Hello Java

Rémi Forax

Histoire des Langages

4 premiers (circa 1960)

- COBOL, FORTAN, LISP, ALGOL

Langage Impératif et Structuré (70)

- C, Pascal

Langages Fonctionnels (80 et 90)

- SML, OCaml, Haskell

Langages OO (80 et 90)

- Smalltalk, C++, Java, Objective C

Impératif vs Fonctionnel

Un langage impératif

- Exécute des commandes
- Modifie un état (une case mémoire)

Un langage fonctionnel

- Exécute des fonctions
- *Referential Transparency*

  La valeur de retour d'une fonction ne dépend que des valeurs des paramètres

Comment on organise les fonctions ?

objet

encapsulation

sous-typage

liaison tardive

Comment on écrit les fonctions ?

impératif  fonctionnel

class	record
méthode	lambda
Collection	String
boucle	Stream

Java

Java est un langage

statiquement typé (même si il utilise de l'inférence)

multi-paradigmique

- impératif, fonctionnel, orienté objet, générique, déclaratif et réflexif

moto: encapsulation, sous-typage, liaison tardive

Créé par James Gosling, Guy Steele et Bill Joy
à SUN Microsystem en 1995

Il est basé sur C (syntaxe) et Smalltalk (machine virtuelle)

Histoire abrégée 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
  - Future
    - Java 25 (2025) Pattern Matching

La plateforme Java

Write Once Run Anywhere

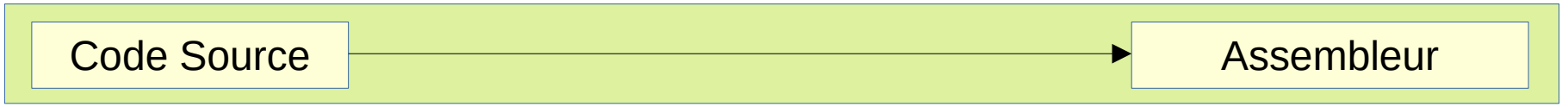
Environnement d’exécution

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

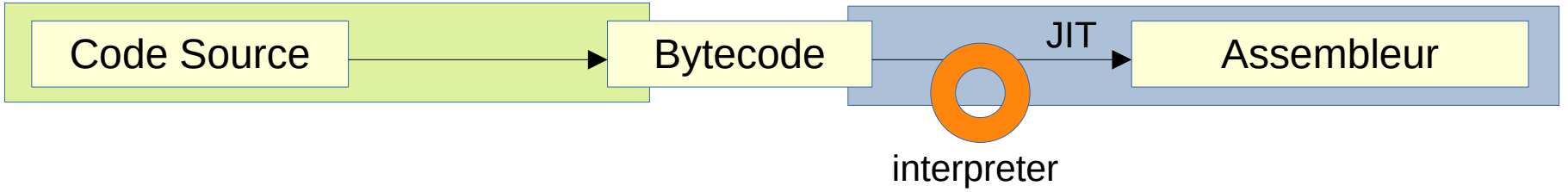
Open Source depuis 2006
http://github.com/openjdk/jdk

Modèle d’exécution

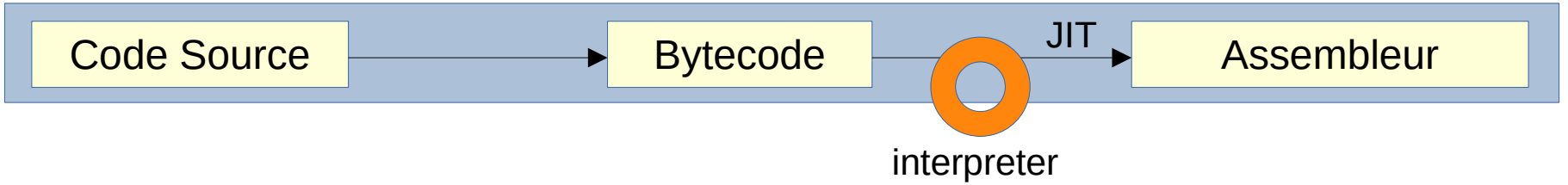
Modèle du C



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 pour les utilisateurs

Java est mort

- Java est *backward compatible* donc il évolue doucement

Java est le royaume des noms (*kingdom of nouns*)

- Java a des lambdas depuis Java 8

Enfants de Java

JavaScript, 1995

- Scripting pour navigateur (langage du Web)

C#, 2001

- Le Java de Microsoft (basé sur COM)

Groovy, 2003

- Java non typé

Scala, 2007

- Fusion POO et Prog. 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

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

Convention 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 block

Ma 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
- Ils ont un sens différent suivant le context (ahhh)

  les 2 "**public**" au dessus ne veulent pas dire la même chose

Compiler et Exécuter

Pour compiler,

javac (java compiler) avec le nom du fichier

javac HelloWorld.java

le résultat est un fichier HelloWorld.class

Pour exécuter,

java avec le nom de la classe

java HelloWorld

Compiler en mémoire / JShell

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

```
java HelloWorld.java
```

Pour tester rapidement ou découvrir une API,
il y a le REPL interactif jshell (avec tab pour compléter)

- jshell> var a = 3
  a ==> 3
- jshell> System.out.println(a)
  3
- jshell> /exit
  | Goodbye

Type et Variable locale

Type

Java a deux sortes de type

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

Variable de Type

Les types primitifs sont manipulés par leur valeur

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

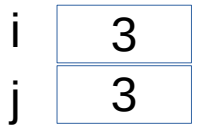
Les types objets sont manipulés par leur adresse en mémoire (référence)

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

En mémoire

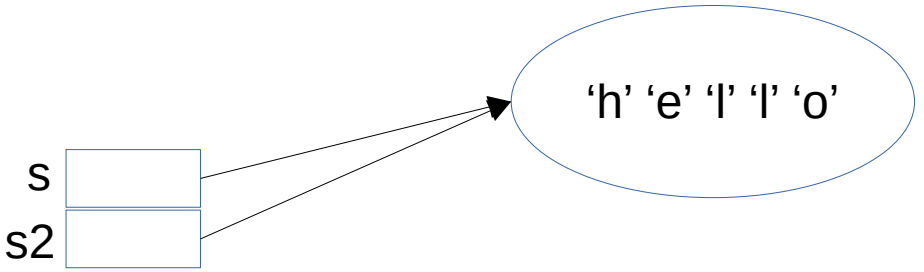
Type primitif

```
int i = 3;
int j = i;
```



Type objet

```
String s = "hello";
String s2 = s;
```



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

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 même adresse en mémoire, pas même contenu !
```

Variable locale

Déclaration

- Type nom; // information pour le compilateur
  // disparait à l'exécution
- Type nom = *expression*;

  équivalent 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*;

Type primitif et processeur

Les processeurs ont 4 types

- int 32bits, int 64bits, float 32bits et float 64bits

donc

- boolean, byte, short, char => int 32bits
  - Le compilateur interdit les opérations numériques sur les booleans
  - Pour les autres types, les opérations renvoient un int

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

Type numérique 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

>>> décale le bit de signe
(>> décale pas le bit de signe)

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

Entier et processeur

Les ints/longs sont bizarres

Définis entre Integer.MIN_VALUE et Integer.MAX_VALUE

sinon Overflow (passe dans les positifs/négatifs)

donc

Integer.MAX_VALUE + 1 == Integer.MIN_VALUE

Integer.MIN_VALUE – 1 == Integer.MAX_VALUE

- Integer.MIN_VALUE == Integer.MIN_VALUE

donc Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE

et

1 / 0, lève une ArithmeticException

Flottant et processeur

Les floats/doubles sont bizarres (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 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)

```
var point = new Point(3, 4);
```

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

Méthode 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

```
var p = new Point(1, 2);
var distance = p.distanceToOrigin();
```

appelle la méthode distanceToOrigin() de Point avec this = p

Méthode 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(~~Point this~~) {
    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 :)

Méthode d’instance ou statique

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(~~Taxi this~~) {  // this implicite
    return this.uber? "Uber": "Hubert?";
  }

  public static String bar() {  // this existe pas ici !
    return "Hello Taxi";
  }
}

…
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();
  }
}
```

Champ

A l'intérieur d'un record, on peut accéder aux champs

```
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)

```
var point = new Point(3, 4);
System.out.println(point.x);  // compile pas
```

Accès implicite à this

Dans une méthode du record, il n'est pas nécessaire de préfixer l'accés 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(Point.sqr(this.x) + sqr(this.y));
  }

  private static double sqr(int value) {
    return value * value;
  }
}
```

private veut dire pas visible de l'extérieur

Accesseurs

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

```
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 int x() {  // 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
    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 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(String name, int age) { // pas de parenthèse
    Objects.requireNonNull(name, "name is null");
    if (age < 0) {
      throw new IllegalArgumentException("age < 0");
    }
    this.name = name;   // ajoutés par le compilateur
    this.age = age;
  }
}
```

qui ne laisse apparaitre que les *pré-conditions*

equals/hashCode/toString

Le compilateur génère aussi les méthodes

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

```
var point = new Point(2, 3);
var point2 = new Point(2, 3);
point.equals(point2)  // true

var point3 = new Point(4, 7);
point.equals(point3)  // false
```

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

```
point.hashCode() // 65
```

- toString() : renvoie une représentation textuelle

```
point.toString()  // Point[x = 2, y = 3]
```

Redefinir 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

ils ont un champ “length” qui correspond à leur taille

```
array.length
```

les tableaux sont mutables, on utilise “[“ et “]”

```
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
```

Boucle sur les tableaux

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

- for(;;)

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

- for(:)

```
var array = …
for(var element: array) {
  ...
}
```

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, 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();
  }
}
```

Enterprise Java Beans

Java Beans

Java possède des frameworks capables

- de déclarer des services Web
  - Spring, JBoss, Quarkus, Micronaut, ...
- de voir une ligne de table de BDD comme un objet
  - Hibernate
- de transformer des objets de JSON / vers JSON
  - Jackson et Gson

  etc

  Toutes ces librairies utilisent la notion de Java Beans

Getters et Setters

Un Java Bean est une classe avec des méthodes spéciales

Elles sont nécessaires pour automatiquement

- stocker les valeurs dans un objet
  - setters: methodes setXX(XX value)
- extraire les valeurs d'un objet
  - getters: methodes getXX() ou isXX()

Les getters et setters **DOIVENT ETRE UTILISÉS UNIQUEMENT** si on utilise ces libraries

En résumé ...

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**

```
var train = new Train(250);
```

c'est conceptuellement équivalent à

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

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();
```