Chapitre 8

La gestion des entrées/sorties

1. Définition du concept d’entrée/sortie
2. Lecture/Ecriture des données sur clavier/console
3. Lecture/Ecriture des données sur fichier

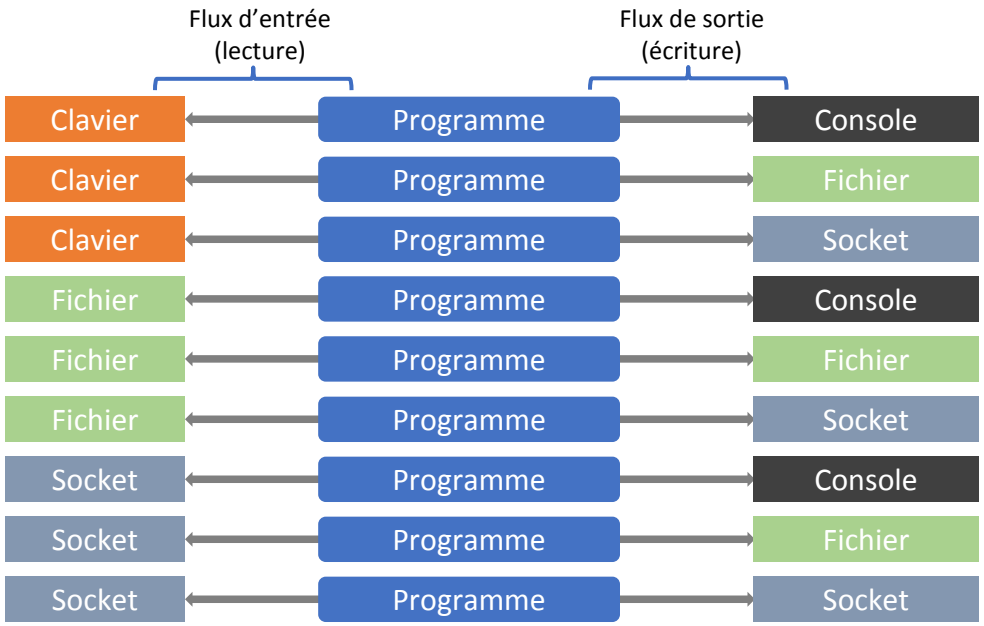
1. Définition du concept d’entrée/sortie
2. Lecture/Ecriture des données sur clavier/console
3. Lecture/Ecriture des données sur fichier

Définition du concept d'entrée/sortie

- Beaucoup de programmes sont sensés interagir avec le monde extérieur. Un programme :
  - réagit suite à des paramètres, données, ou évènements qu'il reçoit en entrée
  - fait des traitements et produit en sortie un résultat en direction d'un utilisateur, un système ou déclenche un autre événement, etc
- Pour ce faire, il est nécessaire de définir des **flux d'entrée/sortie** pour permettre cette interaction
- Un **flux** est une série de données qui circulent de façon unidirectionnelle sur un canal de communication (ex : câble, bus, fibre, tube, etc) entre deux entités et sur lequel écoute un programme.
  - Une entité pouvant être : le **clavier**, la **console** (terminal ou invite de commande), un **fichier** ou un **socket** réseau
  - Java encapsule les flux par des classes situées dans le package ***java.io*** et utilisables dans tout programme
    > Ce qui permet de s'abstraire du type de canal utilisé et de s'atteler à l'utilisation des fonctionnalités fournies
- Pour un programme, les types de flux peuvent donc être :
  - la lecture des données sur un clavier
  - l'affichage des données sur une console
  - la lecture/écriture des données dans un fichier
  - la lecture/écriture des données sur un socket réseau

Définition du concept de flux d'entrée/sortie

Combinatoire des types de flux



Scénarii possibles pour un programme :

Dans ce cours, nous n'aborderons pas les flux d'entrée/sortie impliquant les sockets

1 Définition du concept d'entrée/sortie

2 Lecture/Ecriture des données sur clavier/console

3 Lecture/Ecriture des données sur fichier

Lecture/Ecriture des données sur clavier/console

- La bibliothèque *java.io* propose une classe appelée **System** (*java.io.System*) pour gérer les flux de données entre :
  - un programme et le clavier
  - un programme et la console
- Cette classe **System** possède trois attributs static qui permettent d'atteindre cet objectif :
  - **in** (**System.in**) pour la lecture des données sur le clavier : public static final InputStream in;
  - **out** (**System.out**) pour l'écriture des données sur la console : public static final PrintStream out;
  - **err** (**System.err**) pour l'écriture des erreurs sur la console : public static final PrintStream err;
- Un processus de gestion de flux obéit aux trois règles suivantes :
  1. l'ouverture du flux
  2. la lecture ou l'écriture effective des données
  3. la fermeture du flux

Lecture/Ecriture des données sur clavier/console

Lecture des données sur un clavier / le flux standard *System.in* (1/3)

- La classe ***System.in*** propose, entre autres, les méthodes suivantes pour la gestion des flux d'entrée sur le clavier :

  int read() : lit le prochain octet. La valeur de l'octet est retournée comme un entier entre 0 et 255

  int read(byte[] b) : lit un certain nombre d'octets et les copie dans le tableau de bytes b. Le nombre d'octets lu est retourné

  void close() : ferme proprement le flux d'entrée

  ➔ un ***octet*** lu correspond à un ***caractère***, encodé en un entier compris entre 0 et 255, saisi sur le clavier

- Exemple : le programme suivant lit un caractère sur le clavier, le traite et l'affecte à la variable *varChar*

```
import java.io.IOException;

public class ExempleSystemIn {

   public static void main(String[] args) throws IOException {
     int inChar = System.in.read();
     char varChar = (char) inChar;
     System.in.close();
   }
}
```

Remarques :

La lecture de flux avec la méthode System.in.read() peut générer des erreurs. Java permet de gérer cela en utilisant les **Exceptions** que nous aborderons au chapitre 11. Pour le moment si vous utilisez cette méthode dans vos programmes, ajoutez *throws IOException* sur toutes les méthodes concernées et importez la bibliothèque *java.io.IOException* dans votre classe

- L'inconvénient des méthodes read() réside dans le fait qu'un caractère lu est traité comme un entier. Après la lecture donc, il faut encore travailler au « casting » de cet entier pour retrouver le vrai caractère initialement saisi sur le clavier

Lecture/Ecriture des données sur clavier/console

Lecture des données sur un clavier / le flux standard *System.in* (2/3)

- Pour répondre à l'inconvénient des méthodes read() de System.in, Java propose dans la bibliothèque *java.util*, la classe **Scanner** qui offre des fonctionnalités de haut niveau sur les octets lus sur le clavier
- Quelques fonctionnalités non exhaustives de la classe Scanner :

Méthode	Description
`short nextShort()`	Retourne l'entier de type *short* lu sur le clavier. Génère une erreur si la donnée lue sur le clavier n'est pas un *short*
`int nextInt()`	Retourne l'entier de type *int* lu sur le clavier. Génère une erreur si la donnée lue sur le clavier n'est pas un *int*
`double nextDouble()`	Retourne le réel de type *double* lu sur le clavier. Génère une erreur si la donnée lue sur le clavier n'est pas un *double*
`boolean nextBoolean()`	Retourne le booléen lu sur le clavier. Génère une erreur si la donnée lue sur le clavier est différent des mots *true* ou *false*
`String nextLine()`	Retourne un caractère ou une chaine de caractères lu(e) sur le clavier.
`boolean hasNext()`	Retourne *true* ou *false* si le scanner a encore ou non des données à lire sur le clavier
`boolean hasNextInt()`	Retourne *true* ou *false* si le scanner a encore ou non des données de type int à lire avec nextInt() sur le clavier

Sa documentation officielle et exhaustive est consultable ici

Lecture/Ecriture des données sur clavier/console

Lecture des données sur un clavier / le flux standard *System.in* (3/3)

- Exemples d'utilisation de la classe Scanner pour la lecture sur le clavier :

```
import java.util.Scanner;

public class ExempleScanner1 {

   public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        String str = scanner.nextLine();

        scanner.close();
   }

}
```

Ce programme lit un caractère ou une chaine de caractères saisi sur le clavier et l'affecte à la variable *str* de type String. La fin de la saisie est marquée par le clic sur la touche « Entrée » du clavier

```
import java.util.Scanner;

public class ExempleScanner2 {

   public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        int var = scanner.nextInt();

        scanner.close();
   }

}
```

Ce programme lit une donnée sur la clavier attendue comme un entier de type int et l'affecte à la variable var. La fin de la saisie est marquée par le clic sur la touche « Entrée » du clavier. Si l'on saisi un caractère, une chaine de caractères, un nombre réel, une erreur va se produire et le programme s'arrête

Lecture/Ecriture des données sur clavier/console

Ecriture des données sur la console / le flux standard *System.out* (1/2)

- La classe **System.out** (PrintStream) fournie des fonctionnalités pour l'affichage des données sur la console
- Quelques fonctionnalités non exhaustives :

Méthode		Description
`void`	`print(…)`	Affiche sur la console et sur une seule ligne tout type de données (primitifs, objets, chaîne de caractères)
`void`	`println(…)`	Affiche sur la console tout type de données (primitifs, objets, chaîne de caractères), puis retourne à la ligne
`PrintStream`	`append(CharSequence csq)`	Concatène la chaine csq au texte déjà affiché sur la console

Sa documentation officielle et exhaustive est consultable ici

- L'utilisation est simple (et vous avez d'ailleurs déjà vu certaines dans les exemples des chapitres précédents) :
  - > System.out.println()
  - > System.out.print()
  - > System.out.append()

Sa documentation officielle et exhaustive est consultable ici

Lecture/Ecriture des données sur clavier/console

Ecriture des données sur la console / le flux standard *System.out* (2/2)

- L'exemple ci-dessous :
  > Affiche sur la console un texte invitant l'utilisateur à saisir un nombre entier quelconque
  > lit effectivement le nombre saisi sur le clavier à l'aide des classes Scanner/System.in
  > et réaffiche la donnée lue sur la console

```
import java.util.Scanner;

public class ExempleLectureEcriture {

   public static void main(String[] args) {

        //Affiche sur la console le texte "Ecrire un nombre : "
        System.out.println("Ecrire un nombre : ");

        Scanner scanner = new Scanner(System.in);

        int var = scanner.nextInt();

        //Affiche sur la console le nombre lu
        System.out.println("Vous avez saisi : " + var);
        scanner.close();
   }

}
```

1 Définition du concept d'entrée/sortie

2 Lecture/Ecriture des données sur clavier/console

3 Lecture/Ecriture des données sur fichier

Lecture/Ecriture des données sur fichier

- La bibliothèque *java.io* propose une panoplie de classes pour la gestion des flux de données liée à :
  - la lecture des données dans un fichier
  - l'écriture des données dans un fichier

- Dans ce cours nous étudions uniquement les classes suivantes :
  - **File :** classe représentant une abstraction d'un *fichier* ou d'un *répertoire* physique sur un ordinateur identifié par son chemin.
    C'est la classe de base sur laquelle s'appuie toutes les autres classes suivantes qui fonctionnent de paire :
    - **FileInputStream**
    - **FileOutputStream**

    Effectue la lecture (FileInputStream) (resp. l'écriture (FileOutputStream)) octet par octet dans tout type de fichier binaire ou texte
    - **FileReader**
    - **FileWriter**

    Effectue la lecture (FileReader) (resp. l'écriture (FileWriter)) caractère par caractère uniquement sur des fichiers texte
    - **BufferedReader**
    - **BufferedWriter**

    Effectue la lecture (BufferedReader) (resp. l'écriture (BuffuredWriter)) ligne par ligne uniquement sur des fichiers texte

Lecture/Ecriture des données sur fichier

La classe File

- Toutes les classes de gestion des flux d'entrée/sortie dans les fichiers s'appuient sur la classe **File** de *java.io*
- Pour créer un objet de type File, il suffit de connaitre le chemin absolu vers le fichier physique dont il représente

  Exemple : File monFichier = new File(/path/to/file.txt);

  ➢ l'objet *monFichier* représente ainsi, dans un programme, le fichier file.txt localisé à /path/to/ sur l'ordinateur

- Quelques fonctionnalités non exhaustives sur les objets de type File :

Méthode	Description
`boolean exists()`	Retourne si oui ou non le présent objet représente effectivement un fichier ou un répertoire existant
`boolean delete()`	Supprime physiquement le fichier ou le répertoire représenté par le présent objet et retourne si l'opération s'est bien déroulée ou non
`String getAbsolutePath()`	Retourne le chemin absolu vers le fichier ou répertoire représenté par le présent objet
`boolean isFile()`	Retourne si oui ou non le présent objet est un fichier
`boolean isDirectory()`	Retourne si oui ou non le présent objet est un répertoire
`File[] listFiles()`	Retourne un tableau de File représentant les fichiers et sous-répertoires contenus dans le répertoire représenté par l'objet courant

Sa documentation officielle et exhaustive est consultable ici

Lecture/Ecriture des données sur fichier

Les classes FileInputStream et FileOutputStream (1/2)

- Les classes FileInputStream et FileOutputStream disposent chacune d'un constructeur qui prend en paramètre un **File** ou une **String** représentant le chemin absolu vers le fichier sur lequel elles vont chacune travailler

- **FileInputStream** :
  - Permet de lire les données octet par octet dans un fichier, que ce dernier soit un fichier texte ou un fichier binaire
  - Les méthodes principales de la classe :
    - `int read()` : lit le prochain octet. La valeur de l'octet est retournée comme un entier entre 0 et 255. Retourne -1 si la fin du fichier est atteint
    - `int read(byte[] b)` : lit un certain nombre d'octets et les copie dans le tableau de bytes b. Le nombre d'octets lu est retourné. Retourne -1 si fin de fichier
    - `void close()` : ferme proprement le flux d'entrée vers le fichier

- **FileOutputStream** :
  - Permet d'écrire les données octet par octet dans un fichier, que ce dernier soit un fichier texte ou un fichier binaire
  - Les méthodes principales de la classe :
    - `void write(int o)` : écrit l'octet o dans le fichier
    - `void write(byte[] b)` : écrit le tableau d'octets b dans le fichier
    - `void close()` : ferme proprement le flux de sortie vers le fichier

Lecture/Ecriture des données sur fichier

Les classes FileInputStream et FileOutputStream (2/2)

- Exemple :

Le programme suivant lit les données du fichier file1 et les recopie dans le fichier file2

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Exemple1LectureEcritureFichier {

   public static void main(String[] args) throws IOException {

      File f1 = new File("C:/tmp/bidule/file1");
      File f2 = new File("C:/tmp/bidule/file2");

      FileInputStream fin = new FileInputStream(f1);
      FileOutputStream fout = new FileOutputStream(f2);

      //lecture en boucle de tout le contenu du fichier file1
      //et écriture octet par octet dans le fichier file2
      int var = fin.read();
      while(var != -1) {
         fout.write(var);
         var = fin.read();
      }

      //fermeture propres des flux
      fin.close();
      fout.close();
   }
}
```

Remarques :

- Pour que ce programme s'exécute sans problème, il est impératif que les fichiers C:/tmp/bidule/file1 et C:/tmp/bidule/file2 existent physiquement sur la machine. Sinon une erreur se produira.

- !!! Attention !!! Lorsqu'on instancie un objet de type FileOutputStream (cf. FileOutputStream fout = new FileOutputStream(f2)), il vide automatiquement le fichier passé au constructeur. Elle doit donc être utilisée en bonne intélligence. Sa documentation recommande d'utiliser son constructeur `FileOutputStream(File file, boolean append)` où *append* sera positionné à *true* si l'on veut conserver le contenu du fichier et y appliquer des ajouts.

- La lecture de flux sur les fichiers peut générer des erreurs. Java permet de gérer cela en utilisant les **Exceptions** que nous aborderons au chapitre 11. Pour le moment si vous les utilisez dans vos programmes, ajoutez *throws IOException* sur toutes les méthodes concernées et importez la bibliothèque *java.io.IOException* dans votre classe

Lecture/Ecriture des données sur fichier

Les classes FileReader et FileWriter (1/2)

- Les classes FileReader et FileWriter se concentrent uniquement sur les fichiers texte et disposent chacune d'un constructeur qui prend en paramètre un **File** ou une **String** représentant le chemin absolu vers le fichier sur lequel elles vont chacune travailler

- **FileReader** :
  - Permet de lire les données caractère par caractère dans un fichier texte
  - Les méthodes principales de la classe :
    - `int read()` : lit le prochain caractère. La valeur du caractère est retournée comme un entier entre 0 et 255. Retourne -1 si la fin du fichier est atteint
    - `int read(char[] b)` : lit un certain nombre de caractères et les copie dans le tableau de char b. Le nombre de caractères lu est retourné. Retourne -1 si fin de fichier
    - `void close()` : ferme proprement le flux d'entrée vers le fichier

- **FileWriter** :
  - Permet d'écrire les données caractère par caractère dans un fichier texte
  - Les méthodes principales de la classe :
    - `void write(int c)` : écrit le caractère c dans le fichier texte
    - `void write(char[] b)` : écrit le tableau de caractères b dans le fichier texte
    - `void close()` : ferme proprement le flux de sortie vers le fichier

Lecture/Ecriture des données sur fichier

Les classes FileReader et FileWriter (2/2)

- Exemple : Le programme suivant lit les données du fichier file1.txt et les recopie dans le fichier file2.txt

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Exemple2LectureEcritureFichier {

  public static void main(String[] args) throws IOException {

     File f1 = new File("C:/tmp/bidule/file1.txt");
     File f2 = new File("C:/tmp/bidule/file2.txt");

     FileReader fr = new FileReader(f1);
     FileWriter fw = new FileWriter(f2);

     //lecture en boucle de tout le contenu du fichier file1.txt
     //et écriture dans le fichier file2.txt
     char[] b = new char[10];
     int var = fr.read(b);
     while(var != -1) {
         fw.write(b);
         var = fr.read(b);
     }

     //fermeture propres des flux
     fr.close();
     fw.close();
  }
}
```

Remarques :

- Exactement les mêmes remarques que celles de la slide 17

- Dans ce programme, on définit un tableau de 10 caractères que l'on passe à la méthode fr.read(b) pour lire des lots de 10 caractères dans le fichier file1.txt. Puis on invoque la méthode fr.write(b) pour écrire ce lot de 10 caractères, précédemment lus, dans le fichier file2.txt. On répète ainsi l'action jusqu'à ce qu'on atteigne la fin du fichier file1.txt, marquée par le fait que la méthode fr.read(b) retournera -1.

Lecture/Ecriture des données sur fichier

Les classes BufferedReader et BufferedWriter (1/2)

- Les classes BufferedReader et BufferedWriter se concentrent uniquement sur les fichiers texte et disposent chacune d'un constructeur qui prend en paramètre respectivement un **FileReader** et un **FileWriter** sur lequel elles vont chacune travailler

- **BufferedReader** :
  - Permet de lire les données caractère par caractère ou directement une ligne dans un fichier texte
  - Les méthodes principales de la classe :

    int read() : lit le prochain caractère et le place dans une zone tampon de la mémoire. La valeur du caractère est retournée comme un entier entre 0 et 65535. Retourne -1 si la fin du fichier est atteint

    int read(char[] b) : lit un certain nombre de caractères et les copie dans le tableau de char b placé dans la zone tampon. Le nombre de caractères lu est retourné. Retourne -1 si fin de fichier

    String readLine() : lit toute une ligne dans le fichier et la retourne sous forme d'une String dans une zone tampon de la mémoire

    void close() : ferme proprement le flux d'entrée vers le fichier

- **BufferedWriter** :
  - Permet d'écrire les données caractère par caractère ou directement des chaines de caractères dans un fichier texte
  - Les méthodes principales de la classe :

    void write(int c) : écrit le caractère c dans une zone tampon de la mémoire

    void write(String s) : écrit la chaine de caractères s dans une zone tampon de la mémoire

    void newLine() : écrit le caractère de retour à la ligne

    void close() : ferme proprement le flux de sortie vers le fichier

    void flush() : applique l'écriture effective des données du tampon dans le fichier texte

Lecture/Ecriture des données sur fichier

Les classes BufferedReader et BufferedWriter (2/2)

- Exemple : Le programme suivant lit les données du fichier file1.txt et les recopie dans le fichier file2.txt

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Exemple3LectureEcritureFichier {

   public static void main(String[] args) throws IOException {

      FileReader fr = new FileReader("C:/tmp/bidule/file1.txt");
      FileWriter fw = new FileWriter("C:/tmp/bidule/file2.txt");

      BufferedReader bf = new BufferedReader(fr);
      BufferedWriter bw = new BufferedWriter(fw);

      String ligne = bf.readLine();
      while(ligne != null) {
          bw.write(ligne);
          bw.newLine();
          ligne = bf.readLine();
      }
      bw.flush();

      //fermeture propres des flux
      fr.close();
      fw.close();
      bf.close();
      bw.close();
   }
}
```

Remarques :

- Exactement les mêmes remarques que celles de la slide 17

- L'avantage de l'utilisation des BufferedReader/Writer réside dans le fait qu'on s'affranchit des tailles préalablement fixées lorsqu'on veut lire plusieurs caractères dans un fichier (cf. méthodes read(char[] b) et write(char[] c) des classes FileReader et FileWriter). Avec les BufferedReader/Writer, Java prépare pour chacune une zone mémoire de très grande taille pouvant réceptionner des milliers de caractères lus (ou écrits) . Ce qui a pour conséquence de traiter des fichiers de très grandes tailles beaucoup plus rapidement et efficacement.

Lecture/Ecriture des données sur fichier

Autres classes pour la lecture/écriture dans les fichiers

Il existe beaucoup d'autres classes pour la gestion des entrées/sorties dans les fichiers, chacune apportant sa spécificité :

- DataInputStream
- ObjectInputStream
- InputStreamReader
- DataOutputStream
- ObjectOutputStream
- OutputStreamWriter
- PrintWriter


SUCCESS