Programmation orientée objet. Cours 1

Marie-Pierre Béal
UGE BUT 1

Programmation objet en Java. Records et objets.

Bref historique des langages de programmation

Langages

- Dans les années 60 : COBOL, FORTRAN, LISP, ALGOL
- Langages impératifs et structurés (années 70) : C, Pascal
- Langages fonctionnels (années 80 et 90) : ML, OCaml, Haskell
- Langages orientés objets (années 80 et 90) : Smalltalk, C++, Objective C,
- Langages multiparadigmes : Java, Python

Styles de programmation

Style impératif versus fonctionnel

- **Un langage impératif** :
  - exécute des commandes
  - modifie un état (une case mémoire)
- **Un langage fonctionnel** :
  - exécute des fonctions
  - la valeur de retour d'une fonction ne dépend que des valeurs des paramètres

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#...

Le langage Java

Java est un langage

- typé statiquement
- multi-paradigme (impératif, fonctionnel, orienté objet, générique, déclaratif et réflexif)
- avec encapsulation, sous-typage, liaison tardive

Histoire de Java

Versions de Java

- Java 1.0 (1995), Orienté objet
- Java 1.5 (2004), Types paramétrés
- Java 1.8 (2014), Lambda
- Java 17 (2021), Record + Sealed types
- Java 21 (2023) Pattern Matching
- Java 25 (2025) JEP 511 : import de module, JEP 512 : Classe compacte et méthodes main

créé par James Gosling, Guy Steele et Bill Joy à SUN Microsystem en 1995. Il est basé sur C (syntaxe) et Smalltalk (machine virtuelle).

Open source depuis 2006 `http://github.com/openjdk/jdk`

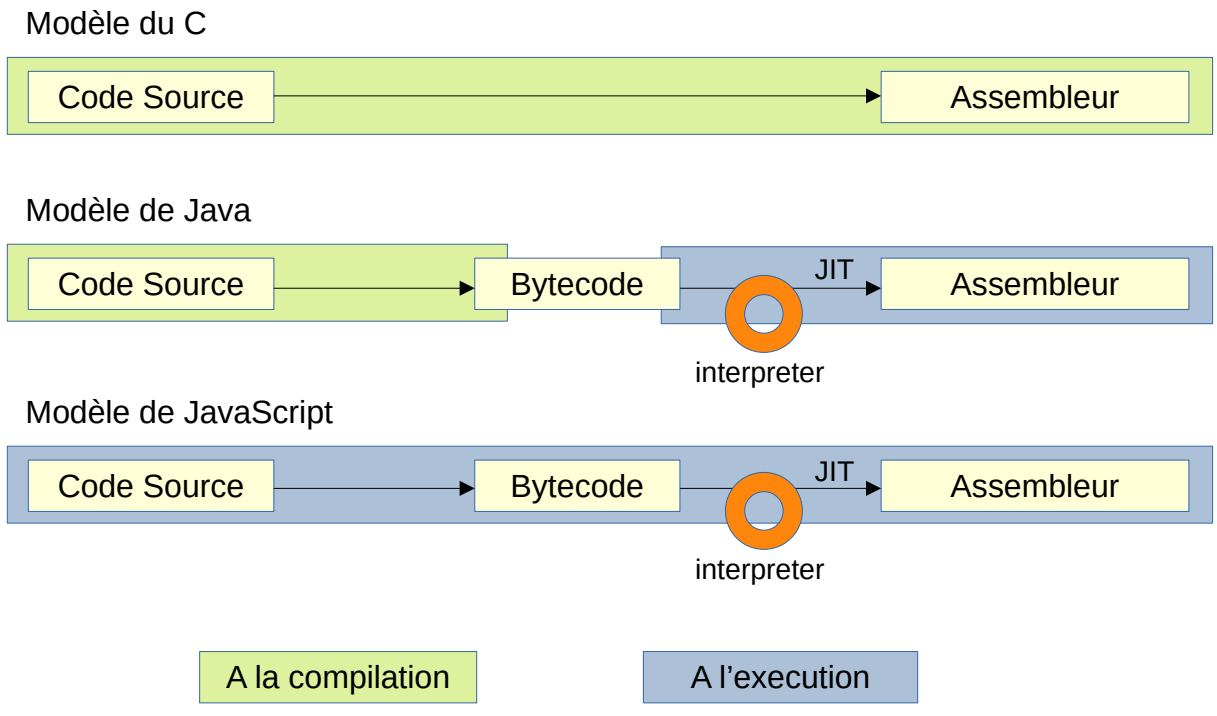
La plateforme Java

Write Once Run Anywhere

Environnement d'exécution

- Machine Virtuelle / Runtime
- Just In Time (JIT) compiler
- Garbage Collectors

Modèle d'exécution



Critiques de Java

Java est trop verbeux. Java est le royaume des noms

- Java préfère un code facile à lire
- Java considère chaque classe comme une librairie (facile à utiliser pour les utilisateurs)

Java est mort

- Java est *backward compatible* donc il évolue doucement

Démarrer en Java

Démarrer en Java

Java est rigide.

Pour permettre une lecture facile

- Les choses sont rangées
  - Le code est rangé dans une fonction appelée méthode
  - Les méthodes sont rangées dans une unité appelée classe
- Conventions de code
  - Un fichier s'écrit en CamelCase (majuscule au début) et finit par le suffixe `.java`
  - Une méthode ou une variable s'écrit en camelCase (minuscule au début)
  - Accolade de début d'une méthode en fin de ligne, accolade de fin alignée avec le début du bloc
  - Si une classe a le nom Foo, alors elle est dans le fichier `Foo.java`

Premier programme

- Dans un fichier `HelloWorld.java` on écrit une classe `HelloWorld`

```
class HelloWorld {
  static void main() {
    IO.println("Hello World!");
  }
}
```

- Le point d'entrée du programme est la méthode main
- `println` est une méthode rangée dans une classe, la classe `java.io.IO`.

Compiler et exécuter

- On lance la compilation seule avec la commande javac suivi du nom du fichier source

  ```
  $ javac HelloWorld.java
  ```

  Un fichier `HelloWorld.class` est créé.

  ```
  $ ls
  HelloWorld.class HelloWorld.java
  $
  ```

- On lance l'exécution (l'interprétation du bytecode) par la machine virtuelle Java avec la commande java suivi du nom du fichier sans le suffixe `.java`.

  ```
  $ java HelloWorld
  Hello World!
  ```

Compiler en mémoire et exécuter

- On peut compiler en mémoire et exécuter en une seule commande avec java suivi du nom du fichier source.

```
$ java HelloWorld.java
Hello World!
```

  Le résultat est sur la sortie standard.

Autre exemple

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

```
class AdditionTest {
  static void main () {
    var n = 100;
    var m = 200;
    IO.println(n + m);
  }
}
```

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

Types et variables

Types

Java a deux sortes de types

- Les types primitifs
  - `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double` (en minuscule)
- Les types objets
  - `String`, `Date`, `Pattern`, `String[]`, etc (en majuscule)

Variables de type

Les types primitifs sont manipulés par leur valeur

```
int i = 3;
int j = i;   // copie 3
```

On peut aussi écrire

```
var i = 3;
var j = i;   // copie 3
```

Les types objets sont manipulés par leur adresse en mémoire (référence)

```
String s = "hello";
String s2 = s;   // copie l'adresse en mémoire
```

Il existe une référence spéciale `null`. On peut l'utiliser comme valeur de n'importe quel type **non primitif**.

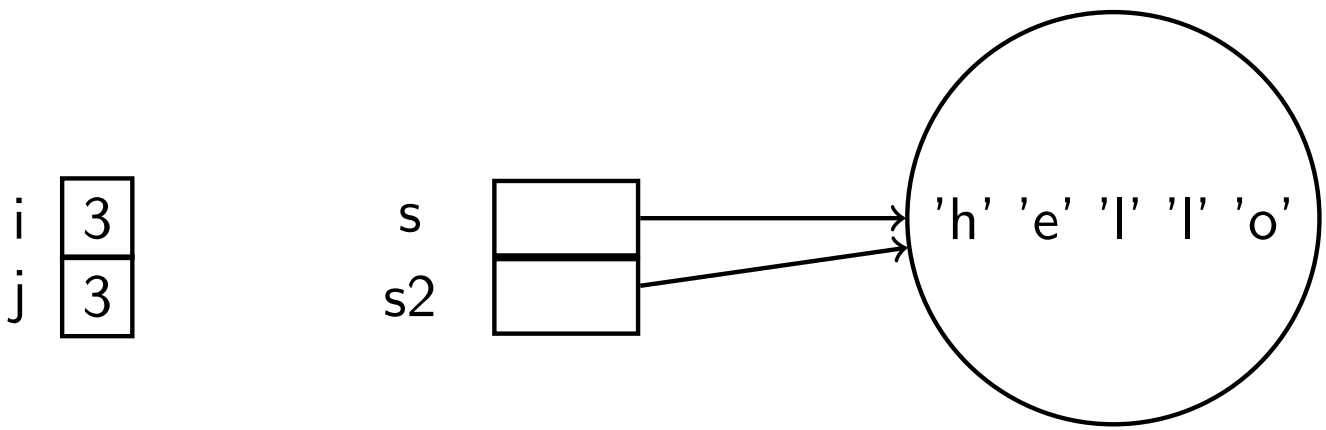
En mémoire

Type primitif

```
int i = 3;
int j = i;   // copie 3
```

Type objet

```
String s = "hello";
String s2 = s;   // copie l'adresse en mémoire
```



Dans le bytecode les variables ne sont pas manipulées par des noms mais par des numéros (0, 1, etc) par ordre d'apparition.

L'opérateur ==

L'opérateur == permet de tester si deux cases mémoire ont la même valeur

```
var i = 3;
var j = 4;
i == j  // renvoie false
i == i  // renvoie true
```

Attention avec les objets, cela teste si c'est la même référence (même adresse en mémoire)

```
var s = ...
var s2 = ...
s == s2  // teste si c'est la même adresse en mémoire,
         // pas le même contenu !
```

Variable locale

Déclaration

```
Type nom;  // information pour le compilateur
           // disparait à  l'exécution
```

```
Type nom = expression;
```

équivaut à

```
Type nom;              // information pour le compilateur
nom = expression;  // assignation à  l'exécution
```

```
var nom = expression;
```

demande au compilateur de calculer (inférer) le type de expression, donc équivalent à

```
Type(expression) nom = expression;
```

Types primitifs et processeur

Les processeurs ont 4 types pour les opérations à l'exécution (donc sur la pile), pas pour le stockage en RAM (sur le tas) : int 32bits, int 64bits, float 32bits et float 64bits. Donc boolean, byte, short, char sont des int 32bits.

Le compilateur interdit les opérations numériques sur les boolean. Pour les autres types, les opérations renvoient un int

```
short s = 3;
short s2 = s + s; // ne compile pas, le résultat est un int
```

Types numériques et processeur

Les types `byte`, `short`, `int`, `long`, `float` et `double` sont signés.

Il n'y a pas d'unsigned à part `char`.

On a des opérations spécifiques pour unsigned

Integer.compareUnsigned(int, int),
Integer.parseUnsignedInt(String),
Integer.toUnsignedString(int),
Byte.toUnsignedInt(byte),
etc

Entiers et processeur

Les `int/long` sont bizarres

Définis entre `Integer.MIN_VALUE` et `Integer.MAX_VALUE` sinon on a un Overflow (passe dans les positifs/négatifs).

```
jshell> Integer.MIN_VALUE
$1 ==> -2147483648
jshell> Integer.MAX_VALUE
$2 ==> 2147483647
```

Donc

- `Integer.MAX_VALUE + 1 == Integer.MIN_VALUE`
- `Integer.MIN_VALUE - 1 == Integer.MAX_VALUE`
- `- Integer.MIN_VALUE == Integer.MIN_VALUE`
- `Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE`
- et 1 / 0 lève une ArithmeticException

Flottants et processeur

Les `float/double` sont bizarres aussi (différemment)

- 0.1 n'est pas représentable donc on a une valeur approchée
- Imprécision dans les calculs $0.1 + 0.2 != 0.3$
- 1. / 0. est `Double.POSITIVE_INFINITY`,
- -1. / 0. est `Double.NEGATIVE_INFINITY`,
- 0. / 0. est `Double.NaN` (Not a Number)
- `Double.NaN` est un nombre (en fait, plusieurs) qui n'est pas égal à lui même
  - `Double.NaN == Double.NaN` renvoie false
  - `Double.isNaN(Double.NaN)` renvoie true
- En java on manipule des flottants avec le type `double`, pas `float`
- On ne manipule jamais des prix avec des flottants

Record

Les objets

Un objet (ou instance)

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

Records

En Java on peut définir des objets avec des records ou des classes.

Un record est un tuple nommé non modifiable.

- 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.

Record

Un record permet de déclarer des tuples nommés

Dans un fichier `Point.java` on met :

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

Dans un fichier `PointTest.java` on met :

```
class PointTest {
  static void main() {
    var point1 = new Point(2, 3);
    var point2 = new Point(1, 4);
    IO.println(point1);
    IO.println(point2);
  }
}
```

On utilise new pour créer une instance (un objet). Il réserve ici un espace mémoire suffisant pour stocker deux entiers (la mémoire est gérée par le garbage collector).

Record

```
$ javac Point.java PointTest.java
$ ls
Point.class  Point.java  PointTest.class  PointTest.java
$ java PointTest
Point[x=2, y=3]
Point[x=1, y=4]
```

Méthodes d'instance

A l'intérieur d'un record, on peut définir des méthodes (fonction rangée dans un record)

```
record Point(int x, int y) {
  double distanceToOrigin() {
    return ...
  }
}
```

```
class PointTest {
  static void main() {
    var point = new Point(2, 3);
    var distance = point.distanceToOrigin();
  }
}
```

Méthodes d'instance

```
record Point(int x, int y) {
  double distanceToOrigin() {
    return Math.sqrt(x * x + y * y);
  }
}
```

```
class PointTest {
  static void main() {
    var point = new Point(2, 3);
    var distance = point.distanceToOrigin();
  }
}
```

Lors de l'appel `point.distanceToOrigin();`, on appelle la méthode `distanceToOrigin()` de Point sur point

```
$ java PointTest.java
3.605551275463989
```

Accesseurs

Dans un record, le compilateur ajoute des méthodes accesseurs automatiquement. Ce sont des méthodes de même noms que les champs.

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

```
class PointTest {
  static void main() {
    var point = new Point(2, 3);
    IO.println(point.x());
  }
}
```

```
$ java PointTest.java
2
```

Méthode equals

Dans un record, le compilateur ajoute une méthode equals qui dit si deux objets ont le même contenu

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

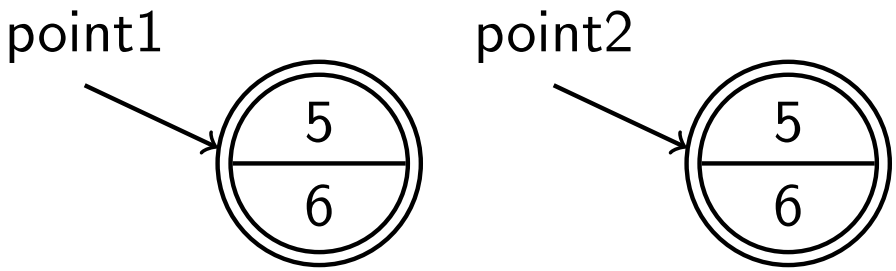
```
class PointTest {
  static void main() {
    var point1 = new Point(2, 3);
    var point2 = new Point(1, 4);
    var point3 = new Point(2, 3);;
    IO.println(point1.equals(point3));
    IO.println(point1.equals(point2));
  }
}
```

```
$ java PointTest.java
true
false
```

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).

```
class Test {
  static void main() {
    var point1 = new Point(5, 6);
    var point2 = new Point(5, 6);
  }
}
```

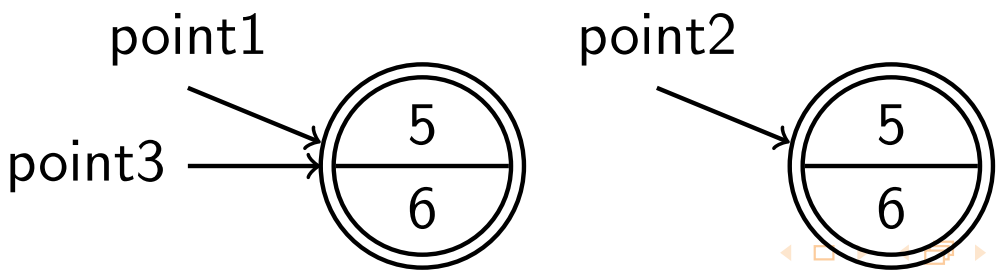


En java, il existe une référence spéciale `null`. On peut l'utiliser comme "non" valeur de n'importe quelle variable d'un type objet. Dans ce cas, il n'y a pas de zone mémoire réservée.

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'adresse de l'objet).

```
class Test {
  static void main() {
    var point1 = new Point(5, 6);
    var point2 = new Point(5, 6);
    var point3 = point1;
    IO.println(point2.equals(point1)); // true
    IO.println(point2 == point1); // false
    IO.println(point3 == point1); // true
  }
}
```



Exercices

Exercice 1 Syntaxe

```
class Mystery {
  static void main() {
    var n = 42; // ou n'importe quelle autre valeur
    if (n <= 1) {
      IO.println(n);
    } else {
      var i = 0;
      var j = 1;
      var k = 1;
      while (k < n) {
        var tmp = i;
        i = j;
        j = i + tmp;
        k++;
      }
      IO.println(j);
    }
  }
}
```

Exercice 1 Syntaxe

1. Quelles sont les différences que vous avez remarquées entre Java et Python dans cet exemple de code ?
2. Comment s'appelle le fichier qui contient ce programme ?
3. Que fait ce programme ?
4. Il existe une boucle `for` en Java dont la forme générale est :

```
for (initialisation; terminaison; increment) {
    instructions;
}
```

L'initialisation est une instruction exécutée une seule fois, avant le début de la boucle. La terminaison est une expression booléenne ; si elle est fausse, la boucle s'arrête. L'incrément est une instruction appelée à la fin de chaque tour de boucle.
Ré-écrivez la boucle while de l'exemple en utilisant une boucle for.

Exercice 2 Premiers records

1. Écrire un record `Bird` pour représenter des oiseaux. Un oiseau a un nom (`name`) et une envergure (`wingspan`) en cm. Quels types peut-on utiliser pour définir ces champs ?
2. Écrire une classe `BirdTest` qui contient une méthode `main` qui crée 2 oiseaux et les affiche.
3. Ajouter de quoi afficher `true` si ces deux oiseaux sont identiques et `false` sinon.
4. Ajouter de quoi afficher `true` si ces deux oiseaux ont la même envergure et `false` sinon.
5. Ajouter une méthode `sameWingSpan` qui permet de tester si deux oiseaux ont la même envergure. Où doit-on l'écrire ? Que doit-elle prendre en paramètre ?
6. Ajouter une méthode `sameName` qui permet de tester si deux oiseaux ont le même nom.

Exercice 3 Boucles

La méthode `Math.random` permet de tirer un nombre aléatoire dans l'intervalle $[0.0, 1.0[$.

1. Écrire un programme qui tire un nombre aléatoire et l'affiche 2 fois de suite.
2. Écrire un programme qui tire un nombre aléatoire et affiche "plus grand" s'il est plus grand que 0.5 et "plus petit" sinon.
3. Écrire un programme qui affiche 100 nombres aléatoires...
   - ... en utilisant une boucle `while` ;
   - ... en utilisant une boucle `for` ;
4. Écrire un programme qui tire 100 nombres aléatoires et affiche le maximum de ces 100 nombres.