de types non primitifs) d'une méthode appelée sur un objet distant (RMI) : le paramètre est sérialisé, les octets sont transmis sur le réseau et le paramètre est reconstruit sur la machine distante

Interface Serializable

- Pour pouvoir être sérialisé, un objet doit être une instance d'une classe qui implémente l'interface Serializable
- Cette interface ne comporte aucune méthode ; elle sert seulement à marquer les classes sérialisables
- La plupart des classes du JDK sont sérialisables
- Certaines classes ne peuvent pas être sérialisées (InputStream par exemple) ; d'autres ne doivent pas l'être (par sécurité)
- Si on ne veut pas qu'une variable d'instance soit sérialisée, on la déclare transient : `private transient int val;`
  - Quand l'objet sera désérialisé, la valeur de cette variable devra être recalculée s'il en est besoin sinon elle recevra la valeur par défaut de son type

Compression/décompression

- Le paquetage `java.util.zip` fournit des classes filtres pour compresser des flots :
  - `GZIPInputStream` (et `GZIPOutputStream`) pour travailler avec des donné au format GZIP
  - `ZipInputStream` (et `ZipOutputStream`) pour lire ou écrire des entrées (le plus souvent des fichiers) compressées au format ZIP
  - Pour lire les fichiers zip, la classe `java.util.zip.ZipFile` permet des traitements plus performants

Exemple de compression

- compresser un objet sérialisé

```
package coursSSI3.exemples.es;

import java.io.*;
import java.util.zip.GZIPOutputStream;

import coursSSI3.exemples.animaux.Chien;
public class Compression {
        public static void main(String[] args)
          throws IOException {
                FileOutputStream fos =
                   new FileOutputStream("fichier");
                GZIPOutputStream gz =
                   new GZIPOutputStream(fos);
                ObjectOutputStream oos =
                   new ObjectOutputStream(gz);
                oos.writeObject(new Chien("Medor"));
        }
}
```

Listing 10 – coursSSI3/exemples/es/Compression.java

Exemple de compression

- récupérer l'objet sérialisé

```
package coursSSI3.exemples.es;

import java.io.*;
import java.util.zip.GZIPInputStream;

import coursSSI3.exemples.animaux.Chien;

public class Decompresser {
        public static void main(String[] args)
          throws IOException, ClassNotFoundException {
                Chien c;
                FileInputStream fis =
                  new FileInputStream("fichier");
                GZIPInputStream gz =
                  new GZIPInputStream(fis);
                ObjectInputStream ois =
                  new ObjectInputStream(gz);
                c= (Chien)ois.readObject();
        }

}
```

Listing 11 – coursSSI3/exemples/es/Decompresser.java