La Programmation en Java

RESUME

En programmation, tout ce que nous faisons est un cas particulier de quelque chose de plus général - et souvent nous nous en apercevons trop vite.

Sameh Afi

III. Les tableaux

Un tableau est un ensemble indexé de données d'un même type. L'utilisation d'un tableau se décompose en trois parties :

- Création du tableau ;
- Remplissage du tableau ;
- Lecture du tableau.

1. Création d'un tableau

Un tableau se déclare et s'instancie comme une classe :

```
int monTableau[ ] = new int[10];
```

ou

```
int [ ] monTableau = new int[10];
```

L'opérateur [ ] permet d'indiquer qu'on est en train de déclarer un tableau.

Dans l'instruction précédente, nous déclarons un tableau d'entiers (int, integer) de taille 10, c'est-à-dire que nous pourrons stocker 10 entiers dans ce tableau.

Si [ ] suit le type, toutes les variables déclarées seront des tableaux, alors que si [ ] suit le nom de la variable, seule celle-ci est un tableau :

```
int [] premierTableau, deuxiemeTableau;
float troisiemeTableau[], variable;
```

Dans ces quatre déclarations, seule variable n'est pas un tableau.

2. Remplissage d'un tableau

Une fois le tableau déclaré et instancié, nous pouvons le remplir :

```
int [] monTableau = new int[10];

monTableau[5] = 23;
```

L'indexation démarre à partir de 0, ce qui veut dire que, pour un tableau de N éléments, la numérotation va de 0 à N-1.

Dans l'exemple ci-dessus, la 6ème case contient donc la valeur 23.

Nous pouvons également créer un tableau en énumérant son contenu :

```
int [] monTableau = {5,8,6,0,7};
```

Ce tableau contient 5 éléments.

Lorsque la variable est déjà déclarée, nous pouvons lui assigner d'autres valeurs en utilisant l'opérateur **new** :

```
monTableau = new int[]{11,13,17,19,23,29};
```

3. Lecture d'un tableau

Pour lire ou écrire les valeurs d'un tableau, il faut ajouter l'indice entre crochets ([et]) à la suite du nom du tableau :

```
int [] monTableau = {2,3,5,7,11,23,17};
int nb;

monTableau[5] = 23;  // ->  2 3 5 7 11 23 17
nb = monTableau[4];  // 11
```

L'indice 0 désigne le premier élément du tableau.

L'attribut **length** d'un tableau donne sa longueur (le nombre d'éléments). Donc pour un tableau nommé **monTableau** l'indice du dernier élément est **monTableau.length-1**.

Ceci est particulièrement utile lorsque nous voulons parcourir les éléments d'un tableau.

```
for (int i = 0; i < monTableau.length; i++) {
    int element = monTableau[i];
    // traitement
}
```

4. Tableaux à plusieurs dimensions

En Java, les tableaux à plusieurs dimensions sont en fait des tableaux de tableaux.

Exemple, pour allouer une matrice de 5 lignes de 6 colonnes :

```
int[][] matrice=new int[5][6];
```

On peut également remplir le tableau à la déclaration et laisser le compilateur déterminer les dimensions des tableaux, en imbriquant les accolades :

```
 int[][] matrice =
    {
        { 0, 1, 4, 3 } , // tableau [0] de int
        { 5, 7, 9, 11, 13, 15, 17 } // tableau [1] de int
    };
```

Pour déterminer la longueur des tableaux, on utilise également l'attribut length :

```
matrice.length     //  2
matrice[0].length  //  4
matrice[1].length  //  7
```

On peut facilement parcourir tous les éléments d'un tableau :

```
int i, j;
for(i=0; i<matrice.length; i++) {
        for(j=0; j<matrice[i].length; j++) {
                //Action sur matrice[i][j]
        }
}
```

5. Les tableaux en Java 5+

Java 5 fournit un moyen plus court de parcourir un tableau.

L'exemple suivant réalise le traitement sur monTableau :

```
for (int element : monTableau){
   // traitement
}
```

Attention néanmoins, la variable element contient une copie de monTableau[i]. Avec des tableaux contenant des variables primitives, toute modification de element n'aura aucun effet sur le contenu du tableau.

```
// Vaine tentative de remplir tous les éléments du tableau avec la valeur 10
for(int element : monTableau){
    element=10;
}

// La bonne méthode :
for(int i=0; i < monTableau.length; i++){
    monTableau[i]=10;
}
```

IV. Les classes en Java

La notion de classe constitue le fondement de la programmation orientée objet. Une classe est la déclaration d'un type d'objet.

JSE offre un certain nombre de classes natives dans différents packages :

Version de Java	Année de sortie	Nombre de classes
9	2017	6 005
8	2014	4 240
7	2011	4 024
6	2006	3 793
5	2004	3 279
1.4.2	2002	2 723
1.3.1	2000	1 840

Il faut au moins connaitre les plus basiques, telles que **String** pour les chaines de caractères.

1. Créer une classe

En Java, les classes sont déclarées à l'aide du mot-clef class, suivi du nom de la classe déclarée, suivi du corps de la classe entre accolades. Par convention, un nom de classe commence par une majuscule.

```
public class MaClasse
{
    // corps de la classe
}
```

Le fichier contenant cette déclaration doit avoir pour extension **.java**. Un fichier peut contenir plusieurs déclarations de classes (ce n'est pas recommandé, il faut partir du principe 1 classe = 1 fichier, pour des problèmes évidents de relecture du code, devoir modifier du code où plusieurs classes sont écrites dans un seul fichier est le meilleur moyen de faire n'importe quoi), mais il ne peut contenir qu'au plus une classe dite publique (dont le mot-clef **class** est précédé de **public**). Le fichier doit obligatoirement porter le même nom que cette classe publique : dans l'exemple ci-dessus, il faudrait donc sauver notre classe dans un fichier nommé **MaClasse.java**.

2. Importer une classe

Un fichier .java peut commencer par une ou plusieurs déclarations d'import. Ces imports ne sont pas indispensables, mais autorisent en particulier l'accès aux classes prédéfinies sans avoir à spécifier leur chemin d'accès complet dans les collections de classes prédéfinies (organisées en packages). Dans le code ci-dessous, on souhaite par exemple utiliser la classe prédéfinie Vector (un type de données comparable à des tableaux dont la taille peut varier dynamiquement). Dans la sous-collection de classes prédéfinies "java", cette classe se trouve dans la sous-collection "util" (ou encore : cette classe est dans le package "java.util").

Sans import, il faut spécifier le nom complet de la classe (packages inclus) :

```
public class MaClasse
{
    // ...
    public static void main(String[] args)
    {
        // sans l'import :
        java.util.Vector v = new java.util.Vector();
        // ...
    }
}
```

Avec import, seul le nom de la classe (sans packages) utilisée est nécessaire :

```
import java.util.Vector;

public class MaClasse
{
    // ...
    public static void main(String[] args)
    {
        // avec l'import :
        Vector v = new Vector();
        // ...
    }
}
```

Quand plusieurs classes du même package sont utilisées, l'import peut utiliser le caractère étoile. Une classe peut donc avoir une longue liste d'import :

```
import java.util.Vector;
import java.util.ArrayList;
import java.util.HashMap;

public class MaClasse
{
```

```
    // ...
    public static void main(String[] args)
    {
        Vector     v = new Vector();
        ArrayList a = new ArrayList();
        HashMap    m = new HashMap();
    }
}
```

ou une liste plus courte :

```
import java.util.*;

public class MaClasse
{
    // ...
    public static void main(String[] args)
    {
        Vector     v = new Vector();
        ArrayList a = new ArrayList();
        HashMap    m = new HashMap();
    }
}
```

Enfin, la définition d'une classe peut aussi être précédée d'une (et une seule) déclaration de package, qui indique à quel emplacement se trouve le fichier dans l'arborescence des répertoires d'un projet Java.

```
package org.classes;

public class MaClasse
{
    // contenu de la classe
}
```

3. Instancier une classe

```
MaClasse r; // déclaration d'une référence de type "référence vers MaClasse"

/* instructions diverses */

r = new MaClasse(); // création d'une instance de MaClasse, puis référencement
par r de l'objet crée.
```

On peut déclarer une référence et lui faire immédiatement référencer une nouvelle instance, créée dans cette déclaration :

```
MaClasse r = new MaClasse(); // on crée une instance de MaClasse, que l'on référence par r
```

V. Héritage

L'héritage est la définition d'une classe par extension des caractéristiques d'une autre classe, exemple : on a créé une **classe Vehicule**. Ainsi les classes **Automobile** et **Avion** ont pu hériter des caractéristiques de **Vehicule**.

L'héritage, est l'un des mécanismes les plus puissants de la programmation orientée objet, permet de reprendre des membres d'une classe (appelée **superclasse** ou **classe mère**) dans une autre classe (appelée **sous-classe**, **classe fille** ou encore **classe dérivée**), qui en hérite. De cette façon, on peut par exemple construire une classe par héritage successif.

En Java, ce mécanisme est mis en œuvre au moyen du mot-clé **extends**

Exemple :

```
public class Vehicule
{
    public int vitesse;
    public int nombre_de_places;
}

public class Automobile extends Vehicule
{
    public Automobile()
    {
        this.vitesse = 90;
        this.nombre_de_places = 5;
    }
}
```

VI. Encapsulation

En Java, comme dans beaucoup de langages orientés objet, les classes, les attributs et les méthodes bénéficient de niveaux d'accessibilité, qui indiquent dans quelles circonstances on peut accéder à ces éléments.

Ces niveaux sont au nombre de 4, correspondant à 3 mots-clés utilisés comme modificateurs : private, protected et public. La quatrième possibilité est de ne pas spécifier de modificateur (comportement par défaut).

Comportement par défaut

Si aucun modificateur n'est indiqué, l'élément n'est accessible que depuis les classes faisant partie du même *package*.

Exemple :

```
package com.moimeme.temps;

 class Horloge
 {
     // corps de la classe
 }

 public class Calendrier
 {
     void ajouteJour()
     {
         // corps de la methode
     }

     int mois;

     // suite de la classe
 }
```

La classe *Horloge*, la méthode *ajouteJour* et l'attribut *mois* ne sont accessibles que depuis les classes faisant partie du *package com.moimeme.temps*.

Modificateur "private"

Un attribut ou une méthode déclarée "private" n'est accessible que depuis l'intérieur de la même classe.

Modificateur "protected"

Un attribut ou une méthode déclarée "protected" est accessible uniquement aux classes d'un package et à ses sous-classes même si elles sont définies dans un package différent.

Modificateur "public"

Une classe, un attribut ou une méthode déclarée "public" est visible par toutes les classes et les méthodes.

En résumé

Le tableau résume les différents modes d'accès des membres d'une classe.

Modificateur du membre	**private**	***aucun***	**protected**	**public**
Accès depuis la classe	Oui	Oui	Oui	Oui
Accès depuis une classe du même package	Non	Oui	Oui	Oui
Accès depuis une sous-classe	Non	Non	Oui	Oui
Accès depuis toute autre classe	Non	Non	Non	Oui

VII. Polymorphisme

Le polymorphisme veut dire que le même service, aussi appelé opération ou méthode, peut avoir un comportement différent selon les situations.

Plusieurs signatures pour une même méthode

On peut donner à une même méthode, plusieurs signatures pour implémenter des comportements différents selon les types des paramètres passés. La signature d'une méthode est composée du nom de celle ci, de sa portée, du type de donnée qu'elle renvoie et enfin du nombre et du type de ses paramètres.

```
public class A {

  private int a;

  public A() { //constructeur 1
    System.out.println("Création de A");
  }

  public A(int a) { //constructeur 2 par surcharge
    this.a = a;
    System.out.println("Création de A");
  }

  public int getter() {
    return this.a;
  }

  public void setter(int a) {
    this.a = a;
```

```
  }

  public static void main(String[] args) {
    A premierA = new A(); //construction par 1
    A secondA = new A(1); //construction par 2
  }

}
```

Proposer le passage d'un nombre inconnu de paramètres

Dans la signature d'une méthode , on peut préciser qu'il est possible de passer plus de 1 paramètre du même type en suffixant le type du paramètre avec « ... ».

```
// supposons la méthode suivante :
public String concatenation(String... elements) {
    // pour l'implementation, il faut considérer le paramètre comme un tableau
    String resultat = "";
    for (String element : elements) {
        resultat += element;
    }
    return resultat;
}

// elle peut être appelée ainsi
concatenation("abc", "de", "f"); // renvoie "abcdef"
```

Polymorphisme d'héritage

En redéfinissant une méthode dans une sous-classe, on peut spécialiser le comportement d'une méthode.

```
public class B extends A {

  private int b;

  public B() {
    super();
    System.out.println("Création de B");
  }

  public B(int a, int b){
    super(a);
    this.b = b;
    System.out.println("Création de B");
  }

  public int getter() {
    return this.b;
  }

  public void setter(int a, int b) {
    super.setter(a);
    this.b=b;
```

```
    }

    public static void main(String[] args) {
        A[] table = new A[2];
        table[0] = new A(10);
        table[1] = new B(20,30);

       for(A a : table){
          System.out.println("* " + a.getter());
       }

      /* les résultats sur console:
      Création de A
      Création de A
      Création de B
      * 10
      * 30
      */
    }

}
```