***SOMMAIRE***

# *INTRODUCTION*

Cette première partie traite des bases du langage JAVA. Les programmes réalisés dans ce support fonctionneront en mode texte (dans une fenêtre MS-DOS, par exemple).

Ce support de formation est constitué d'une liste d'exercices permettant de s'approprier les différentes difficultés du langage JAVA. Chaque exercice est structuré de la façon suivante :

- Description des objectifs visés.
- Explications des techniques à utiliser (Ce qu'il faut savoir).
- Enoncé du problème à résoudre (Travail à réaliser).
- Renvois bibliographiques éventuels dans les ouvrages traitant de ces techniques (Lectures).

Tous ces exercices sont corrigés et commentés dans le document intitulé :

- Proposition de corrigé JAVA_APC
  Apprentissage d'un langage de Programmation Orientée Objet
  JAVA (bases).

Les aspect "Orienté Objet" et graphique ne sont pas traités dans ce support. Ils feront l'objet des parties suivantes:

- La deuxième partie traitera des concepts de la programmation orientée objet en langage JAVA. Elle permettra la construction d'objets en JAVA correctement intégrés dans les bibliothèques standards existantes.
- La troisième partie traitera de la construction d'Applet, composants actifs graphiques, destinées à être incorporées à des pages HTML.

Les ouvrages auxquels il sera fait référence dans le présent support sont les suivants :

- **JAVA (Le Macmillan)**
  Alexander Newman
  Editions Simon & Schuster Macmillan

- **Formation à JAVA**
  Stephen R. Davis
  Editions Microsoft Press

- **JAVA (Megapoche)**
  Patrick Longuet
  Editions Sybex

# T.P. N°1 - REGLES ET CONVENTIONS DE SYNTAXE

## 1.1 OBJECTIFS

- Les règles et les conventions de syntaxe et de présentation d'un programme en langage JAVA.
- La classe application.
- Comment déclarer la fonction main, point d'entrée de l'application.
- Comment déclarer des variables et des constantes, comment les utiliser dans le corps d'un programme.
- Comment afficher à l'écran à travers les flux standards.

## 1.2 CE QU'IL FAUT SAVOIR

### 1.2.1 Règles de syntaxe

#### La classe application

Même si les concepts objets ne seront traité que dans la deuxième partie de ce support, il est nécessaire d'introduire ici le concept de classe ainsi que le mot clé du langage JAVA qui permet de l'implémenter.

Ainsi donc, un programme JAVA est d'abord constitué par une classe particulière appelée classe application. Cette classe se déclare par le mot clé **`class`** de la façon suivante :

```
public class JTP1
{
    Déclaration des fonctions des applications
}
```

Le nom de la classe (ici **JTP1**), est choisi par le programmeur. La construction d'un nom de classe obéit à la règle standard régissant tous les identificateurs du langage JAVA (voir ci-dessous).

#### Le programme principal

Une application JAVA possède toujours au moins une fonction **`main class`** . Cette fonction est obligatoire et constitue le point d'entrée **`class`** du programme principal de l'application. Elle se déclare dans la classe de l'application de la façon suivante :

```
public class JTP1
{
    public static void main (String[] args)
    {
        Instructions
    }
}
```

- La fonction **`main`** est le point d'entrée du programme. Elle doit s'appeler obligatoirement **`main`** et doit être suivie par des parenthèses ouvrante et fermante encadrant les arguments (comme toutes les fonctions en JAVA).

- La fonction **`main`** reçoit du système d'exploitation un seul argument (**`args`**) qui est un tableau de chaînes de caractères, chacune d'elles correspondant à un paramètre de la ligne de commande (unix ou MS-DOS).

- Le corps du programme suit la fonction **`main`** entre deux accolades ouvrante et fermante. Les instructions constituant le programme sont à insérer entre ces deux accolades

## Les variables

- Les variables peuvent être déclarées à tout moment à l'intérieur du corps du programme. Cependant, la déclaration et l'initialisation d'une variable doit impérativement précéder la première utilisation de celle-ci.

- Chaque déclaration de variables est construite sur le modèle suivant :
```
TypeDeLaVariable   nomDeLaVariable ;
```

- Une variable peut être initialisée lors de sa déclaration :
```
TypeDeLaVariable nomDeLaVariable = valeurInitiale;
```

- Les variables ne sont visibles (reconnues par le compilateur) que dans le bloc d'instruction (défini par **`{`** et **`}`**) dans lequel elles sont définies.

- Comme tous les autres identificateurs (identificateur de classe, entre autres) en langage JAVA, les noms de variables peuvent être choisis librement par le programmeur (sauf parmi les *mots-clefs*). La liste des mots clés se trouve en annexe A. Seuls les 32 premiers caractères sont significatifs (interprétés par le compilateur). Les caractères suivants peuvent être utilisés :
  - de **`A`** à **`Z`**
  - de **`a`** à **`z`** (les minuscules sont considérées comme des caractères différents des majuscules)
  - de **`0`** à **`9`**
  - le caractère souligné **`_`** et le caractère dollar **`$`** (même en initiale).

- Comme toutes les instructions du langage JAVA, chaque déclaration de variable DOIT absolument être terminée par un point-virgule **`;`** Le point-virgule ne constitue pas un séparateur, mais plutôt un *terminateur* d'instructions.

## Les constantes

Les constantes se déclarent comme les variables initialisées précédées du mot-clef **`final`**. Leur valeur ne pourra pas être modifiée pendant l'exécution du programme.

```
final TypeDeLaConstante nomDeLaConstante = valeurInitiale;
```

## Les instructions

- Le caractère **`;`** est un *terminateur*. TOUTES les instructions doivent se terminer par un **`;`**.

- Les différents identificateurs sont séparés par un *séparateur*.
- Les *séparateurs* peuvent être indifféremment l'*espace*, la *tabulation* et le *saut de ligne*.

**Les commentaires**

- Les caractères compris entre **/*** et ***/** ne seront pas interprétés par le compilateur.
- Les caractères compris entre **//** et un *saut de ligne* ne seront pas interprétés par le compilateur.
- Les caractères compris entre **/**** et ***/** ne seront pas interprétés par le compilateur. Le texte de commentaire peut être utilisé **javadoc**, un utilitaire du JDK (Java Development Kit) qui permet de générer automatiquement la documentation des classes.

### 1.2.2 Conventions d'écriture

Ces conventions ne sont pas des contraintes de syntaxe du langage JAVA. Elles n'existent que pour en faciliter la lecture.

- Une seule instruction par ligne. Même si tout un programme en langage JAVA peut être écrit sur une seule ligne.
- Les délimiteurs d'un bloc **{** et **}** doivent se trouver sur des lignes différentes et être alignés sur la première colonne de sa déclaration.
- A l'intérieur d'un bloc **{ }** les instructions sont indentées (décalées) par un caractère *tabulation*.
- A l'intérieur d'un bloc **{ }** la partie *déclaration des variables* et la partie *instructions* sont séparées par une ligne vide.
- Les identificateurs de variables sont écrits en minuscule et sont préfixés :

| | | |
|---|---|---|
| **b** | booléen (vrai/faux) | **(bOK)** |
| **n** | nombre entier | **(nAge)** |
| **l** | nombre entier long | **(lAge)** |
| **d** | nombre en virgule flottante | **(dSalaire)** |
| **i** | itérateur (dans une boucle) | **(iCpt)** |
| **str** | chaîne de caractères | **(strNom)** |

etc.

*N.B. Vous verrez dans la suite du cours la signification de ces termes.*

### 1.2.3 Entrées/Sorties standards

Le langage JAVA est un langage orienté objet. Dans ce type de langage, toutes les entrées/sorties se font à travers des *flux*. En langage JAVA, on utilise les methodes des classes **PrintStream** et **InputStream** pour lire et écrire sur les périphériques :

- Le périphérique **System.out** est l'instance de la classe **PrintStream** qui correspond au périphérique standard de sortie (écran).
  ```
  System.out.println(k);
  ```
  Le contenu de la variable **k** est affiché à l'écran.

- Le périphérique **System.in** est l'instance de la classe **InputStream** qui correspond au périphérique standard d'entrée (clavier).
  ```
  byte bArray[] = new byte[80];
  System.in.read(bArray);
  ```
  Les caractères tapés au clavier seront rangés dans le tableau de caractères **bArray**.

Les entrées au clavier font appel à la gestion des exceptions. Cet aspect complexe de la programmation objet sera abordé dans la deuxième partie de ce support. Par conséquent, il faudra admettre, pour les exercices proposés dans cette première partie qui feront appel à des fonctions de lecture du clavier, d'ajouter l'instruction **throws IOException** à la déclaration des fonctions pour qu'elle puisse être compilée sans générer d'erreur. Par exemple, la fonction **main** devra être déclarée :

```
public static void main (String[] args)
throws IOException
{
   // Instructions
}
```

Par ailleurs, la fonction **read** de la classe **InputStream** est très rudimentaire et ne permet que de lire un tableau d'octets. Il faudra donc admettre les syntaxes suivantes pour lire des données plus structurées comme les chaînes de caractères, les entiers ou le nombres en virgule flottante :

```
byte bArray[] = new byte[80];

// Pour lire la chaîne de caractères strNom
System.in.read(bArray);
String strNom = new String(bArray, 0, 80);

// Pour lire l'entier k déclaré de type int
System.in.read(bArray);
String strLine = new String(bArray, 0, 80);
k = (new Integer(strLine)).intValue();

// Pour lire le nombre dSalaire déclaré de type double
System.in.read(bArray);
String strLine = new String(bArray, 0, 80);
dSalaire = (new Double(strLine)).doubleValue();
```

- Le périphérique **System.err** est également une instance de la classe **PrintStream** qui correspond au périphérique standard de sortie (écran) Elle est utilisée pour l'affichage des message d'erreur. Elle permet de désolidariser l'affichage des messages d'erreur ou de service de l'affichage standard en cas de redirection de ce dernier vers un autre périphérique (imprimante par exemple).
  ```
  System.err.println(k);
  ```
  Le contenu de la variable **k** est affiché à l'écran via le flux **err**.

### 1.2.4 Utilisation des fonctions des bibliothèques

Le langage JAVA est un langage peu implémenté. C'est-à-dire qu'il n'existe que très peu de mots réservés, une cinquantaine, dont la liste est publiée dans la documentation du compilateur.

Par contre, il existe plusieurs bibliothèques de classe ou *package* qui offrent au programmeur un grand nombre de fonctionnalités. Ces *packages* sont utilisé grace à.

Pour pouvoir utiliser les fonctions ou les objets des *packages*, il faut que celles-ci soient reconnues par le compilateur (qui ne les connaît pas par défaut). C'est le rôle de l'instruction **`import`**. Au moins un *package* doit être déclaré, le *package* **`java.lang`**, ne serait-ce que pour pouvoir déclarer la classe de l'application ou utiliser la classe **`System`** et les flux **`in`** et **`out`** pour effectuer les entrées/sorties. C'est pourquoi on commence toujours un programme jJAVA par l'instruction suivante :

```
import java.lang.*;
```

Comme le *package* **`java.lang`** est obligatoire, la plupart des compilateurs JAVA considèrent que cette instruction est implicite.

Le choix du *package* à importer dépend, bien sur, des classes d'objet utilisées. Ces informations peuvent être trouvées dans la documentation des classes livrée avec le compilateur.

### 1.2.5 Compilateur JAVA

Les outils les plus simples, pour programmer en JAVA, sont les outils du JDK (Java Development Kit) de SUN. Une version existe pour Windows 95 ou 98.

- JAVAC C'est le compilateur proprement dit.
- JAVA C'est la machine virtuelle JAVA qui permet de "jouer" les programmes JAVA compilés par JAVAC.

Les programmes rédigés en langage JAVA sont rangés dans des fichiers textes. Ces fichiers peuvent être créés à l'aide de n'importe quel éditeur de texte (comme le bloc-notes de Windows) et doivent avoir l'extension **`.java`**.

Les fichiers sources JAVA ne sont pas exécutables. Auparavant, ils doivent être compilés. Pour cela on utilise l'utilitaire **`JAVAC`** du JDK. Le compilateur traduit chaque classe JAVA dans un fichier dont l'extension est **`.class`** :

```
C:\>JAVAC JTP1.CLASS<enter>
```

A l'issus de la compilation, les fichiers compilés prennent le nom de la classe et non pas le nom du fichier source. Par exemple, si le fichier **`toto.java`** contient le code des classes **`tata`** et **`titi`**. La compilation de **`toto.java`** générera les fichiers **`tata.class`** et **`titi.class`**.

En principe, on utilise un fichier source pour chaque classe. Le nom du fichier est, par convention, le nom de la classe. Par exemple, le fichier **`JTp1.java`** contient le

code source de la classe **`JTp1`**. La compilation générera le fichier **`JTp1.class`**.

A l'issue de la compilation, le fichier **`.class`** obtenu n'est pas directement exécutable. Il ne peut être exécuté que dans la machine virtuelle JAVA. Pour exécuter l'application **`JTp1`**, il faut taper :

```
C:\>JAVA JTP1.class<enter>
```

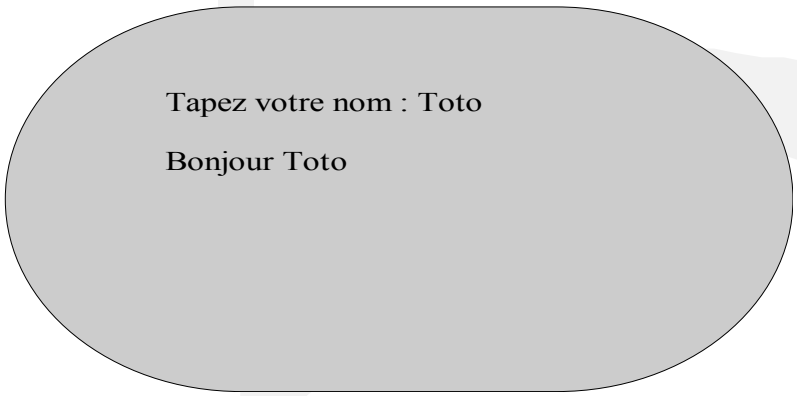
La machine virtuelle JAVA va chercher, dans le fichier **`JTP1.class`**, une fonction **`main`** pour en commencer l'exécution. Si une telle fonction n'existait pas, l'exécution serait impossible.

Pour plus de précision sur le JDK, lire :

- **JAVA (Le Macmillan)**
  Ch. 2 : L'environnement JAVA
  Ch. 3 : Le compilateur JAVA

## *1.3 TRAVAIL À RÉALISER*

En respectant les conventions d'écriture du langage JAVA, rédiger un programme permettant à un utilisateur de saisir son nom et de répondre **Bonjour** :




### *FONCTIONS À UTILISER*

| Fonction | Langage JAVA | Classe | Package |
|---|---|---|---|
| Flux d'affichage à l'écran | `System.out` | `PrintStream` | `java.lang` |
| Flux de lecture au clavier | `System.in` | `InputStream` | `Java.lang` |
| Lecture au clavier | `read` | `InputStream` | `java.lang` |
| Affichage à l'écran | `print` `println` | `PrintStream` " | `java.lang` " |

# T.P. N°2 - GESTION DES DONNÉES NUMÉRIQUES

## 2.1 OBJECTIFS

- La déclaration et l'utilisation des variables et des constantes numériques.
- La lecture et l'écriture des données numériques sur les flux standards et l'utilisation des manipulateurs.
- Les opérateurs de calcul.
- L'utilisation des fonctions mathématiques.
- La structure répétitive ***for***.

## 2.2 CE QU'IL FAUT SAVOIR

### 2.2.1 Les données numériques

De base, le langage JAVA ne permet de gérer que des types de données simples. Celles-ci peuvent être de type scalaire (entiers) ou virgule flottante (réels).

| | |
|---|---|
| **`boolean`** | Booléen, vrai (valeur **true**) ou faux (valeur **false**) |
| **`char`** | Un et un seul caractères. |
| **`byte`** | Entier signé sur 1 octet (-128 à +127) |
| **`short`** | Entier signé sur 2 octets (-32768 à +32767) |
| **`int`** | Entier signé 4 octets (-2147483648 à +2147483647) |
| **`long`** | Entier signé sur 8 octets (-9223372036854775808 à +9223372036854775807) |
| **`float`** | Réel virgule flottante au standard IEEE754 sur 4 octets (1.40239846E-45 à 3.40282347E+38) |
| **`double`** | Réel virgule flottante au standard IEEE754 sur 8 octets (4.94065645841246544E-324 à1.79769313496231570E+308) |

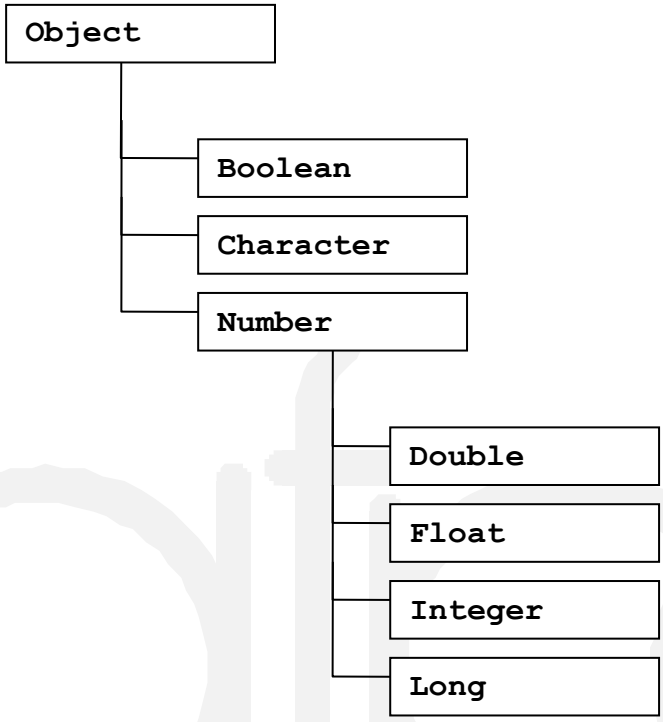
Les constantes numériques entières peuvent être écrites sous 3 formes : en décimal, en hexadécimal (base 16) et en caractères. Ainsi, 65 (décimal), 0x41 (hexadécimal), 'A' désignent la même constante numérique entière (le code ASCII de A est 65).

Pour plus de précision sur les types de données, lire :

- **JAVA (Le Macmillan)**
  Ch. 6 : Les jetons
  Ch. 7 : Les types

### 2.2.2 Objets numériques

*En JAVA, les types de données numériques ne constituent pas des classes d'objets (dans le concept de Programmation Orientée Objet). Il existe des classes dérivées de la classe racine **Object** correspondant à ces types de données. Ces classes ne peuvent être utilisées pour le calcul arithmétique. Par contre, elles sont équipées de méthodes permettant la conversion avec les types de données simples et l'interopérabilité entre elles.*




### 2.2.3 Affichage des données numériques

L'affichage des données numériques se fait à travers le flux de sortie **System.out**. à l'aide des méthodes **print** ou **println**.

En principe, les méthodes **print** et **println** ne peuvent recevoir que des paramètres de type **String**. Mais il existe une méthode de conversion automatique de tous les types numériques en chaîne de caractères. **print** et **println** ne pouvant recevoir qu'un seul paramètre à la fois, il est possible d'afficher plusieurs données en utilisant l'opérateur de concaténation de chaînes de caractères **+**.

Exemple:

```
int k = 5;
double pi = 3.14159;
System.out.println("L'entier k vaut " + k);
System.out.println("Pi vaut " + pi);
```

### 2.2.4 Lecture des données numériques

La lecture des données numériques se fait à travers le flux d'entré **System.in**. à l'aide de la méthode **read**. Cette méthode pose deux problèmes :

- Elle ne permet de lire qu'un tableau de **byte**. Il faudra procéder par plusieurs étapes en passant par un tableau de caractères déclaré **byte[]**, puis par une chaîne de caractères **String**, puis par l'objet numérique associé (**Double**, **Float**, **Long** ou **Integer**) et enfin par le type de donnée numérique visé (**double**, **float**, **long** ou **int**) :

  Exemple :

```
// Pour lire l'entier k déclaré de type int
byte bArray[] = new byte[80];
System.in.read(bArray);
String strLine = new String(bArray);
```

```
int k = (new Integer(strLine)).intValue();

// Pour lire le nombre dSalaire déclaré de type double
byte bArray[] = new byte[80];
System.in.read(bArray);
String strLine = new String(bArray);
double dSalaire = (new Double(strLine)).doubleValue();
```

- Elle génère une exception de type **`IOException`**. Il faudra donc ajouter l'instruction **`throws IOException`** à la déclaration de la fonction ou est utilisée **`read`**.

### *2.2.5 Les opérateurs*

**Opérateurs arithmétiques**

| | | |
|---|---|---|
| **+** | Addition | **a + b** |
| **-** | Soustraction | **a - b** |
| **-** | Changement de signe | **-a** |
| ***** | Multiplication | **a * b** |
| **/** | Division | **a / b** |
| **%** | Reste de la division entière | **a % b** |
| **&** | ET binaire (l'opération ET est effectuée bit à bit sur tous les bits des opérandes) | **a & b** |
| **\|** | OU binaire (l'opération OU est effectuée bit à bit sur tous les bits des opérandes) | **a \| b** |
| **^** | OU exclusif binaire (l'opération OU est effectuée bit à bit sur tous les bits des opérandes) | **a ^ b** |
| **~** | NON binaire (tous les bits de l'opérande sont modifiés) | **~a** |

Exemples :

```
char  a, b, c;

a = 10;   // a est codé   00001010
b = 12;   // b est codé   00001100

c = a + b;   // c vaut 22 00010110
c = b - a;   // c vaut 2  00000010
c = a - b;   // c vaut -2 11111110
c = -a;   // c vaut -10   11110110
c = a & b;   // c vaut 8  00001000
c = a | b;   // c vaut 14 00001110
c = a ^ b;   // c vaut 6  00000110
c = ~ a;  // c vaut -11   11110101
```

**Opérateurs d'affectation**

| | | |
|---|---|---|
| **=** | Affectation | **a = 5;** |

| | | |
|---|---|---|
| **+=** | Incrémentation<br>Les deux exemples sont équivalents | **a += b;**<br>**a = a + b;** |
| **-=** | Décrémentation<br>Les deux exemples sont équivalents | **a -= b;**<br>**a = a - b;** |

*De façon générale, tous les opérateurs binaires (qui ont deux opérandes) arithmétiques (voir tableau précédent) peuvent être combinés avec l'opérateur d'affectation.*

*Si* ***K*** *est un opérateur arithmétique binaire, les expressions suivantes sont équivalentes :*

**a *K*= b; a = a *K* b;**

### Opérateurs d'incrémentation

| | | |
|---|---|---|
| **++** | Pré-incrémentation (+1) | **++a** |
| **++** | Post-incrémentation (+1) | **a++** |
| **--** | Pré-décrémentation (-1) | **--a** |
| **--** | Post-décrémentation (-1) | **a--** |

Exemples :

```
int a, b;

a = 5;
b = a++;
// à la fin, nous avons a = 6 et b = 5;

a = 5;
b = ++a;
// à la fin, nous avons a = 6 et b = 6;
```

**i++** est incrémenté après affectation, alors **++i** est incrémenté avant affectation.

### Opérateurs booléens

Les opérateurs booléens ne reçoivent que des opérandes de type **boolean**.Ces opérandes peuvent avoir la valeur vraie (**true**)ou faux(**false**).

| | | |
|---|---|---|
| **&** | ET logique<br>Le résultat donne vrai (valeur **true**) si les deux opérandes ont pour valeur vrai<br>Le résultat donne faux (valeur **false**) si l'une des deux opérandes a pour valeur faux. | **a > b & a < c** |
| **&&** | ET logique<br>Cet opérateur fonctionne comme le précédent, à la différence que la deuxième opérande (à droite) n'est pas évaluée (calculée) si la première a pour valeur faux. Car quelque soit la valeur de la deuxième opérande, le résultat est forcément faux. | **a > b && a < c** |
| **\|** | OU logique | **a == b \| a == c** |

Le résultat donne vrai (valeur **`true`**) si l'une des deux opérandes a pour valeur vrai.
Le résultat donne faux (valeur **`false`**) si les deux opérandes ont pour valeur faux.

| | | |
|---|---|---|
| `\|\|` | OU logique<br>Cet opérateur fonctionne comme le précédent, à la différence que la deuxième opérande (à droite) n'est pas évaluée (calculée) si la première a pour valeur vrai. Car quelque soit la valeur de la deuxième opérande, le résultat est forcément vrai. | `a == b \|\| a == c` |
| `!` | NON logique<br>Le résultat est faux si l'expression est vraie et inversement | `!(a <= b)` |

### Opérateurs relationnels

Si les opérandes peuvent être des expressions arithmétiques quelconques, le résultat de ces opérateurs est de type **`boolean`**. Ce résultat peut, bien sur, être utilisé comme opérande des opérateurs booléens.

| | | |
|---|---|---|
| `==` | Egalité | `a == b` |
| `!=` | Différence | `a != b` |
| `<` | Inférieur | `a < b` |
| `>` | Supérieur | `a > b` |
| `<=` | Inférieur ou égal | `a <= b` |
| `>=` | Supérieur ou égal | `a >= b` |

### Autre opérateur

Il existe, en JAVA, un opérateur à trois opérandes :

| | | |
|---|---|---|
| `? :` | Structure alternative.<br>L'exemple sera interprété :<br>*Si **k** est supérieur à **3**, **n** vaudra **5** sinon **6**.* | `n = k>3 ? 5 : 6;` |

## 2.2.6 Les fonctions de calcul mathématiques : la classe `Math`

Les fonctions de calcul mathématiques (racine carrée, sinus, cosinus, etc.) ne sont pas directement implémentées en JAVA. Elles sont néanmoins utilisables à partir de la classe **`Math`**. Cette classe intègre toutes les fonctions membres nécessaires aux applications scientifiques, les fonctions circulaires, hyperboliques, transcendantes, etc. Elle intègre également deux propriétés : **`Pi`** et **`E`**. Ces méthodes et propriétés sont déclarées **`static`**, et sont donc considérées par le compilateur comme des méthodes et des propriétés de classe. On ne peut donc les utiliser que conjointement au nom de la classe :

```
System.out.println("La valeur de E est de " + Math.E);
double rayon = 8.0;
```

```
double aire = rayon * rayon * Math.Pi;
double racineDe2 = Math.sqrt(2.0);
```

Pour plus de précision sur la classe **`Math`**, lire :

- **JAVA (Megapoche)**
  Ch. 4, pages191 à 214

### *2.2.7 Conversion implicite et "Casting"*

Quand deux opérandes de part et d'autre d'un opérateur binaire (qui a deux opérandes) sont de types différents, une conversion implicite est effectuée vers le type "le plus fort" en suivant la relation d'ordre suivante :

**`boolean < byte < char < short < int < long < float < double < String`**

Exemples :

| | |
|---|---|
| `3`(**short**) `* 4`(**short**) | donne `12`(**short**) |
| `3.2`(**float**) `* 4`(**short**) | donne `12.8`(**float**) |

Par ailleurs, un opérateur binaire dont les deux opérandes sont de même type, opère dans ce type. Ce qui semble donner parfois des résultats curieux. Pour avoir le résultat attendu, il faut forcer la conversion par un ***casting*** (changement de type) afin de préciser dans quel référentiel on opère. Un ***casting*** se fait en mentionnant le type entre parenthèse avant l'expression à convertir.

Exemples :

```
int n1 = 5;
int n2 = 2;
double x = n1 / n2;
```

Aussi bizarre que cela paraisse, la valeur de x est de 2.0. En effet, 5 et 2 sont des entiers. Le résultat de la division entière de ces deux nombres est 2 (et il reste 1) et non pas 2.5. Pour obtenir cette dernière valeur, il faut forcer la conversion de l'une des opérandes en un nombre virgule flottante :

```
double x = (double)n1 / n2;
```

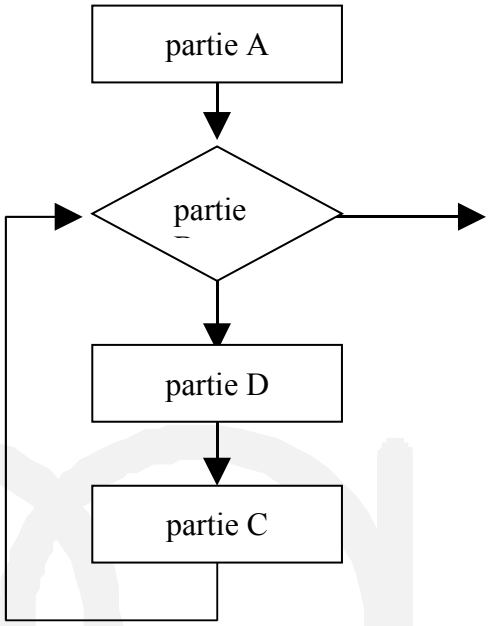
Le fait de faire le *casting* **`(double)`** devant **`5`** force la conversion de 5 (type **int**) et 5.0 (type **double**). L'opérateur / va donc devoir opérer sur un **double** et un **int**. Il y a alors une conversion implicite de 2 (type **int**) en 2.0 (type **double**). L'opérateur peut maintenant opérer sur deux **double** et le résultat (2.5) est un **double**.

Pour plus de précision sur les opérateurs de calcul, lire :

- **JAVA (Le Macmillan)**
  Ch. 10 : Les expressions

### 2.2.8 La structure répétitive FOR

L'instruction **`for`** n'est présentée ici que pour permettre de réaliser le T.P. Elle comprend 4 parties et doit être utilisées de la façon suivante :
**for (**partie A**;** partie B**;** partie C**) {** partie D **}**




et correspond à l'organigramme ci-contre. La partie A permet d'initialiser un compteur appelé *itérateur*. La partie B est la clause de répétition : tant que cette expression booléenne est vrai les partie D puis C sont répétées. La partie C est réservée à l'incrémentation de l'itérateur. Elle est exécutée après la partie D. La partie D ( entre **{** et **}** )peut contenir plusieurs instructions JAVA terminées par un **;**.

Exemples :

```
for (int i = 1; i <= 100; i++)
{
    Intructions;
}
```

Cette structure permet d'exécuter les instructions 100 fois.

## *2.3 TRAVAIL À RÉALISER*

En respectant les conventions d'écriture du langage JAVA, rédiger un programme affichant une table de calcul des nombres entiers à partir de 1, de leur carré et de leur racine carrée. La limite supérieure sera lue au clavier.

```
 1       11.0000000
 2       41.4142136
 3       91.7320508
 4      162.0000000
 5      252.2360680
 6      362.4494897
 7      492.6457513
 8      642.8284271
 9      813.0000000
10     1003.1622777
11     1213.3166248
12     1443.4641016
13     1693.6055513
14     1963.7416574
15     2253.8729833
16     2564.0000000
17     2894.1231056
18     3244.2426407
19     3614.3588989
20     4004.4721360
21     4414.5825757
22     4844.6904158
23     5294.7958315
24     5764.8989795
25     6255.0000000
26     6765.0990195
27     7295.1961524
28     7845.2915026
29     8415.3851648
30     9005.4772256
(...)
```

### *FONCTIONS À UTILISER*

| Fonction | Langage JAVA | Classe | Package |
|---|---|---|---|
| Flux d'affichage à l'écran | `System.out` | `PrintStream` | `java.lang` |
| Flux de lecture au clavier | `System.in` | `InputStream` | `Java.lang` |
| Lecture au clavier | `read` | `InputStream` | `java.lang` |
| Affichage à l'écran | `print` | `PrintStream` | `java.lang` |
| | `println` | " | " |
| Racine carrée | `sqrt` | `Math` | `java.lang` |
| Structure répétitive | `for(...;...;...)` | | |

# T.P. N° 3 - GESTION DES CHAÎNES DE CARACTÈRES STRUCTURATION DES PRGRAMMES

## 3.1 OBJECTIFS

- La déclaration et l'utilisation des variables et des constantes chaînes de caractères.
- L'utilisation des fonctions de manipulation de chaînes de caractères.
- La lecture et l'écriture de celles-ci sur les périphériques standards.
- Les instructions **if**, **swicth**, **for**, **while**, **do**.

## 3.2 CE QU'IL FAUT SAVOIR

### 3.2.1 Les chaînes de caractères

De base, il n'existe pas en JAVA de type chaîne de caractères prédéfini. Le langage JAVA étant un langage évolutif, il existe une classe **`String`** qui implémente celles-ci.

La classe **`String`** est une classe JAVA. Elle permet de gérer des chaînes de caractères sans limite de longueur et possède de nombreuse méthode permettant leur manipulation.

### 3.2.2 Déclaration et instanciation des variables chaîne de caractères

Une variable chaîne de caractères se déclare de la façon suivante :

```
String strNom;
```

La variable **`strNom`** ne *contient* pas la chaîne de caractères. En fait, **`strNom`** est la référence d'une chaîne de caractères. Telle qu'elle est déclarée si dessus, la valeur de **`strNom`** est **`null`**. En effet, il est d'usage, en JAVA, d'initialiser une référence par ce que l'on appelle un instanciation. Cette instanciation peut s'effectuer de plusieurs manières :

- Initialisation d'une chaîne de caractères à une valeur constante :

```
String strNom = "AFPA";
```

- Instanciation d'une chaîne de caractères avec l'instruction **`new`** à partir d'un tableau d'octets lus au clavier par exemple :

```
byte bArray[] = new byte[80];
System.out.read(bArray)
String strNom = new String(bArray);
```

- Instanciation d'une chaîne de caractères avec l'instruction **`new`** par recopie :

```
String strNom = "AFPA";
String strNom2 = new String(strNom);
```

### 3.2.3 Constantes chaînes de caractères

Les constantes chaînes de caractères s'expriment comme une liste de caractères entre doubles-cotes :

```
"AFPA"
```

Entre les doubles-cotes, le caractère \ est utilisé comme joker et interprète le caractère immédiatement après pour coder des caractères non-affichables comme le saut de ligne ou une tabulation :

| | |
|---|---|
| **\n** | Saut de ligne |
| **\r** | Retour de chariot |
| **\t** | Tabulation |
| **\b** | Retour arrière |
| **\v** | Tabulation verticale |
| **\f** | Saut de page |
| **** | Le caractère \ lui-même |
| **\'** | Le caractère ' (cote) lui-même |
| **\"** | Le caractère " (double cote) lui-même |
| **\xHH** | Caractère de code HH en hexadécimal |
| **\NNN** | Caractère de code ASCII NNN en décimal |

### *3.2.4 Affichage des chaînes de caractères*

L'affichage des chaînes de caractères se fait à travers le flux de sortie **System.out**. à l'aide des méthodes **print** ou **println**.

**print** et **println** ne pouvent recevoir qu'un seul paramètre à la fois. Il est possible d'afficher plusieurs données en utilisant l'opérateur de concaténation de chaînes de caractères **+**.

Exemple:

```
String strNom = "AFPA";
System.out.println("Votre nom est " + strNom);
```

### *3.2.5 Lecture des chaînes de caractères*

La lecture des chaînes de caractères se fait à travers le flux d'entré **System.in**. à l'aide de la méthode **read**. Cette méthode pose deux problèmes :

- Elle ne permet de lire qu'un tableau de **byte**. Il faudra procéder en deux étapes en passant par un tableau de caractères déclaré **byte[]**, avant d'obtenir une chaîne de caractères **String** :

```
byte bArray[] = new byte[80];
System.in.read(bArray);
String strLine = new String(bArray);
```

- La chaîne de caractères **strLine** a pour longueur 80 caractères (la taille du tableau) et non pas uniquement les caractères saisis au clavier avant le retour de chariot (qui a été lui-même copié dans le tableau). Si on ne veut, dans **strLine**, que les caractères qui précèdent le retour de chariot, il faut ajouter l'instruction suivante :

```
byte bArray[] = new byte[80];
System.in.read(bArray);
String strLine = new String(bArray);
strLine = strLine.substring(0, strLine.indexOf("\r\n"));
```

- Elle génère une exception de type **IOException**. Il faudra donc ajouter l'instruction **throws IOException** à la déclaration de la fonction ou est utilisée **read**.

### 3.2.6 Affectation de chaînes de caractères

Les variables de type **String** ne contiennent pas la chaîne elle-même. Comme c'est le cas pour toutes les classes JAVA, elle ne contiennent qu'une référence. Dans le cas ci-dessous, **str1** et **str2** contiennent la même valeur, à savoir la référence de la même chaîne **"AFPA"** :

```
String str1 = "AFPA";
String str2 = str1;
```

Si on veut obtenir une véritable affectation, comme on l'aurait avec les types numériques, il faut instancier la variable **str2** par recopie de **str1** :

```
String str1 = "AFPA";
String str2 = new String(str1);
```

Dans le cas ou l'expression à droite de l'affectation est le résultat d'un fonction de manipulation de chaînes de caractères, il n'est pas nécessaire de réinstancier, c'est la fonction de manipulation qui s'en charge. Il en est de même pour l'opérateur de concaténation :

```
String strNom = "Michel ARDANT";
String strPrenom = strNom.substring(0, 5);
String strMsg = "Bonjour " + strPrenom;
```

### 3.2.7 Concaténation de chaînes de caractères

La concaténation de chaînes de caractères s'obtient grâce à l'opérateur **+**. Le résultat de cette opération est une nouvelle chaîne de caractères qui peut être affectée à une variable de type **String** :

```
String strMsg = "Bonjour " + strPrenom;
```

### 3.2.8 Les fonctions de manipulation des chaînes de caractères

La classe **String** possède intrinsèquement les méthodes (fonctions membres) nécessaires à la manipulation des chaînes de caractères. Comme toutes les méthodes définies dans une classe d'objet, elle utilise conjointement à une variable de type **String** avec le caractère point **.** (point).

Toutes les méthodes de la classe **String**, ne seront pas traitées ici. Pour plus de précision lire :

- **JAVA (Megapoche)**
  Ch. 4, page s155 à 190

#### Comparaison de chaînes de caractères

Il est impossible d'utiliser les opérateurs relationnels avec les chaînes de caractères, il faut utiliser la méthode **compareTo** de la classe **String**.

```
if (str1.compareTo(str2))
{
```

```
    //…
}
```

**compareTo** a pour résultat 0 si les deux chaînes sont identiques, une valeur négative, si **str1** est avant **str2**, une valeur positive si **str1** est après **str2**.

#### Longueur d'une chaîne de caractères

La longueur d'une chaîne de caractères est calculée par la méthode **length** de la classe **String**.

```
int nLen = str.length();
```

#### Position d'une chaîne dans une autre

La méthode **indexOf** de la classe **String** permet de calculer la position d'une séquence de caractères dans une chaîne. La valeur 0 de cette position correspond au premier caractère de la chaîne. Il existe plusieurs versions de cette fonction dont les plus utilisées sont les suivantes :

```
int pos1 = str1.indexOf(str2);
int pos2 = str1.indexOf(str2, nPos);
```

Dans le premier cas **pos1** contiendra la position de **str2** dans **str1**. Dans le second cas, la position de **str2** sera cherchée à partir du **nPos**$^{ième}$ caractères de **str1**. ce qui permet de repérer plusieurs occurrences de **str2** dans **str1**.

#### Extraction d'une sous-chaîne

La méthode **substring** de la classe **String** permet d'extraire uns sous-chaîne d'une chaîne de caractères :

```
String strPrenom = strNom.substring(0, 5);
```

### *3.2.9 Conversions entre numériques et chaînes de caractères*

La conversion de tous les types numériques en chaînes de caractères est implicite. Cela permet de passer des paramètres numériques dans des fonctions qui normalement attendent des paramètres de type **String**. C'est le case notamment pour les méthodes **print** ou **println**. **:**

```
int k = 3;
System.out.print("K = ");
System.out.println(k);
```

Il n'en est pas de même dans l'autre sens. Il faut passer par les classes d'objets associées aux types numériques. Par exemple pour les entiers et les nombres en virgule flottante :

```
// Pour les entiers
String str1 = "355";
Integer int1 = new Integer(str1);
int n1 = int1.intValue();
```

```
//Pour les nombres en virgule flottante
String str2 = "64.7658";
Double dou2 = new Double(str2);
double n2 = dou2.doubleValue();
```

- La construction, avec l'instruction **new**, d'un entier de classe **Integer** ou d'un nombre en virgule flottante de classe **Double** peut générer une exception de type **NumberFormatException**. Il faudra donc ajouter l'instruction **throws NumberFormatException** à la déclaration de la fonction ou est utilisée cette construction.

### 3.2.10 Structuration d'un programme en JAVA

Lorsqu'on utilise une méthode de programmation, il est nécessaire de disposer d'instructions gérant les différentes formes de répétitives et d'alternatives afin de structurer un programme. JAVA est un langage structuré, c'est-à-dire qu'il propose au programmeur de telles instructions :

#### Structure répétitive WHILE

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

***condition*** est une expression booléenne (type **boolean**). Les **Instructions** sont exécutées plusieurs fois tant que le résultat de l'expression ***condition*** est vraie (valeur **true**).

Si au départ le résultat de l'expression ***condition*** est faux (valeur **false**), les **Instructions** ne sont jamais exécutées.

#### Structure répétitive DO WHILE

```
do
{
   Instructions;
} while (condition);
```

***condition*** est une expression booléenne (type **boolean**). Les **Instructions** sont exécutées plusieurs fois tant que le résultat de l'expression ***condition*** est vraie (valeur **true**).

Les **Instructions** sont exécutées au moins une fois, même si au départ le résultat de l'expression ***condition*** est faux (valeur **false**).

#### Structure répétitive FOR

```
for (expression_a; expression_b; expression_c)
{
   Instructions;
}
```

La structure ci-dessus est rigoureusement équivalente aux instructions ci-dessous :

```
expression_a;       // expression_a = initialisation des
                    // itérateurs
while (expression_b) // expression_b = condition de répétition
{
    Instructions;
    expression_c;     // expression_c = incrémentation des
itérateurs
}
```

Bien que la structure **for** permette une utilisation relativement large, il est préférable de ne l'utiliser que dans le cas d'une boucle sur un itérateur.

Exemple :

```
for (i = 1; i <= 10; i++)
{
    Instructions(i);
}
```

### Structure alternative IF

```
if (condition)
{
    Instructions_si_vrai;
}
else
{
    Instructions_si_faux;
}
```

***condition*** est une expression booléenne (type **`boolean`**).. Si le résultat évalué est faux (valeur **`false`**), alors **Instructions_si_faux** sont exécutées. Si le résultat évalué de condition est vraie (valeur **`true`**), alors **Instructions_si_vrai** sont exécutées.

### Structure alternative SWITCH

```
switch (expression)
{
case valeur1
    Instructions_Valeur1;
case valeur2
    Instructions_Valeur2;
case valeur3
    Instructions_Valeur3;
default
    Instructions_Defaut;
}
```

Si **expression** est égale à **valeur1** on exécute les instructions **Instructions_Valeur1** puis **Instructions_Valeur2**, **Instructions_Valeur3** jusqu'à **Instructions_Defaut** comprises.

Si **expression** est égale à **valeur2** on exécute les instructions **Instructions_Valeur2** puis **Instructions_Valeur3** jusqu'à **Instructions_Defaut** comprises.

Si **expression** est égale à **valeur3** on exécute les instructions **Instructions_Valeur3** jusqu'à **Instructions_Defaut** comprises.

Si **expression** n'est égale à aucune des valeurs énumérées, on exécute **Instructions_Defaut**.

Pour éviter d'exécuter tous les pavés d'instructions en séquence à partir du moment où l'égalité a été trouvée, il est conseillé de mettre une instruction **`break`** (rupteur) à la fin de chaque pavé.

### Rupteurs

Les rupteurs, en JAVA, sont des instructions de branchement inconditionnel à des points stratégiques du programme.

**`continue`** arrêt de la boucle en cours et débranchement à l'instruction responsable de la boucle.

**`break`** arrêt de la boucle en cours et débranchement derrière l'accolade fermante.

**`return`** dans une fonction retour au programme appelant quel que soit le niveau d'imbrication des structures. La valeur **n** constitue le résultat de la fonction. Si l'instruction **`return`** est trouvée dans la fonction **`main`**, on retourne au système d'exploitation. La valeur **n** constitue alors la valeur de retour du programme (*status*) qui peut être utilisé par le système d'exploitation (**IF ERRORLEVEL** sous DOS).

*Les rupteurs sont parfois dangereux pour une structuration correcte d'un programme. Ils doivent être utilisés à bon escient. Ce sont des GOTO déguisés.*

*Par ailleurs, il existe d'autres rupteurs, tels que* **`try`**, **`catch`**, **`throw`** *et* **`throws`**, *liés à la gestion des exceptions en JAVA et qui feront l'objet d'un T.P. utltérieur.*

## *3.3 TRAVAIL À RÉALISER*

En utilisant la bibliothèque STRING, rédiger un programme capable d'extraire un champ quelconque d'une chaîne de caractères constituée de plusieurs champs séparés par des virgules :

```
AAAAAAAA,BBBBBBB,CCCCCCCCCC,DDDDDDD,EEEEEEEE
    1       2         3        4       5
```

Contrôler votre programme pour les numéros de champs particuliers (premier, dernier). Que se passe-t-il si vous tentez d'extraire un champ dont le numéro est supérieur au nombre de champs contenus dans la chaîne ?

## *FONCTIONS À UTILISER*

| Fonction | Langage JAVA | Classe | Package |
|---|---|---|---|
| Flux d'affichage à l'écran | `System.out` | `PrintStream` | `java.lang` |
| Flux de lecture au clavier | `System.in` | `InputStream` | `Java.lang` |
| Lecture au clavier | `read` | `InputStream` | `java.lang` |
| Affichage à l'écran | `print` | `PrintStream` | `java.lang` |
| | `println` | " | " |
| Longueur d'une chaîne | `length` | `String` | `java.lang` |
| Position d'une chaîne | `indexOf` | `String` | `java.lang` |
| Extraction d'une chaîne | `substring` | `String` | `java.lang` |
| Structure répétitive | `for(...;...;...)` | | |
| Structure alternative | `if (…)` | | |

# T.P. N°4 - LES FONCTIONS

## 4.1 OBJECTIFS

- Structuration d'un programme en fonctions élémentaires.
- Syntaxe et présentation des fonctions.
- Prototype des fonctions.
- Passage des paramètres.

## 4.2 CE QU'IL FAUT SAVOIR

### 4.2.1 Les fonctions

Une fonction est une partie de programme qui comporte un ensemble d'instructions qui a besoin d'être utilisé plusieurs fois dans un programme ou dans différents programmes.

Parce que JAVA est un langage orienté objet, les fonctions ne peuvent être déclarée qu'à l'intérieur d'une classe. Il est donc impossible de déclarer des fonctions isolées.

En JAVA, la plupart des fonctions renvoient un résultat. Ce résultat n'est pas forcément exploité. Certaines fonctions (déclarée **`void`**) ne renvoient pas de résultat.

### 4.2.2 Les bibliothèques de classes

De nombreuses classes (avec leurs fonctions appelées *méthodes* en programmation orientée objet) souvent utilisées sont déjà écrites et livrées avec l'environnement de développement sous la forme de *packages*. Nous avons déjà utilisées plusieurs de celle-ci pour implémenter les données comme les classes **`Boolean`**, **`Integer`**, **`Long`**, **`Double`**, **`String`** (*package* **`java.lang`**), pour les entrées sortie comme les classe **`PrintStream`**, **`InputStream`** (*package* **`java.io`**) et pour le calcul arithmétique ou mathématique comme la classe **`Math`** (*package* **`java.lang`**).

Le langage JAVA est peu implémenté. Ce qui signifie qu'il ne connaît initialement aucune fonction ni aucune classe. Pour pouvoir utiliser les fonctions d'une classe (appelées *méthodes* en programmation orientée objet) en JAVA, il faut qu'elle soit connue par le compilateur. Pour cela il faut la déclarer. Cette opération peut s'appeler importation. Cette importation se fait grâce au mot-clé **`import`**. suivi du nom de la classe et terminé par un point-virgule :

```
import java.io.InputStream;
```

Dans le cas où plusieurs classes du même *package* sont utilisées par un programme, il est possible d'importer toutes les classes du *package* par une seule instruction **`import`** :

```
import java.io.*;
```

Dans la plupart des environnements de développement, il n'est pas nécessaire d'importer les classes du *package* **`java.lang`**. Cette importation est implicite. Cela concerne la classe **`System`** utilisée pour les entrées/sorties standards, la classe **`Math`**

pour le calcul mathématique ou les classes **`Boolean`**, **`Integer`**, **`Long`**, **`Double`** et **`String`** pour les données.

### *4.2.3 Implémentation d'une fonction*

Le développement d'une fonction en instruction s'appelle l'implémentation de la fonction. Cette implémentation ne peut se faire qu'à l'intérieur d'une classe (la classe application par exemple) et doit obéir à des règles de syntaxe qui peuvent se résumer au modèle ci-dessous :

```
class NomDeLaClasse
{
    public static void main (String[] args)
    {
        // Instruction de la fonction pricipale
    }

    TypeResultat nomDeFonction(Type1 par1, Type2 par2)
    {
        // Instructions de la fonction
        // ...
        // Si TypeResulat n'est pas void
        // La fonction renvoie un résultat de ce type
        TypeResultat result = ...;
        return result;
    }

    // Autres fonctions
}
```

Les identifiants de fonctions obéissent aux mêmes règles que les identifiants de variables (voir page 6 du présent support). Par convention, ils commence par une minuscule (JAVA distingue les majuscules des minuscules).

### *4.2.4 Appel à une fonction*

En JAVA, on distingue plusieurs type d'appel à une fonction. Cependant, dans tous les cas, il faut utiliser les parenthèses, même pour les fonctions sans paramètres :

- Si la fonction est l'une des fonction membre de la classe que l'on est en train de développer, par exemple dans la classe de notre application, l'appel se fait en utilisant uniquement le nom de la fonction :

```
public static void main (String[] args)
{
    System.out.println("Taper une chaine de caracteres :");
    String strLine = getString();
}

public static String getString()
{
    //...
}
```

- Si la fonction est une fonction membre d'une classe définissant le type d'une variable, l'appel de la fonction doit être construit avec le nom de la variable et le nom de la fonction séparés par un point. En fait, cela correspond au concept de message de la programmation orientée objet. La fonction est le message envoyé à

la variable. Celle-ci doit répondre en renvoyant un résultat que est le retour de la fonction en question. Ce type d'appel a déjà été utilisé lors de la manipulation des chaîne de caractères. Dans l'exemple ci-dessous, le message **substring** est envoyé à la variable **str1**, message précisant la position du premier et du dernier caractère à extraire. **str1** répond en renvoyant une nouvelle chaîne contenant les caractères demandés qui peut être affectée à une autre variable.

```
String str1 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
String str2 = str1.substring(6,10);
```

- Certaines classes JAVA possèdent intrinsèquement des méthodes, dites *méthodes de classe*, dont l'appel, contrairement au cas précédent, ne nécessite pas la déclaration d'une variable. Ces fonctions particulières sont déclarées par le mot clé **static**. Leur appel doit être construit avec le nom de la classe et le nom de la fonction séparés par un point. C'est le cas des méthodes de la classe **Math**, utilisée pour le calcul mathématique :

```
double nombre = 5
double racine = Math.sqrt(nombre);
```

### 4.2.5 Passage des paramètres

En JAVA, la manière dont sont passés les paramètres dépend de leur nature :

- Si les paramètres sont de type simple (**boolean**, **char**, **byte**, **short**, **int**, **long**, **float** et **double**), les paramètres sont passés par valeur, c'est à dire qu'une copie de ceux-ci est passée à la fonction. Si celle-ci modifie le paramètre , ce dernier n'est pas modifié dans le programme appelant.

```
class JTP
{
    public static void main (String[] args)
    {
        int k = 5;
        fonctionTest(k);
        // k vaut toujours 5
        System.out.println("k = " + k);
    }

    public static void fonctionTest(int n)
    {
        n = 3;
    }
}
```

- Si les paramètres sont du type de l'une des classes JAVA (par exemple **Integer**, **Double**, **String**, etc.), les paramètres sont passés par référence, c'est à dire que la fonction reçoit une référence sur le même objet que le programme appelant. Si la fonction modifie l'objet en question, il est aussi modifié dans le programme appelant. En fait, ceci est très rare car la plupart des fonctions JAVA standards ne modifie pas une instance donnée de l'objet mais en crée une nouvelle à laquelle est affectée une autre valeur. Pour illustrer ce phénomène, voici un petit exercice que l'on peut essayer avec un débogueur :

```
public class Test
{
   public static void main (String[] args)
   {
      String str1 = new String("AAAA");
      StringBuffer sb1 = new StringBuffer(str1);
      fonctionTest(str1, sb1);
      // str1 contient toujours la même valeur
      System.out.println("str1 = " + str1);
      // sb1 a été modifiée par fonctionTest
      System.out.println("sb1 = " + sb1);
   }
   public static void fonctionTest(String s, StringBuffer sb)
   {
      s = "BBBB";
      for (int i = 0; i < 4; i++)
      {
         sb.setCharAt(i, 'B');
      }
   }
}
```

Deux variables `str1` et `sb1` sont crées avec une valeur identique, à savoir **"AAAA"**. **sb1** est de type **StringBuffer**, une autre classe qui permet de modifier une chaîne caractère par caractère, possibilité sans laquelle on ne peut mettre en évidence le phénomène.

La fonction **fonctionTest** reçoit deux paramètres **s** et **sb** à travers lesquels **str1** et **sb1** sont passés par référence.

Dans la fonction **fonctionTest**, **s** est modifié pour prendre la valeur **"BBBB"**. En fait, la variable **s** prend la valeur d'une autre référence sur une autre chaîne de caractères. La référence précédente est perdue.

Par contre, le contenu de la chaîne **sb** est modifiée dans la boucle **for**. **sb** garde la même valeur de référence, mais son contenu a changé et vaut maintenant **"BBBB"**.

A la sortie de la fonction **fonctionTest**, dans la fonction **main**, le contenu de **str1** n'a pas changé, par contre le contenu de **sb1** a été modifié.

### *4.2.6 Récursivité*

Le langage JAVA permet les fonctions récursives. C'est-à-dire que l'une des instructions d'une fonction peut être un appel à la fonction elle-même. C'est très pratique pour coder certains algorithmes comme la factorielle ( factorielle(n) = n * factorielle(n - 1) ) ou l'algorithme d'Euclide ( si n2 > n1, pgcd(n1, n2) = pgcd(n1, n2 - n1) ).

Ce principe est basé sur une notion mathématique : la *récurence* :

Pour démontrer qu'une propriété est vraie quelle que soit la valeur de n, on démontre que :

- La propriété est vraie pour n=1.
- Si la propriété est vraie pour n-1, elle est vraie pour n.

Ainsi, si les deux théorèmes précédents sont démontrés, on saura que la propriété est vraie pour n=1 (1er théorème). Si elle est vraie pour n=1 elle est vraie pour n=2 (2ème théorème). Si elle est vraie pour n=2, elle est vraie pour n=3... Et ainsi de suite.

La création d'une fonction récursive risque d'engendrer un phénomène sans fin. C'est pourquoi, on prévoira toujours une fin de récursivité. Cette fin correspond en fait au codage du premier théorème :

```
public static long factorielle(long n)
{
    if (n == 1)
    {
        return 1;                    //  1er théorème
    }
    else
    {
        return n * factorielle(n - 1);    // 2ème théorème
    }
}
```

### 4.2.7 Surcharge des fonctions

Le langage JAVA permet que deux fonctions différentes aient le même nom à condition que les paramètres de celles-ci soient différents soit dans leur nombre, soit dans leur type. Par exemple, les trois fonctions prototypées ci-dessous sont trois fonctions différentes. Cette propriété du langage C++ s'appelle la *surcharge*.

```
double maFonction(double par1);
double maFonction(double par1, double par2);
double maFonction(int par1);
```

Contrairement aux autres langages de programmation, ce qui permet au compilateur d'identifier et de distinguer les sous-programmes, ce n'est pas le nom seul de la fonction, mais c'est la *signature*. Deux fonctions sont identiques si elles ont la même signature, c'est-à-dire le même **nom**, le même **nombre de paramètres**, dans le même **ordre** et de même **type**.

Il est évident que l'abus de cette possibilité peut amener à écrire des programmes difficiles à maintenir. Cependant, cela permet de donner le même nom à des fonctions qui jouent le même rôle mais dont le nombre et le type de paramètres doivent être différents. C'est le cas, par exemple, de la fonction **`indexOf`** de la classe **`String`** qui peut recevoir 1 ou 2 paramètres. Dans les deux cas, le premier paramètre (unique dans le premier cas) est la portion de chaîne que l'on veut repérer dans la chaîne globale et dans le second cas, le second paramètre permet de désigner une position à partir de laquelle rechercher (voir paragraphe 3.2.8 du présent support).

## *4.3 TRAVAIL À RÉALISER*

Le T.P. n° 4 va être effectué à partir du programme réalisé au T.P. n° 3.

Créer 3 fonctions pour automatiser la lecture de données standards au clavier :

- une fonction **`getString`** pour lire une chaîne de caractères au clavier,
- une fonction **`getInteger`** pour lire un entier au clavier,
- une fonction **`getDouble`** pour lire un nombre virgule flottante au clavier.

Créer deux fonctions supplémentaire pour extraire d'une chaîne de caractères un champ quelconque (l'un des paramètres de la fonction) séparés par un séparateur quelconque (autre paramètre de la fonction). La deuxième version de cette fonction doit permettre au programmeur de ne passer que le N° de champ, la virgule étant considérée comme séparateur par défaut.

### *FONCTIONS À UTILISER*

Mêmes fonctions que le T.P. n° 5.

# T.P. N° 5 - LES EXCEPTIONS

## 5.1 OBJECTIFS

- Repérer les fonctions qui sont susceptibles de générer des exceptions
- Capturer des exceptions pour afficher les messages d'erreur adéquats.
- Générer des exceptions

*La création d'exceptions particulières à une application nécessite une connaissance préalable des concepts de la programmation orientée objet et de leur implémentation en JAVA. Cela sera traité ultérieurement.*

## 5.2 CE QU'IL FAUT SAVOIR

Les exceptions, en JAVA, est un mécanisme qui, pour paraître complexe, est en réalité relativement simple à appréhender. Il permet de traiter les cas d'erreur sans avoir à modifier les algorithmes de base conduisant à la programmation d'une application, c'est à dire sans intégrer des cascades de tests implémentés sous la forme de structures répétitives.

Pour bien comprendre l'avantage à utiliser les exceptions, considérons le T.P. précédent :

L'utilisateur doit saisir un numéro de champ. Cette donnée est en principe numérique. cependant, l'utilisateur de ce programme possède un clavier standard et a donc la possibilité de taper des caractères quelconques. Essayons de taper une valeur non numérique. On assiste à un plantage "sec" du programme.

Il faudrait donc prévoir le cas ou l'utilisateur tape une valeur non numérique. Mais où va-t-on traiter l'erreur, dans la fonction **`main`** ou dans la fonction **`getInteger`** ? Le plus simple serait de la traiter dans la fonction **`main`**. Or ceci est impossible, puisque le plantage a lieu avant le retour de la fonction **`getInteger`**. Il faut donc traiter l'erreur dans cette fonction. Mais comment la signaler au programme appelant ? Quel entier retourner ?

Les exceptions donnent une réponse à tous ces problèmes techniques.

### 5.2.1 Capture d'une exception

Considérons une séquences d'instructions JAVA utilisant des fonctions dont certaines sont susceptibles de générer des exceptions. On *capture* ces exceptions, sans se préoccuper de savoir quelle est la fonction qui l'a générée, en utilisant les mots clés **`try`** et **`catch`**. **`try`** est utilisé pour *surveiller* les instructions en cause et **`catch`** permet de capturer l'exception pour la traiter (en affichant un message d'erreur par exemple).

```
// Séquence d'instructions non surveillées
try
{
   // Séquence d'une ou plusieurs instructions
   // susceptibles de générer une exception
   // qui doivent être surveillées
```

```
}
catch (Exception e)
{
    // Traitement effectué uniquement dans le cas
    // ou une exception a été générée par l'une des
    // instructions surveillées par try
}
// Suite des instructions à exécuter
// Dans ce cas le programme ne plante pas
// et peut continuer en séquence.
```

Dans le cas où aucune exception n'est générée par les instructions contrôlées par **try**, le bloc contrôlé par **catch** est sauté et les instructions suivant **catch** sont exécutées.

Dans le cas ou une instruction contenue dans le bloc contrôlé par **try** génère une exception, les instructions suivantes ne sont pas exécutées. Un saut est effectué sur la première instruction du bloc contrôlé par **catch**. Les instructions de ce bloc sont exécutées.

Dans les deux cas, les instructions qui précèdent et qui suivent l'ensemble des deux blocs **try** et **catch** sont exécutées. Le programme ne plante donc pas. Ce qui assure une grande robustesse aux programmes écrit en JAVA.

### 5.2.2 Affichage d'un message d'erreur

En général, le traitement à effectuer dans le bloc contrôlé par **catch** consiste en l'affichage d'un message d'erreur. La syntaxe de **catch** permet d'initialiser une variable de type **Exception** (**e** dans le cas ci dessous). Cette variable peut être passée en paramètre d'une fonction **print** ou **println** car la classe **Exception** possède la possibilité de se convertir implicitement en **String**, la chaîne de caractères obtenue contenant le message d'erreur associé au type d'exception. Comme ces messages sont en anglais, ils peuvent être remplacés ou accompagné d'un message plus explicite.

```
catch (Exception e)
{
    System.err.println(e);
    System.err.println("Une erreur s'est produite.");
}
```

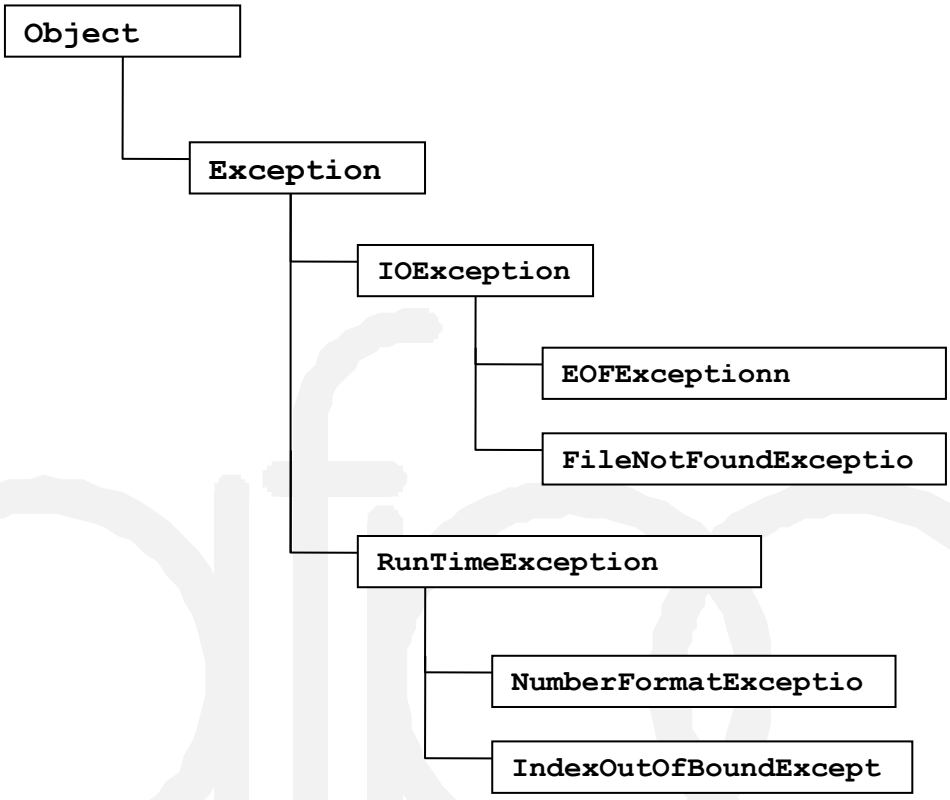
### 5.2.3 Distinction entre plusieurs type d'exception

Les exceptions sont en fait des classes d'objet JAVA organisée de façon hiérarchique

On comprend aisément que l'on ne peut traiter de façon identique une exception de type **EOFException**, générée à la fin d'un fichier lors de la lecture de celui-ci, une exception de type **FileNotFoundException** générée parce que le fichier en question n'a pas été trouvé et une exception de type **NumberFormatException** générée lors de la conversion des caractères lus dans le fichier en donnée de type numérique.

La capture des exceptions, telle qu'elle a été effectué ci-dessus, ne permet pas de distinguer entre toutes ces exceptions. La classe **Exception** étant trop générique. cependant il est possible d'enchaîner les blocs catch chacun correspondant à un type d'exception traité :

```
try
{
   // Séquence d'instructions correspondant à l'ouverture
   // d'un fichier, la lecture de données numériques ou
   // alphanumériques dans celui-ci.
}
catch (NummberFormatException nfe)
{
   System.err.println("Erreur dans le type de données
                       lues.");
}
catch (FileNotFoundException fnfe)
{
   System.err.println("Fichier inexistant.");
}
catch (EOFException eofe)
{
   // Rien a faire ici sinon fermer le fichier.
}
catch (Exception e)
{
   // Ce message est affiché pour toutes les autres
   // exception susceptibles d'être générées.
   System.err.println("Erreur de lecture du fichier);
}
```




Comme le montre le graphe ci-dessus, il y a un ordre pour traiter les exceptions. Il faut commencer par les moins génériques (celle qui se trouve au bout de l'arbre à droite) et finir par les plus générique (en principe **`Exception`** qui est la classe générique de toutes les exceptions.

Il existe une trentaine de classe d'exception en JAVA standard, sans compter les classes que l'on peut être amené à créer soi-même pour affiner le traitement des erreurs. Le question se pose alors de trouver quelles sont les exceptions à traiter effectivement. La documentation des bibliothèques de classes joue ici un rôle essentiel. En effet, cette information y est mentionnée pour toutes les fonctions qui en génèrent.

### *5.2.4 Génération d'une exception*

Dans le traitement d'une fonction, il peut être intéressant de générer une exception. Par exemple, dans le T.P. du chapitre précédent, si l'on considère la fonction qui opère l'extraction d'un champ dans une chaîne de caractères, telle qu'elle a été implémentée, rien ne permet de distinguer le renvoi d'un champ vide (par exemple le 2ème champ de la chaîne **`"AAAA,,BBBB,CCCC"`**) d'une erreur sur le numéro de champ (par exemple 0 ou 5). Dans tous ces cas, la fonction renvoie une chaîne vide. Afin de distinguer ces deux cas, on peut générer une exception lorsque le N° de champ est inférieur à 1 ou supérieur au nombre de champs contenus dans la chaîne.

Pour générer une exception, on utilise le mot-clé **`throw`** :

```
IndexOutOfBoundException ioobe = new IndexOutOfBoundException();
throw ioobe;
```

Le choix de l'exception dépend bien sur du type de problème à traiter. Il existe une trentaine de classes d'exceptions dans l'API JAVA. La classe JAVA **`NumberFormatException`** signale l'utilisation dans une fonction d'un index hors des limites permises. Ce qui correspond au problème de la fonction d'extraction d'un champ.

*Il est possible, en JAVA, de créer ses propres classes d'exceptions. Ce qui nécessite une connaissance préalable des concepts de la programmation orientée objet et de leur implémentation en JAVA. Cela sera traité ultérieurement.*

**`throw`** effectue un branchement direct à un bloc **`catch`** traitant l'exception passée en paramètre. Les instruction suivantes ne seront donc jamais exécutées.

**`try`**, **`catch`**, **`throw`** *sont en fait des rupteurs au même titre que* **`break`**, **`continue`** et **`return`**. *Ce sont des GOTO déguisés dont l'utilisation en dehors des contextes exceptionnels pour lesquels ils ont été créés, peut se révéler dangereuse et nuire à la maintenance des applications JAVA alors qu'ils sont sensés rendre le service inverse.*

### 5.2.4 Différence entre throws et throw

Il existe un différence importante entre l'instruction **`throws`** utilisée déjà lors des premier T.P. et le rupteur **`throw`** :

- **`throw`** génère une exception. Il effectue un branchement inconditionnel au bloc d'instructions contrôlé par **`catch`** associé à la classe d'exception générée.

- **`throws`** est utilisée après la déclaration d'une fonction pour indiqué au compilateur que cette fonction est susceptible de générer une exception et que cette exception doit être traitée par les rupteurs **`try`** et **`catch`** dans le programme qui fait appel à cette fonction.

Le fait d'utiliser **`throws`** dans la fonction **`main`**, lors des T.P. précédents, donnait comme consigne au compilateur de considérer que les exceptions générées devait être traitées par le programme appelant, en l'occurrence la machine virtuelle JAVA. Ce qu'elle faisait consciencieusement en "plantant" notre programme.

En conséquence, toutes les fonctions doivent :

- soit traiter, avec **`try`** et **`catch`**, les exceptions générées par la fonction elle-même (par l'utilisation du rupteur **`throw`**) et par les fonctions qu'elle utilise,

- soit indiquer au compilateur, avec **`throws`**, que ces exceptions devront être traitées par le programme appelant.

## *5.3 TRAVAIL À RÉALISER*

Le T.P. n° 5 va être effectué à partir du programme réalisé au T.P. n° 4.

- Enumérer toutes les fonctions utilisées dans le T.P. précédent (N° 4) en identifiant leur noms, les classes auxquelles elles appartiennent et les exceptions qu'elles génèrent. Cela pourra être présenté sous la forme d'un tableau.

- modifier la fonction **`main`** pour qu'elle gère correctement les exceptions générées. Cette fonction devra contenir un jeux d'essai permettant de tester toutes les fonctions du programmes.

- Eventuellement, modifier les trois fonctions **`getString`**, **`getInteger`** et **`getDouble`** pour quelle gèrent ou transmettent les classes exactes d'exception.

- Modifier les deux fonctions d'extraction de champs pour qu'elles génèrent une exception de classe **`IndexOutOfBoundException`** lorsque le N° de champ demandé n'est pas correct (quand il est inférieur à 1 ou quand il est supérieur au nombre de champs contenus dans la chaîne..

## *CONCLUSION*

Ici se termine la première partie de ce cours. Il est difficile d'aller plus loin dans la programmation en JAVA sans aborder les concepts de la Programmation Orientée Objet. Ces concepts feront l'objet de la deuxième partie de ce cours.

# *ANNEXE A*

## *MOTS CLES DU LANGAGE JAVA*

| | | | | |
|---|---|---|---|---|
| **abstract** | **boolean** | **break** | **byte** | **case** |
| **cast** | **catch** | **char** | **class** | **const** |
| **continue** | **default** | **do** | **double** | **else** |
| **extends** | **final** | **finally** | **float** | **for** |
| **future** | **generic** | **goto** | **if** | **implements** |
| **import** | **inner** | **instanceof** | **int** | **interface** |
| **long** | **native** | **new** | **null** | **operator** |
| **outer** | **package** | **private** | **protected** | **public** |
| **rest** | **return** | **short** | **static** | **super** |
| **switch** | **synchronized** | **this** | **throw** | **throws** |
| **transient** | **try** | **var** | **void** | **volatile** |
| **while** | | | | |

# *INDEX ANALYTIQUE*