L3. Programmation orientée objet. Cours 1

Marie-Pierre Béal (Cours de Rémi Forax)

Démarrer en Java

Bref historique des langages de programmation

Langages

- Dans les années 60 : COBOL, FORTAN, 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

Bref historique des langages 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

Comment on organise et écrit les fonctions ?

Comment on organise les fonctions ?

- classe
- encapsulation
- sous-typage
- liaison tardive

Correspondance des concepts entre impératif et fonctionnel

impératif	**fonctionnel**
Objet + méthode	lambda
Mutable (Collection)	Non mutable (String)
boucle	Stream

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 23 actuel, JEP 477 Implicitly Declared Classes and Instance Main Methods, JEP 476 Module Import Declarations

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

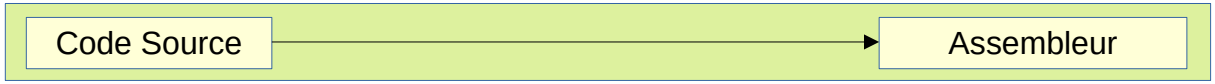
Write Once Run Anywhere

Environnement d'exécution

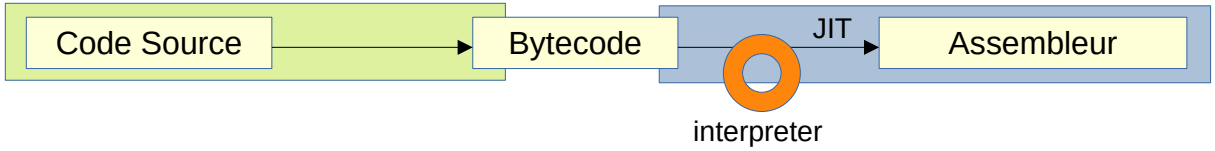
- Machine Virtuelle / Runtime
- Just In Time (JIT) compiler
- Garbage Collectors

Modèle d'exécution

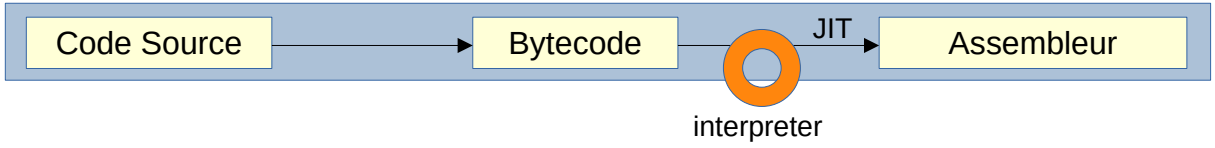
Modèle du C



Modèle de Java



Modèle de JavaScript



A la compilation

A l'execution

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

Enfants de Java

- JavaScript 95
  - Scripting pour navigateur (langage du Web)
- C#, 2001
  - Le Java de Microsoft
- Groovy, 2003
  - Java non typé
- Scala, 2004
  - Fusion programmation orientée objet et programmation fonctionnelle
- Google Go, 2010
  - Le Java de Google, compilation statique
- Kotlin, 2011
  - Scala en plus simple
- Swift, 2014
  - Java de Apple (basé sur Objective C)

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 un nom (cf cours 3), Foo alors elle déclarée est dans le fichier `Foo.java`

Premier programme

- Dans un fichier `HelloWorld.java` on écrit :

```
void main() {
  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`.
  On peut aussi écrire

```
void main() {
  IO.println("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 --enable-preview HelloWorld.java
Hello World!
```

Le résultat est sur la sortie standard.

Compiler et exécuter

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

  ```
  $ javac --release 23 --enable-preview 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  --enable-preview HelloWorld
  Hello World!
  ```

JShell

Pour tester rapidement ou découvrir une API, il y a la fenêtre interactive REPL jshell (avec tab pour compléter)

```
$ jshell --enable-preview
|  Welcome to JShell -- Version 23
|  For an introduction type: /help intro
jshell> var a = 3
a ==> 3
jshell> IO.println(a)
3
jshell> /exit
|  Goodbye
```

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**.

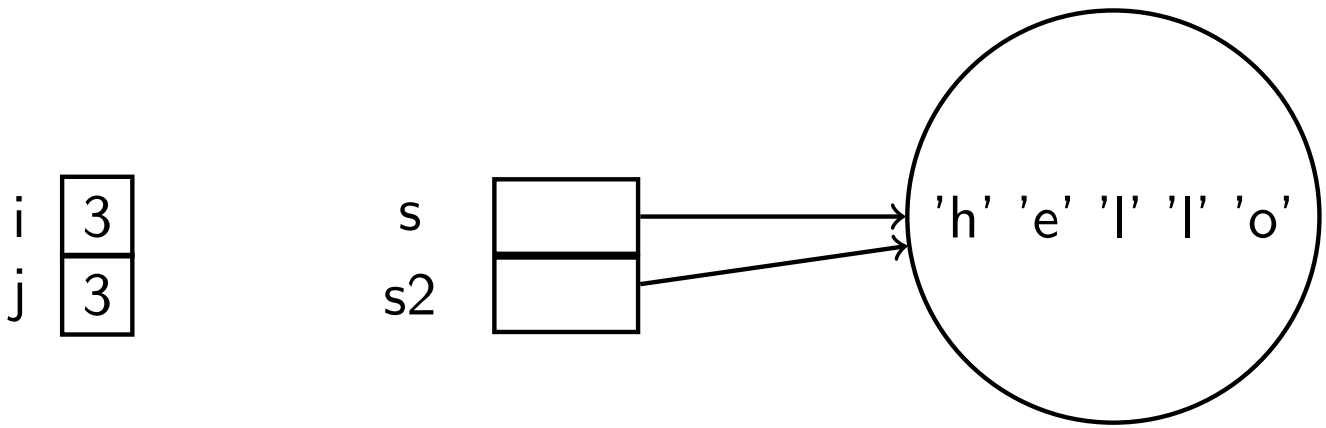
En mémoire

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

Record

Un record permet de déclarer des tuples nommés
Dans un fichier `PointTest.java` on met :

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

void main() {
  var point1 = new Point(2, 3);
  var point2 = new Point(1, 4);
  println(point1);
  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

```
$ java --enable-preview PointTest.java
Point[x=2, y=3]
Point[x=1, y=4]
$ ls
PointTest.java

$ javac --release 23 --enable-preview PointTest.java
$ ls
PointTest$Point.class    PointTest.java     PointTest.class

$ java  --enable-preview 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 ...
  }
}

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

Méthodes d'instance : this explicite

"this" (comme "self" en Python) est un paramètre qui permet d'accéder aux champs de l'objet courant

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

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

"this" correspond à la valeur de l'objet avant le ".". Pour de l'appel `point.distanceToOrigin();`, on appelle la méthode `distanceToOrigin()` de Point avec this = point

Méthodes d'instance : this implicite

Il n'est pas nécessaire de déclarer this, le compilateur l'ajoute automatiquement.

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

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

Note : il est très rare de voir une méthode avec un this explicite au point que certaines personnes ne savent pas que la syntaxe existe.

accès à "this" de façon implicite

Si on accède à une variable qui n'est pas déclarée, le compilateur ajoute "this"

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

```
$ java  --enable-preview 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){}

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

```
$ java --enable-preview PointTest.java
2
```

Méthode equals

Dans un record, le compilateur ajoute une méthode equals qui renvoie un boolean

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

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

```
$ java --enable-preview PointTest.java
true
false
```

Méthodes `hashCode()` et `toString()`

Dans un record, le compilateur ajoute aussi automatiquement des méthodes `hashCode` et `toString`.

- La méthode `toString` renvoie une `String` représentant l'objet.
- La méthode `toString` est la méthode appelée lorsque l'on fait un `println`.
- La méthode `hashCode` renvoie un entier (presque tiré au hasard), utilisé par exemple quand on met on objet dans une collection utilisant du hachage. Deux objets égaux par `equals` ont le même hashCode (voir plus tard).

Méthodes hashCode() et toString()

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

void main() {
  var point1 = new Point(2, 3);
  var point2 = new Point(2, 3);
  var point3 = new Point(4, 5);
  println(point1);
  println(point1.toString());
  println(point1.hashCode());
  println(point2.hashCode());
  println(point3.hashCode());
}
```

```
$ java --enable-preview PointTest.java
Point[x=2, y=3]
Point[x=2, y=3]
65
65
129
```

Les tableaux

Les tableaux

Les tableaux sont des types objets.

- Ils peuvent contenir des objets ou des types primitifs
  - `String[]` ou `int[]`
- On utilise new pour les créer
  - Créer un tableau en fonction d'une taille
    - `String[] array = new String[16];`
    - `var array = new String[16];`
    - initialisé avec la valeur par défaut : false, 0, 0.0 ou null
  - Créer un tableau en fonction de valeurs
    - `var array = new int[] { 2, 46, 78, 34 };`

Les tableaux

Les tableaux ont une taille fixe (pas fixée à la compilation).
Ils ont un champ `length` qui correspond à leur taille

- `array.length` (attention pas `array.length()`)
- Les indices vont de `0` à `array.length - 1`

Les tableaux sont mutables, on utilise `[` et `]`

```
var array = new int[12];
var value = array[4];
array[3] = 56;
```

On ne peut pas sortir des bornes

```
var array = new int[12];
array[25] = ...  // lève ArrayIndexOutOfBoundsException
array[-1] ... // lève ArrayIndexOutOfBoundsException
```

Boucles sur les tableaux

Java possède une façon raccourcie d'écrire une boucle sur un tableau, le `for(:)`

- `for(;;)`

```
var array = new int[12];
for (var i = 0; i < array.length; i++) {
  var element = array[i];
  ...
}
```

- `for(:)`

```
var array = new int[12];
for (var element: array) {
  ...
}
```

`equals/hashCode/toString` pour les tableaux

Ces méthodes existent aussi sur les tableaux mais n'ont pas le comportement attendu

- `equals()` renvoie vrai si c'est la même adresse en mémoire

```
var array1 = new int[] { 1, 2 };
var array2 = new int[] { 1, 2 };
array1.equals(array2);  // false
```

- `hashCode()` renvoie un nombre presque tiré au hasard, le même pour chaque instance
- `toString()` renvoie hashCode + "@" + type du tableau

La classe java.util.Arrays

La classe Arrays (package java.util) fournit des méthodes utilitaires statiques.

Pour afficher un tableau, on utilisera par exemple

```
var numbers = new int[] { 4, 11, 24, 7, 18, 91, 6, 2, 55 };
println(Arrays.toString(numbers));
// [4, 11, 24, 7, 18, 91, 6, 2, 55]
```

```
void main() {
  var array1 = new int[] { 4, 11, 24, 7, 18, 91, 6, 2, 55 };
  var array2 = new int[] { 4, 11, 24, 7, 18, 91, 6, 2, 55 };
  println(Arrays.equals(array1, array2));  // true
  println(Arrays.hashCode(array1));  // -1095323397
  var array3 = new int[] { 4, 11, 24 };
  println(Arrays.hashCode(array3));  // 34000
}
```

Résumé

La méthode qui sert de point d'entrée s'appelle `main`

```
void main(){
  println("Hello");
}
```

Un objet est instancié (créé) en appelant new

```
record Train(int weight){}

void main() {
  var train = new Train(250);
}
```

Résumé

Une méthode rangée dans un record est une méthode d'instance

On appelle une méthode d'instance sur un objet avec le symbole "." après l'objet.

```
record Train(int weight){
  void method(){
    ...
  }
}

void main() {
  var train = new Train(250);
  train.method();
}
```