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 ?

- objet
- 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 actuel, Sequenced Collections
- Futur Java 25 (2025) Pattern Matching

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

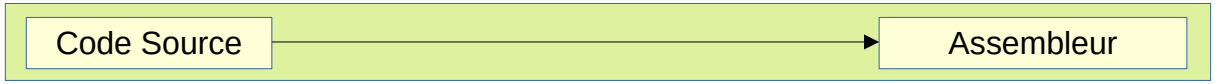
Write Once Run Anywhere

Environnement d'exécution

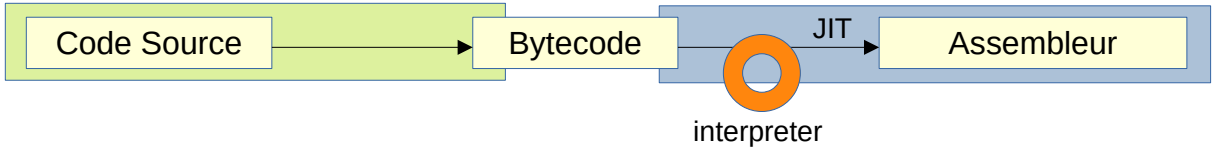
- Machine Virtuelle / Runtime
- Just In Time (JIT) compiler
- Garbage Collectors

Modèle d'exécution

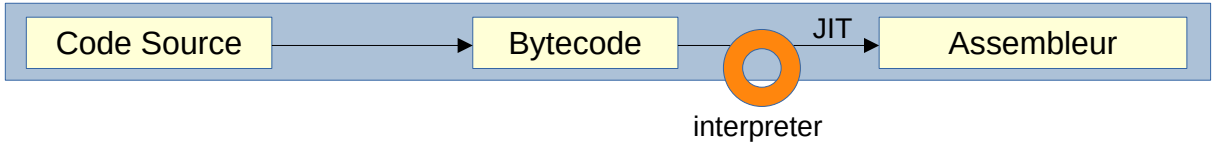
Modèle du C



Modèle de Java



Modèle de JavaScript



A la compilation

A l'execution

Critiques de Java

Java est trop verbeux

- Java 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

Java est le royaume des noms

Enfants de Java

- JavaScript 95
  - Scripting pour navigateur (langage du Web)
- C#, 2001
  - Le Java de Microsoft
- Groovy, 2003
  - Java non typé
- Scala, 2007
  - 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 méthode
  - Les méthodes sont rangées dans une classe
- Conventions de code
  - Une classe s'écrit en CamelCase (majuscule au début)
  - Une méthode ou une variable s'écrit en camelCase (minuscule au début)
  - Une classe Foo est dans le fichier Foo.java
  - Accolade de début de méthode en fin de ligne, accolade de fin alignée avec le début du bloc

Première classe

- Le point d'entrée est la méthode main

```
public class HelloWorld {
  public static void main(String[] args) {
    // mettre du code ici
    System.out.println("Hello World!");
  }
  // pas là
}
// ni là
```

- public ou static sont des modificateurs. Ils sont importants car ils transforment le sens du code. Ils ont un sens différent suivant le contexte. Les deux "public" au dessus ne veulent pas dire la même chose.

Compiler et exécuter

- On lance la compilation qui crée le bytecode avec javac (java compiler) et le nom du fichier source java.

  ```
  $ javac HelloWorld.java
  ```

  Le résultat est un fichier HelloWorld.class.
- On lance l'exécution (l'interprétation du bytecode) par la machine virtuelle Java avec la commande java suivi du nom du fichier sans suffixe.

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

Compiler en mémoire

On peut compiler en mémoire et exécuter en une seule commande (ne marche qu'avec une seule classe)

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

Compiler et exécuter avec les preview features

Dans le fichier HelloWorld2.java on écrit

```
void main() {
  System.out.println("Hello world!");
}
```

La méthode `main` est dans une "unnamed class".

```
$ javac --release 21 --enable-preview HelloWorld2.java
$ java --enable-preview HelloWorld2
Hello world!
```

JShell

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

```
$ jshell
|  Welcome to JShell -- Version 18.0.2.1
|  For an introduction type: /help intro
jshell> var a = 3
a ==> 3
jshell> System.out.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
```

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**.

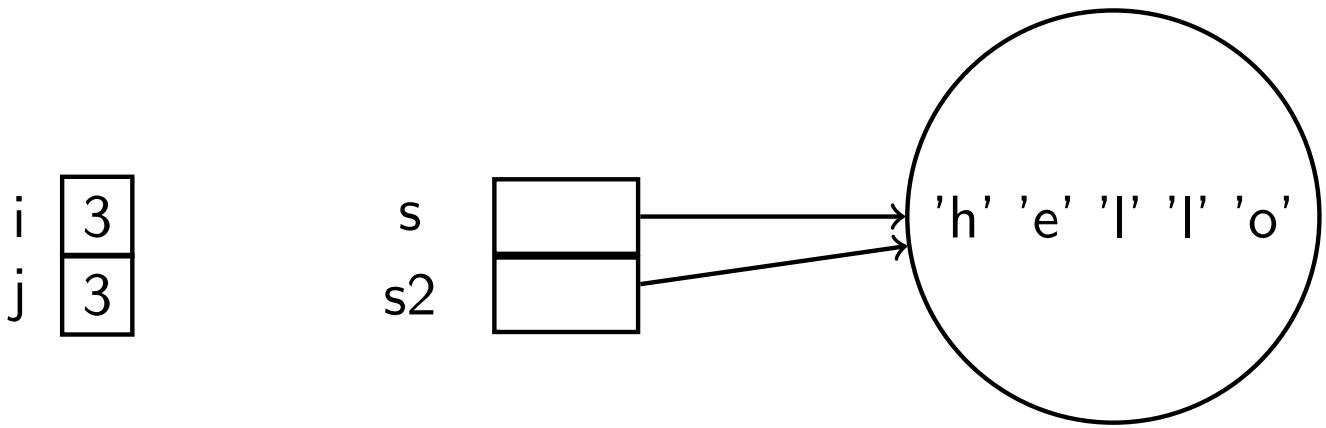
En mémoire

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

```
int i = 3;
int 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)

```
String s ...
String 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 floats/doubles 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

Record

Record

Un record permet de déclarer des tuples nommés

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

On utilise new pour créer une instance (un objet)

```
public class Test {
  public static void main(String[] args) {
    var point = new Point(3, 4);
  }
}
```

réserve un espace mémoire suffisant pour stocker deux entiers (la mémoire est gérée par le garbage collector).

Méthodes d'instance

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

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

Le paramètre `this` est un nom réservé qui correspond à la valeur avant le '.'. Lors de l'appel

```
public class Test {
 public static void main(String[] args) {
    var p = new Point(1, 2);
    var distance = p.distanceToOrigin();
 }
}
```

Méthodes d'instance : this implicite

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

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

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. Il faut mettre self en Python.

Méthodes d'instance et méthodes statiques

On appelle

- une méthode d'instance sur une instance
- une méthode statique sans instance, sur la classe

```
public record Taxi(boolean uber) {
  public String name() {  // this implicite
    return this.uber? "Uber": "Hubert?";
  }
  public static String bar() {  // this n'existe pas ici !
    return "Hello Taxi";
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    System.out.println(new Taxi(true).name()); // Uber
    Taxi.bar();   // Hello Taxi
  }
}
```

Et le main ?

Un record comme une classe peut contenir un main() qui sert de point d'entrée

```
public record Hello(String message) {
  public void print() {
    System.out.println(message);
  }

  public static void main(String[] args) {
    new Hello("Hello Record !").print();
  }
}
```

Les champs

Les champs sont accessibles uniquement entre l'accolade ouvrante et l'accolade fermante du record.

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

Les champs ne sont pas accessibles en dehors du record (accessibles seulement entre les “ et ” du record)

```
public class Test {
  public static void main(String[] args) {
    var point = new Point(3, 4);
    System.out.println(point.x);  // ne compile pas
  }
}
```

Accès implicite à this

Dans une méthode du record, il n'est pas nécessaire de préfixer l'accès à un champ ou une méthode par this ou le nom de la classe

```
public record Point(int x, int y) {
  public double distanceToOrigin() {
    return Math.sqrt(sqr(x) + sqr(y));//pas Point.sqr(this.x) + ..
  }
  private static double sqr(int value) {
    return value * value;
  }
}
```

`private` veut dire non visible de l'extérieur. Les champs d'un record sont private.

Accesseurs

Pour accéder aux valeurs des composants d'un record à l'extérieur, on utilise les accesseurs

```
public class Test {
  public static void main(String[] args) {
    var point = new Point(5, -2);
    System.out.println(point.x());  // 5
    System.out.println(point.y());  // -2
  }
}
```

Un accesseur est une méthode générée par le compilateur pour accéder aux valeurs des champs à l'extérieur

```
public record Point(int x, int y) {
  public int x() {  // il est inutile de l'écrire
    return x;
  }
}
```

Constructeur canonique

Le constructeur est une méthode spéciale appelée lors du new pour initialiser les champs. Le constructeur “canonique” est généré par le compilateur

```
public record Person(String name, int age) {
  public Person(String name, int age) {// généré automatiquement
    this.name = name;
    this.age = age;
  }
}
```

Redéfinir le constructeur

Il est souvent nécessaire de remplacer le constructeur car on veut empêcher de créer des objets avec des valeurs erronées

```
public record Person(String name, int age) {
  public Person(String name, int age) {
    Objects.requireNonNull(name);
                    // plante (lève une NullPointerException)
                    // si name est null
    if (age < 0) {
      throw new IllegalArgumentException("age < 0");
    }
    this.name = name;
    this.age = age;
  }
}
```

On vérifie les pré-conditions.

Redéfinir le constructeur compact

Le constructeur canonique a une version “compacte”

```
public record Person(String name, int age) {
  public Person { // pas de parenthèses
    Objects.requireNonNull(name, "name is null");
    if (age < 0) {
      throw new IllegalArgumentException("age < 0");
    }
  }
}
```

qui ne laisse apparaître que les pré-conditions.
Le compilateur ajoute les `this.name = name;` etc à la fin du constructeur compact.

equals/hashCode/toString

Le compilateur génère aussi les méthodes

- equals() : indique si deux Points ont les mêmes valeurs

```
public class Test {
  public static void main(String[] args) {
    var point = new Point(2, 3);
    var point2 = new Point(2, 3);
    System.out.println(point.equals(point2));  // true
    var point3 = new Point(4, 7);
    System.out.println(point.equals(point3));  // false
    System.out.println(point.hashCode()) // 65
  }
}
```

- hashCode() : renvoie un entier "résumé", cf cours 4

equals/hashCode/toString

- toString() : renvoie une représentation textuelle

```
public class Test {
  public static void main(String[] args) {
    var point = new Point(2, 3);
    System.out.println(point.toString());
                              // Point[x = 2, y = 3]
    System.out.println(point);
                              // Point[x = 2, y = 3]
  }
}
```

Redéfinir les méthodes existantes

On peut changer l'implantation des accesseurs ou des méthodes toString/equals/hashCode

```
public record Pair(String first, String second) {
  @Override
  public String toString() {
    return "Pair(" + first + ", " + second + ")";
  }
}
```

On utilise l'annotation `@Override` pour aider à la lecture, faire la différence entre une nouvelle méthode et le remplacement d'une méthode existante.

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] ;
    - initialisé avec la valeur par défaut : false, 0, 0.0 ou null
  - Créer un tableau en fonction de valeurs
    - int[] 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 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 sur les tableaux mais n'ont pas le comportement attendu

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

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

- hashCode() renvoie un nombre tiré au hazard, 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 };
System.out.println(Arrays.toString(numbers));
// [4, 11, 24, 7, 18, 91, 6, 2, 55]
```

```
import java.util.Arrays;

public class ArraysTest {
  public static void main(String[] args) {
    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 };
    System.out.println(Arrays.equals(array1, array2));  // true
    System.out.println(Arrays.hashCode(array1));  // -1095323397
  }
}
```

Package et import

Package

Une librairie en Java est composée de plusieurs packages.

Pour le JDK (la librarie par défaut de Java)

- java.lang : classes de base du langage
- java.util : classes utilitaires, structures de données
- java.util.regex : expression régulière (cf cours 2)
- java.sql : pour accéder à une BDD
- java.io : pour faire des entrées/sorties
- java.nio.file : entrées/sorties sur les fichiers
- etc.

La directive import en Java

En début de fichier, on spécifie des classes/records appartenant à des packages que l'on veut utiliser

```
import java.util.ArrayList;  // ne pas oublier le ';'
import java.util.Scanner;
```

`import` n'importe pas de fichier au sens de Python/C mais dit que l'on peut utiliser ArrayList à la place de java.util.ArrayList dans le code. Le mot-clef devrait s'appeler "alias" pas "import".

Exemple

```
import java.util.ArrayList;
public class Hello {
  public static void main(String[] args) {
    var list = new ArrayList(); // Le vrai nom de String est
                                // java.lang.String mais
                                // le compilateur "import"/alias
                                // automatiquement les classes
                                // du package java.lang
  }
}
```

est équivalent au code sans import

```
public class Hello {
  public static void main(String[] args) {
    var list = new java.util.ArrayList();
  }
}
```

Résumé

Une classe (ou un record) est une définition à partir de laquelle est créée une instance d'une classe.

```
public record Train(int speed) {}
```

Un objet est instancié en appelant new

```
public class Test {
  public static void main(String[] args) {
      var train = new Train(250);
  }
}
```

c'est conceptuellement équivalent à

```
var train = new Train;
          // alloue la zone mémoire et renvoie l'adresse de début
train.Train(250)
          // appelle le constructeur avec this
          // (ici "train") et les arguments
```

Résumé

On appelle une méthode d'instance sur un objet.

```
train.method();
```

Une méthode statique n'a pas besoin d'instance, on l'appelle sur la classe.

```
Train.aStaticMethod();
```