1. JAVA : Compilation et exécution

Une classe par fichier, le fichier porte le nom de la classe postfixé par .java, le nom de la classe commence TOUJOURS par une majuscule.
Compilation d'une classe :

```
javac NomDeClasse.java
```

Exécution :

```
java NomDeClasseContenantLaMethodeMain
```

2. Expressions

Commentaires

```
// fin de ligne commentée
/* commentaire borné aux extrémités */
/** pour javadoc */          pour générer de la doc de façon automatique
```

Constantes :

```
-4                                  // cte entière de type int
4L                                  // entier long de valeur 4
0777                                // cte octale
0XFF                                // hexadécimal
10e45 ou .36E-2 ou 5.62             // cte virgule flottante double
2.56f                               // cte float
2.56                                // cte double
```

Boolean : true ou false
Caractères :

```
'a'         // caractère a
\n          // <RC, LF>        \r      // passage à la ligne
\t          // tab             \f      // saut de page
\b          // back            \\      // anti slash
\'          // guillemet simple \"     // guillemet double
\d          // octal           \x      // hexadécimal
\u          // unicode (symboles codés)
```

Constantes chaînes :

```
"ceci est une chaîne"
""          // chaîne vide
"ceci est un chaîne avec \" guillemets \" à l'intérieur"
\u2122      // ™ (trade mark)
"concaténation " + "de 2 chaînes" équivaut à "concaténation de 2 chaînes"
```

Opérateurs et expressions :

```
+       -       *       // opérateurs arithmétiques surchargés
/           // int / int -> int ; float / int -> float ; float / float -> float
%           // modulo
==          // ATTENTION syntaxe du test d'égalité identique à C  !!!!
!=          // différent
<       <=      >       >=
&&          // ET logique
||          // OU logique
!           // NON logique
^           // XOR bit à bit
~           // NOT bit à bit
<<          // décalage à gauche bit à bit
>>          // décalage à droite bit à bit
>>>         // décalage à droite avec remplissage avec des 0 bit à bit
&           // ET bit à bit
|           // OU bit à bit
new         création d'une nouvelle instance de classe
```


3. Instructions

Toute instruction est **toujours** terminée par un point virgule **;**

```
i = 1;                  // affectation
int i;                  // déclaration
int i = 1;              // déclaration avec affectation d'une valeur initiale à la variable
titeuf.age = 10;        / initialisation d'une variable d'instance
```

Affichage à l'écran d'un message composé de la concaténation de chaînes de caractères et de valeurs numériques converties automatiquement en chaînes de caractères.

```
System.out.print ("texte entre les guillemets " + entier1 + "...." + string1 + flottant1 + .... + " \n ");
```

"\n" permet de passer à la ligne après l'impression de la ligne.
Sinon utiliser println (voir la classe java.io.PrintStream de la javadoc)

Bloc d'instructions

```
{           // l'utilisation la plus courante des blocs est lorsqu'il y a plus d'une instruction
  ...       // contrôlée par une structure de contrôle.
}
```

Instruction conditionnelle

```
if ( cond ) {
   instructions;                    // ou une seule instruction, sans bloc
} else {
   instructions;                    // ou une seule instruction, sans bloc
}
```

Opérateur conditionnel

```
test ? true_result : false_result; // test true => exécution de true_result, false_result sinon
int smaller = x < y ? x : y;
```

Branchement conditionnel

```
switch (variable) {         // type de base autorisé : byte, char, short, int, long
    case v1 : {
          ...               // bloc d'instructions
    }
    break;
    case v2 :
       instruction;
    break;
    default : instruction;
}
```

Boucle for

```
for (cond. initiales; cond d'arrêt; expression devant faire évoluer vers la cond d'arrêt) {
   instructions;
}
for (int i = 0; i < 10; i ++) {
    // instructions à exécuter
}
```

Boucle « foreach »

```
int[] tableau;          // tableau d'entiers [ou collection]
for ( int val : tableau) {      // pour tts les val. (ici int) contenues dans le tableau [ou collection]
    // traitement utilisant val
}
```

Boucle while

```
while (condition) {
   instructions;
}
```

Boucle do - while

```
do {
...;
} while (cond);
```

4. Variables et types

4.1. Variables

Trois types de variables :

- Les **variables locales** sont déclarées et utilisées dans les blocs.
- Les **variables de classe** dont les valeurs sont stockées au niveau de la classe et sont connues par toutes les instances de la classe.
- Les **variables d'instance** sont stockées au niveau de l'instance et représentent les attributs de la classe. Elles constituent **l'état** d'une instance donnée.

On ne trouve pas de variables globales.

Déclaration de variables :
type nomDeVariable = valeurInit;
type n1 = v1, n2 = v2, n3 = v3;
Toute variable locale doit être initialisée avant d'être utilisée.

Les variables qui sont des instances de classe sont initialisées à **null (référence nulle)** par défaut à la déclaration.
Les variables d'instance et de classe sont initialisées par défaut à la création de l'instance :

valeur numérique :	entière : 0	réelle : 0.0
caractère :	caractère nul : '\0'	
valeur booléenne :	false	

Convention de noms de variables :

- une variable commence **toujours** par une minuscule,
- si elle est composée de plusieurs mots, le premier est en minuscule et les autres en majuscule,
- seuls les noms de classe commencent par une majuscule.

Exemple : Button theButton; *// déclaration de l'instance theButton de classe Button*

4.2. Types

8 types de données de base :

byte	(octet)	float	(4 octets)
short	(2 octets)	double	(8 octets)
int	(4 octets)	char	(2 octets non signés)
long	(8 octets)	boolean	(true, false)

Toute classe est un ***type*** permettant la déclaration d'instances :

```
String  lastName;   // lastName instance de String
Font    basicFont;  // basicFont instance de Font
```


4.3. Constantes (mot clef final)

Les constantes sont déclarées dans les classes ou en local :

```
final float PI = 3.141592f;
(ou mieux, accéder à la constante définie dans la classe Math de java.lang :
double Math.PI)
final int MAX_SIZE = 4000000;
```

On peut déclarer une constante sans l'initialiser à la déclaration et l'initialiser **une seule fois** dans chaque constructeur.
Une constante s'écrit tout en majuscule avec les mots séparés par un souligné : '_'.

4.4. Tableaux

Les tableaux sont typés. Le type des éléments du tableau est celui du type précisé à la déclaration (ici int).
Il faut déclarer une variable (ici hits) qui stocke l'adresse du tableau (nulle à la déclaration) :

```
int  hits [ ];    // ou
int  [ ] hits;    // deux écritures possibles, [] à droite ou à gauche  de la variable
```

Il faut ensuite **allouer** le tableau et affecter son adresse à la variable.

ATTENTION de la même façon qu'en C les index pour un tableau de 10 éléments vont de 0 à 9 !!

Un tableau peut contenir des variables de base numériques ou booléennes :

```
int  hits [ ];          // tableau d'entiers, se comporte comme un tableau en C
```

Il peut également contenir des références d'instances de classes :

```
String chaines [ ];     // tableau de String, contiendra après allocation des références d'instances
                        // de String, qui seront au départ toutes ces références sont nulles
```

4.4.1. Allocation de tableaux

```
// tableau de 10 références nulles vers 10 String      (et non une chaîne de 10 carac!)
String  names [ ] = new String [10];
// tableau de 2 références vers des Point, créés et initialisées en même temps que le tableau
Point hits [2] = new Point { {10, 20}, {11, 21} };
// tableau de 3 références vers 3 String, créées et initialisées en même temps que le tableau
String riz [ ] = {"rond", "long", "parfumé"};          // initialisé par les données
```

4.4.2. Accès à un élément

```
names.length  : attribut du tableau, fournit le nombre d'éléments alloués pour le tableau
hits [0].x    : fournit la valeur 10
hits [1].y    : fournit la valeur 21
```

ATTENTION vérification de dépassement de capacité, peut lever une exception.

```
names [10] = "toto"  => erreur de compilation ou d'exécution si l'indice est évalué à l'exécution.
int len = names.length;        // len recevra la valeur 10
```

4.4.3. Multidimensionnels

```
int coords [ ][ ] = new int [4][4];
...
coords [0][0] = 15;
coords [0][1] = 12;
```

5. Entrées / Sorties

5.1. Introduction

Les entrées/sorties concernent tout ce qui se rapporte :

- à la lecture d'informations émises par une source externe,
- à l'envoi d'informations à une destination externe.

Suivant les cas, cette source ou cette destination peuvent être : le réseau, un fichier sur le disque dur en local, un autre programme s'exécutant en parallèle, ou bien sûr le clavier et l'écran...
Dans les programmes Java, les entrées sorties sont gérées par les objets de flux.

5.1.1. Les « flux » (« Streams »)

Un flot est un canal de communication dans lequel les données sont écrites ou lues de manière séquentielle.

- Pour lire des informations, il suffira d'ouvrir un «flux en lecture » sur la source. Ce flux permettra de lire les informations de manière séquentielle.
- Au contraire, pour envoyer des informations, il suffira d'ouvrir un « flux en écriture » sur la destination. Les informations seront écrites encore une fois de manière séquentielle.

5.1.2. Les classes d'entrées/sorties

	Entrée	**Sortie**
Binaire (manipulent des octets)	InputStream	OutputStream
Texte (manipulent des caractères Unicode 16 bits)	Reader	Writer

Table 1.Les classes d'entrées/sorties les plus courantes sont définies dans le paquetage java.io

5.2. Les flux à disposition dans la librairie Java

Les flux sont organisés en hiérarchie. Tout en haut ce cette hiérarchie, se trouvent 4 superclasses abstraites : «Reader», «Writer», «InputStream» et «OutputStream».

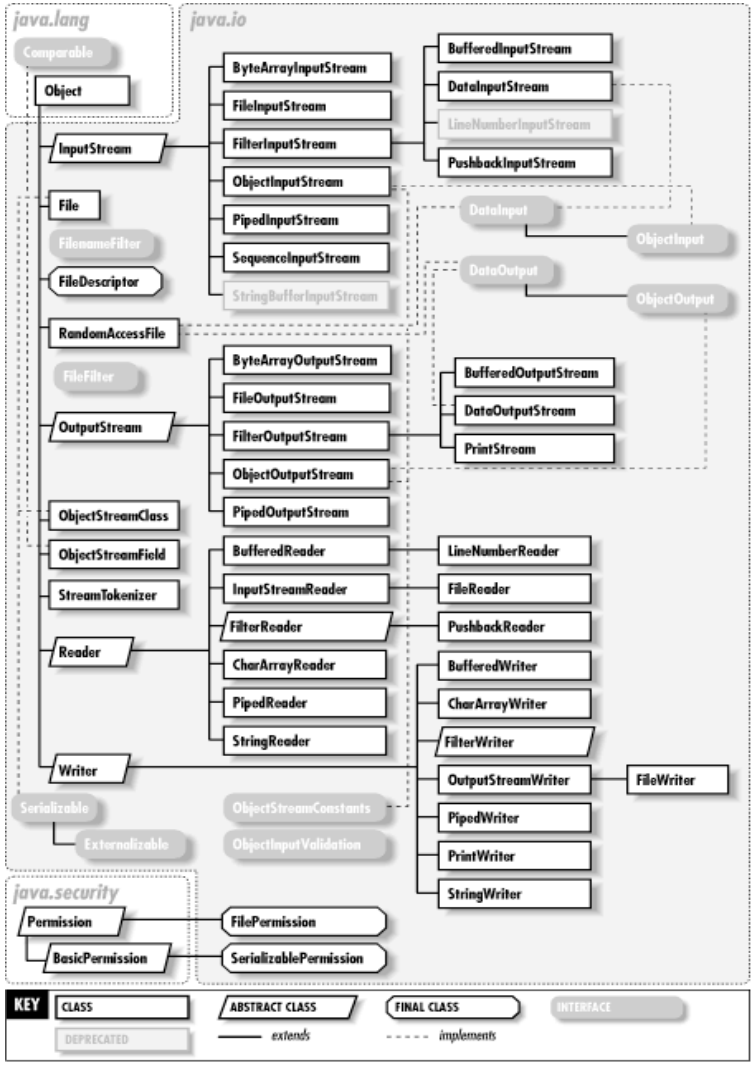
	Entrée	**Sortie**
Binaire	ObjectInputStream	ObjectOutputStream
Texte	BufferedReader	PrintWriter

Table 2.Les classes d'entrées/sorties les plus utilisées

	Entrée	**Sortie**
Binaire	read()	write()
Texte	readLine()	println()

Table 3.Les classes d'entrées/sorties les plus utilisées

Le paquetage java.io contient également la classe File qui permet de gérer tous les accès aux informations relatives au système de fichiers.



Classes de java.io. Schéma extrait de :

O'REILLY®

JAVA™ IN A NUTSHELL *Third Edition*

5.3. Ecran / clavier

5.3.1. Affichage à l'écran

La classe System possède le stream de sortie standard, appelé simplement : out.
Il est possible d'afficher directement toute variable d'un type de base en le passant en paramètre de la fonction print ou println (passage à la ligne après affichage).
Pour une instance, la méthode toString( ) définie dans la classe Object permet de retourner une chaîne de caractères représentative de l'état de l'instance.
Si la méthode ne convient pas il faut la redéfinir dans la classe que l'on désire afficher.
Toute classe héritant de la classe Object, l'utilisation de toString( ) dans une structure polymorphe se fera sans problème.

Pour générer une chaîne de caractères appel plusieurs entités de type différent, il faut les donner dans l'ordre d'affichage, simplement séparées par un + (symbole de concaténation).

System.out.print ("un certain texte " + unEntier + " ou " + unFloat + " ou " + unObjet.toString( ));

5.3.2. Lecture au clavier

Illustré à partir d'un exemple qui montre l'usage que l'on peut faire des classes de java.io et java.lang.System.

La classe System possède le stream d'entrée standard, appelé simplement : in.
Il est utilisé pour instancier la classe InputStreamReader qui va récupérer les entrées clavier.
L'instance ainsi créée (bien que n'ayant pas été stockée dans une variable d'instance) va être utilisée pour créer une instance de BufferedReader qui va stocker dans un buffer les entrées clavier.

Il est possible de créer une instance "virtuelle", dont la référence n'est pas stockée pour être ensuite explicitement libérée, grâce au garbage collector qui assure la libération de la place mémoire allouée dès que la référence n'est plus utilisée (!!à ne pas faire en C++!!).


```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class ReaderKeyboard {
  // L'instance myInput sait lire au clavier et stocker les informations lues
  // Pour pouvoir lire au clavier, il faut instancier
  // l'InputStreamReader avec System.in (entree standard)
  // et instancier BufferedReader avec l'InputStreamReader
  BufferedReader myInput = new
  BufferedReader(new InputStreamReader(System.in));
  // Toute saisie est stockee dans un String.
  String c;

  public ReaderKeyboard() {
    myInput = new BufferedReader(new InputStreamReader(System.in));
  }
  // la methode est susceptible de retourner une Exception
  public int readInt() throws IOException, NumberFormatException {
    // recuperation de l'entree clavier dans un String
    c = myInput.readLine(); // peut lever une IOException
    // la cha”ne est convertie en Integer
    // dont on extrait ensuite la valeur entiere de type de base int
    return Integer.valueOf(c).intValue(); // peut lever NumberFormatException
  }
  // la methode est susceptible de retourner une Exception
  public float readFloat() throws java.io.IOException, NumberFormatException {
    // Version plus condensee mais pas toujours plus lisible ...
    return Float.valueOf(myInput.readLine()).floatValue();
  }
  public static void main(String args[]) {
    // L'instance myInput sait lire au clavier et stocker les informations lues.
    // Pour pouvoir lire au clavier, il faut instancier
    ReaderKeyboard rk = new ReaderKeyboard();
    int i = 0;
    float f = 0.0f;
    boolean error;
    System.out.print("Entrer un entier : ");
    do {  // saisie protegee
      try {
        i = rk.readInt();
        error = false;
      } catch (Exception e) { // peu importe le type de l'exception ...
        System.out.print("Entrer un entier : ");
        error = true;
      }
    } while(error);
    System.out.print("Entrer un flotant : ");
    try {
      f = rk.readFloat(); // saisie non protegee
    } catch(IOException ioe) { // bloc try-catch obligatoire
      // pas de traitement ...
      System.out.print("Erreur de saisie ");
    }
    System.out.println("resultat " + i + "  et  " + f);
  }
}
```

Table 4.Exemple de code pour effectuer une écriture et une lecture au clavier

5.3.3. La classe Scanner : Lecture au clavier simplifiée

La classe Scanner permet de simplifier les lectures au clavier par encapsulation des mécanismes montrés plus haut.

Exemple :

```
import java.util.Scanner;

public class TestScanner {
   public static void main (String args[]) {
      //créee une instance connectée à un flot d'entrée,
      //ici l'entrée standard System.in
      Scanner entree = new Scanner(System.in);

      // Lecture d'un entier
      System.out.print("i = ");
      int i = entree.nextInt();
      // Lecture d'un flottant
      System.out.print("x = ");
      float x = entree.nextFloat();
      // Lecture d'une chaîne
      System.out.print("s = ");
      String s = entree.nextLine();

      // La classe Scanner suit les particularités régionnales :
      // nombres décimaux avec une virgule à la place du point.
      // Idem avec printf, mais pas avec print et println.
      System.out.printf(String.format("i = %d, x = %f - une ','-, s = %s\n", i, x, s));
      System.out.println("i = " + i + ", x = " + x + " -un '.'-, s = " + s);
      System.out.println("Ca génére un pb ...");
      // Lecture d'un flottant
      System.out.print("x = ");
      x = entree.nextFloat();
      // Lecture d'une chaîne corrigée
      entree.skip("\n");
      System.out.print("Il a fallu un skip\ns = ");
      s = entree.nextLine();
      System.out.println("i = " + i + ", x = " + x + " -un '.'-, s = " + s);
      System.out.print("là, pas besoin du skip ... \ns = ");
      s = entree.nextLine();
      System.out.println("i = " + i + ", x = " + x + " -un '.'-, s = " + s);

   }
}
```

Table 5.Exemple de code pour lire au clavier avec la classe Scanner

5.4. Entrées/Sorties dans un fichier

5.4.1. Fichier texte

Pour écrire dans un fichier, il faut disposer d'un flux d'écriture.

5.4.2. PrintWriter

Pour écrire dans un fichier texte, il est préférable d'utiliser la classe PrintWriter dont les constructeurs les plus souvent utilisés sont les suivants :

- PrintWriter(Writer out) l'argument est une instance de Writer, d'une classe qui hérite de Writer, c.a.d. un flux d'écriture.
- PrintWriter(Writer out, Boolean autoflush) idem.
  Le second argument gère la mise en buffer des lignes. Lorsqu'il est à faux (son défaut), les lignes écrites sur le fichier sont stockée dans un buffer en mémoire. Lorsque celui-ci est plein, le buffer est vidé dans le fichier. Cela permet de diminuer les accès disque.

Les méthodes utiles de la classe PrintWriter sont les suivantes :

- void print(Classe T) écrit la donnée T qu'elle soit de classe String, int, etc.
- void println(Type T) idem en terminant par une marque de fin de ligne.
- void flush() vide le buffer si on n'est pas en mode autoflush.
- void close() ferme le flux d'écriture.

5.4.3. Lecture dans le fichier texte

Pour lire le contenu d'un fichier, il faut disposer d'un flux de lecture associé au fichier. On peut utiliser pour cela la classe FileReader et le constructeur suivant :

- FileReader(String nomFichier) ouvre un flux de lecture à partir du fichier indiqué. Lance une exception si l'opération échoue.

La classe FileReader possède un certain nombre de méthodes pour lire dans un fichier, méthodes héritées de la classe Reader. Pour lire des lignes de texte dans un fichier texte, il est préférable d'utiliser la classe BufferedReader avec le constructeur suivant :

- BufferedReader(Reader in) ouvre un flux de lecture bufferisé à partir d'un flux d'entrée in.

Les méthodes utiles de la classe BufferedReader sont les suivantes :

- int read() lit un caractère
- String readLine() lit une ligne de texte
- int read(char[] buffer, int offset, int nb) lit nb caractères dans le fichier et les met dans le tableau buffer à partir de la position offset.
- void close() ferme le flux de lecture

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class IOStandard {
  public static void main(String []args) {
    try {
       // Ecriture dans un fichier texte
       PrintWriter pw = new PrintWriter("toto.data");
       pw.println(name);
       pw.println(age);
       pw.close();

       // Lecture dans le fichier texte
       BufferedReader fIn=new BufferedReader(new FileReader("toto.data"));
       name = fIn.readLine();
       age = Integer.parseInt(fIn.readLine());
       System.out.println("A partir du fichier toto.data : "+name+" de "+age+" ans.");
    } catch (FileNotFoundException fnfe) {
       System.out.println("le fichier n'existe pas");
    } catch (IOException ioe) {
       System.out.println("erreur avec le parseInt ou le readLine");
    }
  }
}
```

Table 6.Exemple de code pour effectuer une écriture et une lecture dans un fichier texte

5.4.4. Fichier d'instances (fichier binaire)

Démonstration par l'exemple :
Voici une classe qui va écrire dans un fichier une structure de données (ici une liste implantée dans un Vector) et la récupérer.

> ATTENTION : la sauvegarde d'instances de classes pour fonctionner correctement implique que la classe déclare : implements Serializable qui ne nécessite la mise en œuvre d'aucune méthode mais peut être vue comme le positionnement d'un état.

```
import java.io.Serializable;

public  class LigneComptable implements Serializable {
  // attributs
  /** Nom du Produit ou de la Personne */
  protected String nom;
  /** Total de la transaction */
  protected Euro total = new Euro(0.0);
  /** Quantite de Produit vendu ou annule*/
  protected int quantite = 0;
  /** Prix unitaire du Produit concerne*/
  protected Euro prixUnitaire = new Euro(0.0);
 // ..........
}
```

Table 7.Classe LigneComptable

```
import java.util.ArrayList;
public class Comptes {
  protected ArrayList comp;               // une liste de lignes comptables
   // ..........
  int soldeInit; //....
  public void writeOnFile(File file) {    // "LignedComptables.dat"
    try {    // fichier cree vide
        FileOutputStream fos = new FileOutputStream(file);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(soldeInit);
        for(int i = 0; i < comp.size(); i++)
          oos.writeObject(comp.get(i));
        // placer un element null pour signaler la fin
        oos.writeObject(null);
        oos.close();
    } catch (Throwable t) {
        System.out.println("----->erreur de fichier compte <----"+t);
    }
  }
}
```

Table 8.Classe Comptes, qui a comme attribut un ArrayList de LigneComptable


```
 /** Methode principale */
  public static void main(String[] args) {
    Comptes cpt;
    FileReader sourceFile;
    File file;
    FileInputStream fis;
    ObjectInputStream ois;
    LigneComptable lc;
    ArrayList sddlc = new ArrayList();
    boolean termine = false;

    try {
      file = new File(new String("LignesComptables.dat"));
      Object obj;
      if (file.exists() && file.isFile())  {
        fis = new FileInputStream(file);
        if (file.canRead()) {
          ois = new ObjectInputStream(fis);
          while (!termine) {
            obj = ois.readObject();
            if (obj == null)
              termine = true;
            else
              sddlc.add((LigneComptable)obj);
          }
          ois.close();
        }
      }
//................
    } catch (Throwable t) {
        System.out.println("----->erreur de fichier compte <-----"+t);
        System.exit(0);
    } // catch
  } // main
```

Table 9.Ecriture et lecture d'instances dans un fichier binaire

Table des matières