Programmation Objet. Cours 1

Marie-Pierre Béal et Carine Pivoteau
UGE BUT 1

Programmation objet en Java. Records et objets.

Principaux styles de programmation

Style **impératif**

- Exécution d'instructions modifiant l'**état de la mémoire**.
- Très forte utilisation
  - des structures de contrôle (`if`, boucles, ...)
  - et des structures de données.
- Usage intensif de l'itération.
- Langages typiques : Fortran, C, Pascal...
- Exemple : Python tel que vous l'avez découvert au 1er semestre.

Principaux styles de programmation

Style applicatif (fonctionnel)

- **Évaluation d'expressions** :
  - le résultat dépend uniquement de la valeur des arguments
  - ne dépend **pas** de l'état de la mémoire.
- Donne programmes courts, faciles à comprendre.
- Usage intensif de la récursivité.
- Langages typiques : Lisp, Caml, Haskell...

Principaux styles de programmation

Style objet

- Un programme est vu comme
  - une communauté de modules autonomes (objets)
  - disposant de leurs ressources propres
  - et de moyens d'interaction.
- Utilise des *classes* pour décrire les structures et leur comportement.
- Usage intensif des **relations entre les objets**.
- Langages typiques : C++, Java, Ruby, Python, C#...

Principaux styles de programmation

Souvent un même langage permet différents styles de programmation.

Java

1995 : naissance de Java (nom dérivé de Kawa).
2013 : JDK 1.8 (aussi appelée Java 8).
2022, version actuelle : Java 17. (prochaine version : mars 2022).

Caractéristiques du langage :

- orienté objet,
- fortement typé,
- compilé–interprété,
- à partir de la version 1.5, Java offre de la généricité (utilisation de paramètres de types, par ex. `List<String>`),
- à partir de la version 1.8, Java devient fonctionnel : introduction des lambdas (ex : `x -> x + 1`) et des interfaces fonctionnelles,
- à partir de la version 14, arrivée de `records`,
- énorme API : `https://docs.oracle.com/en/java/javase/17/docs/api/`

Hello World

- On écrit le code de la *classe* `Hello` dans un fichier `Hello`**.java** (attention à la majuscule !) :

```java
public class Hello { // doit correspondre au nom du fichier !
  public static void main (String[] args) {
    System.out.println("Bonjour le monde !");
  }
}
```

- On compile (dans un terminal) avec la **commande** `javac`

```
$ javac Hello.java
$ ls
Hello.class Hello.java
```

- On lance l'exécution (l'interprétation du bytecode) par la machine virtuelle Java avec la commande `java`

```
$ java Hello
Bonjour le monde !
```

Autre exemple

- On écrit le code dans le fichier `AdditionTest.java` :

```java
public class AdditionTest {
  public static void main (String[] args) {
    int n = 100;
    int m = 200;
    System.out.println(n + m);
  }
}
```

Important : le code écrit en Java doit toujours se trouver à l'intérieur d'une classe (ou d'un record ou d'une interface).

- On compile avec `javac AdditionTest.java`
- On lance l'exécution avec `java AdditionTest`, on obtient 300.

Expressions et types

Toute expression (portion de code que l'on peut évaluer) a une valeur et un type. Les valeurs peuvent être :

- des valeurs **primitives** (entiers, booléens, caractères, ...)
- des **références (pointeurs) à des objets** (ou des tableaux, mais nous verrons ça plus tard).

Les **types primitifs** sont :

`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`.

Les **objets** peuvent être vus comme des types plus "sophistiqués" déjà présents dans le langage (par ex. `String`) ou créés par le programmeur.

Remarque : les types primitifs commencent tous par des minuscules. Par convention, **les types objets doivent tous commencer par des majuscules !**

Variables

Une *variable* est le nom d'un emplacement mémoire qui peut contenir une valeur. Son *type* décrit la nature des valeurs qu'elle peut contenir.

- Si le type est un type primitif, la valeur est de ce type.
- Si le type correspond à un objet, la valeur est une référence à un objet de ce type (ou d'un sous-type, voir cours suivants).

```
public class Test {
  public static void main(String[] args) {
    int i = 1;
    String word = "mot";
    System.out.println(word + " " + i);
  }
}
```

En Java, un objet ne peut être manipulé que par une référence.

Les objets

Un objet (ou instance)

- est un module autonome,
- qui a ses propres ressources (ses champs),
- et des actions qu'il peut effectuer (ses méthodes).

Pour **utiliser** un objet, il suffit de connaître ses méthodes (par exemple, pour utiliser une `String`, on peut consulter la doc de ses méthodes : `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html`)

Pour **concevoir** un objet, il faut manipuler ses champs.

Records

Depuis Java 14, il est possible de définir simplement ses propres objets comme des *records* (dans la suite du cours, nous verrons que l'ont peut également les définir comme des *classes*).

Record

- Un record est un **tuple nommé non modifiable**.

Autrement dit :

- C'est un *enregistrement* de données.
- Ces données sont stockées dans des **champs** qui portent des noms.
- Les valeurs des champs ne peuvent **pas être modifiées**.

Similaires aux *tuples* de Python et aux *struct* du C.

Définir et créer de nouveaux records

Pour définir un nouveau record, il suffit de

- donner son nom,
- et les données qui le constituent, en indiquant leur type.

Dans le fichier `Point.java`, on écrit la définition du *record* `Point` :

```
public record Point(int x, int y) { }
```

Chaque record est écrit dans son propre fichier `.java` qui porte obligatoirement le même nom que le record. Et ce nom doit commencer par une majuscule !

On peut ensuite créer des points en utilisant le mot-clé `new` et en choisissant les valeurs de `x` et `y` :

```
public class Test {
  public static void main(String[] args) {
    Point point1 = new Point(5, 6);
    Point point2 = new Point(-1, 3);
  }
}
```

Exemple d'utilisation

```java
public class Test {
  public static void main(String[] args) {
    Point point1 = new Point(5, 6);

    int x = point1.x(); // appel de l'accesseur du champs x
    int y = point1.y(); // appel de l'accesseur du champs y

    System.out.println("(" + x + " " + y + ")"); // (5, 6)

    System.out.println(point1.toString()); // Point[x=5, y=6]

    Point point2 = new Point(x, y);

    System.out.println(point2.equals(point1)); // true
    System.out.println(point2.equals(new Point(8, 4))); // false

    System.out.println(point2); // utilisation implicite de toString()
    // Point[x=5, y=6]
  }
}
```

Records : constructeur et méthodes

Pour chaque nouveau *record*, le langage fournit (entre autres) :

- Un **constructeur** canonique permettant de créer des objets de ce type avec `new` et en indiquant les valeurs des champs.
- Les accesseurs pour chaque champs (qui portent le même nom). Chaque accesseur renvoie la valeur du champs du record sur lequel on l'appelle. Attention, cette valeur est typée comme le champs.
- Une méthode `toString` qui permet d'afficher le record. Plus précisément, la méthode renvoie une chaîne de caractères (`String`) qui représente ce record.
- Une méthode `equals` qui permet de tester l'égalité. Elle prend en paramètre un autre record et renvoie le booléen `true` s'il est égal au record sur lequel on l'appelle, et `false` sinon.

On *appelle* une méthode sur un objet avec le point (symbole $\cdot$).

Le test d'égalité

Pour **tester l'égalité des objets, on doit obligatoirement utiliser la méthode** `equals` qui teste l'égalité "structurelle", c'est à dire l'égalité champs par champs.
Si un champs est un objet, l'égalité est testée en utilisant `equals` sur ce champs (sinon, c'est un primitif, on utilise ==).

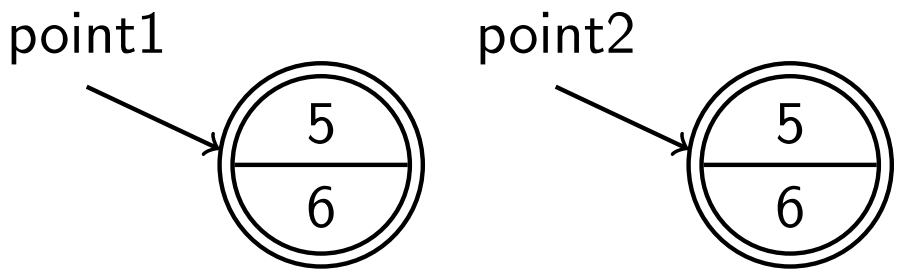
```
public class Test {
  public static void main(String[] args) {
    Point point1 = new Point(5, 6);
    Point point2 = new Point(5, 6);
    Point point3 = new Point(6, 5);
    Point point4 = point1;

    System.out.println(point2.equals(point1)); // true
    System.out.println(point1.equals(point2)); // true, c'est symétrique

    System.out.println(point1.equals(point3)); // false

    System.out.println(point1.equals(point4)); // true
  }
}
```

Les objets et leurs références

Un objet est un module situé dans une zone mémoire.

On le manipule par son adresse appelée **référence** (ou pointeur, mais sans arithmétique).

```
public class Test {
  public static void main(String[] args) {
    Point point1 = new Point(5, 6);
    Point point2 = new Point(5, 6);
  }
}
```

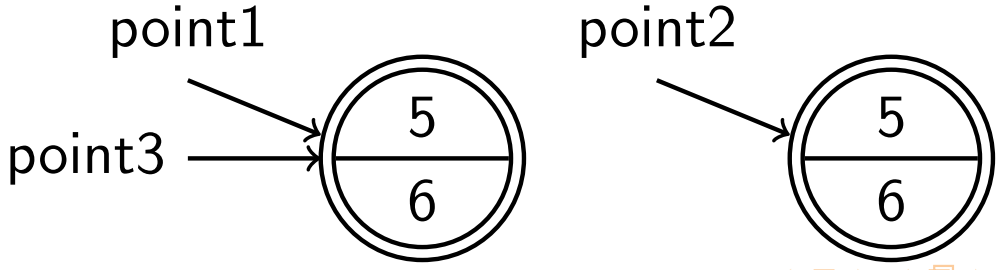


En java, **il existe une référence spéciale** `null`**.** On peut l'utiliser comme "non" valeur de n'importe quelle variable d'un type objet. Dans ce cas, il n'y a pas de zone mémoire réservée.

L'opérateur ==

- Il sert à tester l'égalité des primitifs.
- Sur des objets, l'opérateur == permet uniquement de tester l'égalité des références (c'est à dire, l'endroit ou pointe la flèche).

```java
public class Test {
  public static void main(String[] args) {
    Point point1 = new Point(5, 6);
    Point point2 = new Point(5, 6);
    Point point3 = point1;

    System.out.println(point2.equals(point1)); // true
    System.out.println(point2 == point1); // false
    System.out.println(point3 == point1); // true
  }
}
```



D'autres méthodes dans les records ?

On peut ajouter des méthodes propres au record à l'intérieur de sa définition, pour définir de nouvelles actions.

L'en-tête d'une méthode est toujours de la forme (avec, devant, un éventuel modificateur de visibilité que l'on verra plus tard) :

```
typeDeRetour nomMethode (type1 param1, type2 param2, ...)
```

Par exemple, on peut demander à un `Point` peut s'il est au dessus d'une ligne horizontale :

```
public record Point(int x, int y) {
  public boolean isAbove(int horizontalLine) {
    return y > horizontalLine;
  }
}
```

```
...
    Point point = new Point(-5, 6);
    System.out.println(point.isAbove(4)); // true
    System.out.println(point.isAbove(10)); // false
...
```

Autres exemples

Un point peut renvoyer son opposé :

```
public record Point(int x, int y) {
  public Point opposite() {
    return new Point(-x, -y);
  }
```

ou calculer à quelle distance il se trouve d'un autre point :

```
  public double distance(Point other) {
    int xDiff = x - other.x;
    int yDiff = y - other.y;
    return Math.sqrt(xDiff * xDiff + yDiff * yDiff);
}
```

```
public class Test {
  public static void main(String[] args) {
    Point point = new Point(-5, 6);
    Point point2 = point.opposite();
    System.out.println(point2); // Point[x=5, y=-6]
    System.out.println(point2.distance(point)); // 15.620499351813308
  }
}
```