Programmation Orienté Objet
- JAVA -

Chapitre 3
Eléments de base du langage JAVA

bekkalimohammed@gmail.com

Plan

I. Variables et Objets
   i. Types primitifs
   ii. Classes et objets
   iii. Tableaux
II. Types Wrappers
III. Classe Object
IV. Gestion des chaines de caractères
V. Quelques propriétés du langage Java
VI. Gestion des exceptions

Variables et Objets

En Java, il existe 3 catégories de types:

- Les types de base (types primitifs)
- Le type Classe
- Le type tableau

Types primitifs

Type	Désignation	Plage de valeurs
byte	Entier signé 1 octet	-128 → 127
short	Entier signé 2 octets	-32768 → 32767
int	Entier signé 4 octets	-2^{31} = -2147483648 → 2^{31}-1 = 2147483647
long	Entier signé 8 octets	-2^{63} = -9223372036854775808 → 2^{63}-1 = 9223372036854775807
float	Réel sur 4 octets	
double	Réel sur 8 octets	-3.4×10^{38} → 3.4×10^{38}
char	Caractère 2 octets	Code en Unicode: `char c = 'x';` ou `char c = '\u0058';`
boolean	Type logique	**true** et **false**

NB:

Le mot **void** n'est plus un type, il est utilisé uniquement pour désigner les procédures

Classes et Objets

- **Création**

La création est objets est toujours réalisée par l'intermédiaire de l'opérateur **new** :

```
NomDeClasse objet;

objet = new NomDeClasse(paramètres d'un constructeur);
```

Ou encore:

```
NomDeClasse objet = new NomDeClasse(paramètres d'un constructeur);
```

Exemple :

```
String  S = new String("ceci est une chaîne de caractères");
```

Classes et Objets

- **Copie**

Un objet peut aussi référencer l'adresse d'un autre objet existant. Les deux objets désigneront la même information. Cependant la destruction d'un objet ne causera pas la destruction de l'autre.

```
NomDeClasse objet1 = new NomDeClasse(paramètres d'un constructeur);
NomDeClasse objet2 = objet1;
```

Exemple :

```
String  S1 = new String("ceci est une chaîne de caractères");
String  S2 = S1;
```

Classes et Objets

- **Destruction**

Un objet en mémoire est détruit automatiquement par l'intermédiaire du Garbage Collector. Aucune destruction explicite n'est alors nécessaire. Cependant peut être affecté à un objet pour supprimer le lien avec l'espace mémoire qu'il référençait avant. Ce qui entraînera la libération de la mémoire si celle-ci n'est pas référencée par un autre objet.

Exemple:

Objet = null;

Les tableaux

- **Déclaration**

Un tableau est objet dynamique déclaré toujours dynamiquement:

Type nomDuTableau[ ]; //type soit primitif ou classe

Exemples :

```
int T[];
String TS[];
```

S'il s'agit d'un tableau de plusieurs dimension on utilise la syntaxe suivante:

Type nomDuTableau[ ][ ]…;

Exemple :

```
int M[][];
```

Les tableaux

- **Création du tableau**

Les tableaux sont, comme les instances de classes, créés par l'intermédiaire de l'opérateur **new** tout en précisant la taille désirée.

```
NomDuTableau = new Type[taille] ;
```

Exemples :

```
T1 = new int[20] ;
T2 = new Vector[10] ;
T3 = new String[15] ;
```

Remarque:

Dans le cas des tableaux de classes, seul le tableau est créé mais pas les éléments ce qui nécessite pour le cas de T2 et T3 la création de chaque T2[i] et chaque T3[i].

Les tableaux

- **Création des éléments d'un tableau**

➢ Pour T1 (tableau de primitifs)

T1[0] = 12; T1[1] = 25; ….

➢ Pour T2 et T3 (tableau d'objets): ***les éléments doivent être créés avant de s'en servir***

T2[0] = new Vector(); T2[0].add(…); …..

T3[0] = new String("….. "); if(T3[0].equals(…)) …..

Les tableaux

- **Taille d'un tableau**

Après la création d'un tableau sa taille ne peut pas être modifiée, sauf si on crée un nouveau tableau et on le référencie par la même variable. Dans tel cas les éléments de l'ancien sont perdus.

Remarque :

La taille d'un tableau peut être déterminé automatiquement par l'intermédiaire de la propriété prédéfinie pour tous les tableaux : **length**

```
int L = T.length;
```

Les tableaux

- **Initialisation d'un tableau**

Un tableau peut être créer un initialisation. Celle-ci se fait de la même manière qu'en C : les éléments du tableau sont fournis entre accolades.

Exemple :

```
// Création d'un tableau à 5 éléments :
        int T[] = { 4, 10, 20, 30, 105 };
// Création d'une matrice à 2 lignes et 3 colonnes :
        int T[][] { {1, 2, 3},
        {4, 5, 6} };
// Création d'un tableau de deux chaînes de caractères :
        String T[] = {new String("abc"), new String("def")};
```

Les tableaux

- **Tableau à plusieurs dimensions**

Type nomDuTableau[ ][ ]…

Exemple: int M[ ][ ]

Création : M = new int[2][3]

Accès : M[i][j] = …

Remarque:

Si on suppose que la 1 ere dimension c'est les lignes et que la 2 eme dimension c'est les colonnes, alors :

M.Length : le nombre de ligne

M[i].length : le nombre de colonne de la ligne i.

Les tableaux

- **Tableau à plusieurs dimensions**

Exemple :

```
int M [ ][ ] = { {1, 2, 4},
                 {2, 8, 95}
};
```

→ Chaque M[i] est un tableau : M[0] → {1, 2, 4}

M[1] → {2, 8, 95}

→ M.length = 2

M[0].length = M[1].length = 3

Les tableaux

- **Tableau à plusieurs dimensions**

Résultat :

Chaque ligne de la matrice peut avoir un nombre d'éléments différents

Exemple:

```
int M [ ][ ] = {
        {1, 2},
        {2, 8, 6, 3},
        {5, 8, 15}
};
```

Les types wrappers

Définition:

Les objet de type wrappers (enveloppeurs : Integer, Long, Float, Double) représentent des objets qui encapsulent une donnée de type primitif et qui fournissent un ensemble de méthodes qui permettent notamment de faire des conversions.

Remarque :

Ces classes ne sont pas interchangeables avec les types primitifs d'origine car il s'agit d'objet

```
Float reel = new Float("3.1415");
System.out.println(5 * reel);      // erreur à la compil
```

La classe Object

C'est la super classe de toutes les classes Java : toutes ces méthodes sont donc héritées par toutes les classes.

Voici la liste des méthodes de la classe Object:

- protected Object clone()
- boolean equals(Object obj)
- protected void finalize()
- Class getClass()

La classe Object

Les méthodes de la classe Object (Suit)

- int hashCode()
- void notify()
- void notifyAll()
- String toString()
- void wait()
- void wait(long timeout)
- void wait(long timeout, int nanos)

La gestion des chaine de caractère

- **La classe String**

Une chaîne de caractères est contenue dans un objet de la classe **String**
On peut initialiser une variable **String** sans appeler explicitement un constructeur : le compilateur se charge de créer un objet.

Exemple : deux déclarations de chaines identiques

String uneChaine = "bonjour";

String uneChaine = new String("bonjour");

L'opérateur + permet de concaténer deux chaines de caractères

On peut tester si une chaine est vide ou non en utilisant isEmpty().

La gestion des chaine de caractère

- **La classe String**

La comparaison de deux chaînes doit se faire via la méthode **equals**() qui compare les objets eux même et non l'opérateur == qui compare les références de ces objets.

Exemple :

```
String nom1 = new String("Bonjour");
String nom2 = new String("Bonjour");
System.out.println(nom1 == nom2);     // affiche false
System.out.println( nom1.equals(nom2));    // affiche true
```

La gestion des chaine de caractère

- **La classe String : méthodes**

Méthodes la classe String	Rôle
charAt(int)	renvoie le nième caractère de la chaine
compareTo(String)	compare la chaîne avec l'argument
concat(String)	ajoute l'argument à la chaîne et renvoie la nouvelle chaîne
endsWith(String)	vérifie si la chaîne se termine par l'argument
equalsIgnoreCase(String)	compare la chaîne sans tenir compte de la casse
indexOf(String)	renvoie la position de début à laquelle l'argument est contenu dans la chaine
lastIndexOf(String)	renvoie la dernière position à laquelle l'argument est contenu dans la chaine
lenght()	renvoie la longueur de la chaine
replace(char,char)	renvoie la chaîne dont les occurrences d'un caractère sont remplacées
startsWith(String int)	Vérifie si la chaîne commence par la sous chaîne
substring(int,int)	renvoie une partie de la chaine
toLowCase()	renvoie la chaîne en minuscule
toUpperCase()	renvoie la chaîne en majuscule
trim()	enlève les caractères non significatifs de la chaine

La gestion des chaine de caractère

- **La classe StringBuffer :**

Les objets de cette classe contiennent des chaines de caractères variables, ce qui permet de les agrandir ou de les réduire.

Cet objet peut être utilisé pour construire ou modifier une chaîne de caractères chaque fois que l'utilisation de la classe String nécessiterait de nombreuses instanciations d'objets temporaires.

La classe StringBuffer dispose de nombreuses méthodes qui permettent de modifier le contenu de la chaîne de caractère.

Quelques propriétés du langage Java

1 - Les tableaux tels qu'ils sont définis en Java (objets dynamiques) ne peuvent pas remplacer la notion de liste. En effet, dès qu'ils sont créés, on ne peut plus leur modifier la taille, sauf si on remplace le tableau créé par un autre.

2 - Pour gérer les listes, on peut:

- Les créer à l'aide de classes.
- Ou encore utiliser une classe Liste existante : **LinkedList**, **Vector** ou **Stack** du package **java.util**.

Quelques propriétés du langage Java

3 - Un tableau de char n'est pas une chaine de caractère en Java. Il ne peut donc pas être géré comme une chaine.

Exemple:

L'écriture : char S[] = "abc" ;

est incorrecte.

4 – Il est possible de convertir un tableau **char** [] en une chaine (**String**) par l'intermédiaire de l'un des constructeurs de la classe **String**.

Exemple:

char S1[] = {'a', 'b', 'c'};

String s2 = new String(S2);

Quelques propriétés du langage Java

5 - Les variables de la classe (ou attributs) peuvent être initialiser lors de leur déclaration (à l'inverse de C++).

```
Class Classe1 {
        private int x = 12, y = 25;
        .....
}
```

6 - Toutes les classes prédéfinies ou définies par l'utilisateur lui même dévirent directement ou indirectement de la classe **Objet** (**java.lang.Object**) . En conséquent les méthodes qui acceptent des paramètres de type Objet peuvent recevoir des arguments d'instance de n'importe quelle classe.

Quelques propriétés du langage Java

7 - Pour déterminer le nom de classe d'un objet, on peut utiliser dans l'une des méthodes (non static) de la classe l'instruction : **getClass().getName()**. La méthode **getClass()** retourne la classe (objet de type **Class**), et la méthode **getName()** (de la classe **Class**) retourne, sous forme de chaine de caractère, le nom de la classe :

8 - Pour déterminer le nom de la classe mère d'un objet, on peut utiliser dans l'une des méthodes non static de la classe l'instruction : **getClass().getSuperclass().getName()**.

La méthode **getSuperclass()** de la classe **Class** retourne un objet de type **Class** qui contient la classe mère de l'objet en cours.

La gestion des exceptions

La gestion des exceptions en Java offre des moyens structurés pour capturer les erreurs d'exécution d'un programme et de fournir des informations significatives à leur sujet. Pour le traitement des exceptions, on utilise les mots clés **try**, **catch** et **finally** :

```
try {
  // Code pouvant se terminer en erreur et déclencher une exception.
}
catch( Exception e ) {
  // Code de traitement de l'exception.
  // La ligne suivante ouvre un suivi de pile de l'exception :
  e.printStackTrace();
}
finally {
      // Le code inséré ici sera toujours exécuté,
      // que l'exception ait été déclenchée dans le bloc try ou non.
}
```

La gestion des exceptions

Le bloc **try** doit être utilisé pour entourer tout code susceptible de déclencher une exception ayant besoin d'être gérée. Si aucune exception n'est déclenchée, tout le code du bloc try est exécuté. Mais, si une exception est déclenchée, le code du bloc try arrête l'exécution à l'endroit où l'exception a été déclenchée et le contrôle passe au bloc catch, dans lequel l'exception est gérée.

La gestion des exceptions

Le bloc **catch** permet de traiter l'exception. On peut faire tout ce dont on a besoin pour gérer l'exception dans une ou plusieurs blocs catch. Le moyen le plus simple de gérer des exceptions peut être réalisé à travers un seul bloc catch. Pour cela, l'argument entre les parenthèses suivant catch doit indiquer la classe Exception, suivie d'un nom de variable à affecter à cette exception. Cela indique que toute exception qui est une instance de java.lang.Exception ou de n'importe laquelle de ses sous-classes sera capturée.

La gestion des exceptions

Le bloc **finally** est optionnel. Le code se trouvant dans le bloc finally sera toujours exécuté, même si le bloc try qu'il déclenche une exception et ne se termine.