# Concept de prog. en Java

Rémi Forax

## Historique (rapide)

1957-1959: Les 4 fondateurs FORTRAN, LISP, ALGOL, COBOL

1967: Premier langage Objet SIMULA (superset d'ALGOL): Class + héritage + GC

1970-1980: Les langages paradigmiques C, Smalltalk, APL, Prolog, ML, Scheme, SQL

1990-2000: Les langages multi-paradigmes Python, CLOS, Ruby, Java, JavaScript, PHP, ...

## Paradigmes

#### Impératif, structuré (FORTRAN, Algol, C, Pascal)

- séquence d'instructions indiquant comment on obtient un résultat en manipulant la mémoire

#### **Déclaratif** (Prolog, SQL)

- description de ce que l'on a, ce que l'on veut, pas comment on l'obtient

#### Fonctionnel (ou applicatif) (Scheme, Caml, Haskell)

 évaluations d'expressions/fonctions où le résultat ne dépend pas de la mémoire (pas d'effet de bord)

#### Orienté Objet (Modula, Objective-C, Self, C++)

- unités réutilisables, abstraites, contrôle les effets de bord

#### Orienté Data (SML, Caml, Haskell, F#)

- les données sont plus importantes que le code

## Différents styles de prog.

Un style de programmation n'exclue pas forcément les autres

La plupart des languages les plus utilisés actuellement (C++, Python, Ruby, Java, PHP) permettent de mélanger les styles

avec plus ou moins de bonheur :)

#### Java

#### Prog impérative

- Code séquentiel, Boucle, Collection (mutable)

#### Prog fonctionnelle

Record, Lambda, String, Collection (non-mutable)

#### Prog déclarative

- Stream, Regex

#### Prog Orienté Object (API)

- Classe (encapsulation), sous-typage, polymorphisme/liaison tardive

#### Prog Orienté Data

- Record, Sealed interface, Pattern Matching

## La plateforme Java

## Le bytecode est portable

#### Modèle du C **Code Source** Assembleur Modèle de Java JIT Bytecode **Code Source** Assembleur interpreter Modèle de JavaScript JIT Bytecode Assembleur **Code Source** interpreter A la compilation A l'execution

#### OpenJDK vs Java

OpenJDK contient les sources

http://github.com/openjdk/jdk

Java est un ensemble de distributions

Oracle Java, RedHat OpenJDK, Amazon Coretto, Azul Zulu, SAP SapMachine, ...

Une distribution doit passer le *Test*Compatibility Kit (TCK) pour s'appeler Java

#### Java

#### Java est deux choses

- Le langage: Java Language Specification (JLS)
- La platforme: Java Virtual Machine Specification (JVMS)
   https://docs.oracle.com/javase/specs/

Java est <u>pas</u> le seul langage qui fonctionne sur la plateforme Java

 Groovy, Scala, Clojure ou Kotlin fonctionne sur la plateforme Java

#### Le langage Java

Le langage et la machine virtuelle n'ont pas les mêmes features

La VM ne connait pas

- Les exceptions checkées
- Les varargs
- Les types paramétrées
- Les classes internes, les records, les enums

. . .

Le compilateur a un système de types plus riche que ce que comprend la machine virtuelle

## Evolution du language Java

Les languages de programmation évoluent ... car les besoins des logiciels évoluent

#### Java

- OOP (Java 1.0 1995)
- Type paramétré + Enum (Java 5 2004)
- Lambda + Stream (Java 8 2014)
- Record + Pattern matching (Java 21 2023)

## Le monde selon Java (Edition 2023)

## Object Oriented Programming

#### En Java

- Classe (encapsulation)
   Séparation de l'API publique et de l'implantation interne
- Interface (sous-typage)
   Pour considérer des classes différentes comme implantant un même type
- Liason tardive (à l'exécution)
   o.toString() appel la bonne implantation

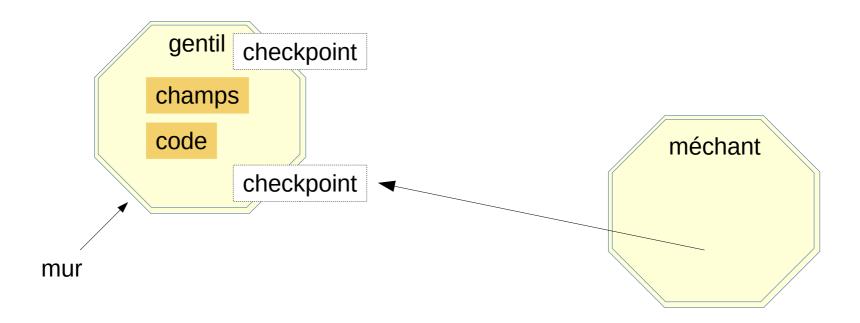
## Encapsulation

#### Le monde selon Trump

#### Une classe considère

Son intérieur (les champs, le code des méthodes) comme étant de confiance

... et les autres classes comme méchantes Les méthodes publiques fournissent un d'accès restreint (checkpoint)



#### API publique vs Implantation

(information hiding)

Une classe encapsule ses données

L'API publique est l'ensemble des méthodes visibles par un utilisateur

donc les méthodes publiques (et protected)

L'implantation, c'est comment l'API public est codée champs privée, méthodes privées, classes internes privées

Le fait de protéger les données est appelée encapsulation

#### Encapsulation

```
class Person {
    private String name;
    private List<String> pets;

    public Person(String name, List<String> pets) {
        this.name = name;
        this.pets = pets; // oops mistake
    }
}
```

var person = new Person("John", List.of("Garfield"));

## Violation de l'encapsulation

Une violation classique de l'encapsulation consiste à utiliser des objets mutables à la construction sans faire de copie défensive

```
class Person {
    private String name;
    private List<String> pets;

    public Person(String name, List<String> pets) {
        this.name = name;
        this.pets = pets; // oops mistake
    }
}

var pets = new ArrayList<String>(); // mutable list
    var person = new Person("John", pets);
    pets.add("Garfield"); // violation !
```

#### Classe Non Mutable

Avoir des champs non modifiables rend l'encapsulation plus facile à implanter + copie défensive

```
class Person {
    private final String name;
    private final List<String> pets;

public Person(String name, List<String> pets) {
    this.name = name;
    this.pets = List.copyOf(pets); // defensive copy
    }
}

var person = new Person("John", List.of("Garfield"));
person.pets().add("Garfield"); // exception !
```

## Programmation par contrat

## Programmation par contrat

"L'état d'un objet doit toujours être valide"

Inventé par Bertrant Meyer (Eiffel 1988)

Extension de la notion d'encapsulation

Utiliser par le JDK

- Précondition + vérification des invariants

Blow early, blow often Josh Blosch

## Exemple de prog par contrat

```
class Book {
 public Book(String author, long price) {
  Objects.requireNonNull(author);
  if (price < 0) { throw new IAE(...); }
 public long priceWithDiscount(long discount) {
  if (discount < 0) { throw new IAE(...); }</pre>
  if (discount > price) { throw new ISE(...); }
  return price - discount;
var book = new Book("Dan Brown", 40);
book.priceWithDiscount(50); // exception
```

## Exceptions + assert

Pré-conditions sur les arguments

NullPointerException (**NPE**), IllegalArgumentException (**IAE**) si l'argument n'est pas valide

IllegalStateException (**ISE**), si l'état de l'objet ne permet pas l'exécution la méthode

... sont documenté dans la javadoc!!

Les invariants

```
stack.push(element);
assert stack.size() != 0;
```

... sont documentés dans le code

## Type (compilateur) en Java

#### Typage

Java est un language typé (variables & champs)
le type est déclaré explicitement
Point point = ...
ou implicitement
var point = new Point(...);

Les records, classes, interfaces sont aussi des types

```
Point point = new Point(...);
Type (record, class, interface) record ou classe
```

### Type vs Classe

#### Type (pour le compilateur)

- ensemble des opérations que l'on peux appeler sur une variable (méthodes applicables)
- existe pas forcément à l'exécution

#### Classe (pour l'environnement d'exécution)

- taille sur le tas (pour new)
  - Les champs sont des offsets par rapport à une addresse de base
- codes des méthodes
  - Les méthodes sont des pointeurs de fonctions dans la *vtable*

## Type primitif vs Object

On Java, les types primitif et les types objets sont séparés et il n'existe pas un type commun

#### Type primitif

- boolean (true ou false)
- byte (8 bits signé)
- char (16 bits non signé)
- short (16 bits signé)
- int (32 bits signé)
- long (64 bits signé)
- float (32 bits flottant)
- double (64 bits flottant)

#### Type objet

Référence (réprésentation opaque)

## Type primitif vs Object

La séparation existe pour des questions de performance

#### Grosse bétise!

=> la VM devrait décider de la représentation pas le programmeur

Impact majeur sur le design du langage :(

## Auto-boxing/Auto-unboxing

Le langage Java *box* et *unbox* automatiquement

```
Auto-box
int i = 3;
Integer big = i;
Auto-unbox
Integer big = ...
int i = big;
```

Le compilateur appel *Wrapper*.valueOf() et *Wrapper.xxx*Value().

#### Identitée

Faire un new sur un wrapper type est déconseillé (warning)

- Ce sera erreur dans une future version
- L'identitée d'un wrapper (addresse en mémoire) est pas bien définie

L'implantation actuelle partage au moins les valeurs de -128 à 127

```
Integer val = 3;

Integer val2 = 3;

val == val2 // true

Integer val = -200;

Integer val2 = -200;

val == val2 // true or false, on sait pas
```

On fait **pas** des **==** ou **!=** sur les wrappers Java 16 émet un warning !

#### Programmation Orienté Object Sous-typage, Interface et Liaison tardive

## Principe de Liskov

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Barbara Liskov (1988)

## Corollaire du principe de Liskov

Si dans un programme, on manipule à un **même** endroit deux instances de classes différentes alors ils sont sous-types

```
void foo(....) {
    if (...) {
        v = new Car();
    } else {
        v = new Bus();
    }
    v.drive(); // type Car | Bus
}

Exemple 1

foo(new Car());
    ...
    foo(new Bus());
    ...
    void foo(... v) {
        ...
        v.drive(); // type Car | Bus
}
Exemple 2
```

#### Interface

En Java, il n'existe pas de type A | B

On définie une interface interface Vehicle { void drive(); }

On indique que A et B implémente I

record Car() implements Vehicle { public void drive() { ... } }

record Bus() implements Vehicle { public void drive() { ... } }

Les interfaces servent au typage, pas à l'exécution!

## Liaison tardive (polymorphisme virtuel)

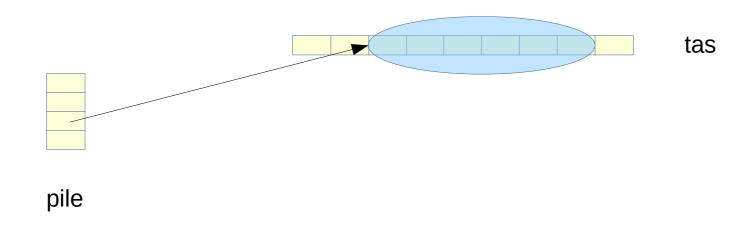
Lors de l'appel de méthode, la méthode de la classe à l'exécution est utilisée

```
void driveAll(List<Vehicle> vehicles) {
 for(Vehicle vehicle: vehicles) {
  vehicle.drive(); // à la compilation Vehicle::drive()
                   // à l'exécution appel Car::drive()
                              puis appel Bus::drive()
List<Vehicle> list = List.of(new Car(), new Bus());
driveAll(list);
```

## Classe (pour la Virtual Machine) et liaison tardive

# Objet, référence et mémoire

Un objet en Java correspond à une adresse mémoire non accessible dans le langage\*



\* les GCs peuvent bouger les objets en mémoire

# Objet et mémoire

## Le contenu d'un objet est

- alloué dans le tas (par l'instruction **new**)
  - Pas sur la pile!
  - Accessible par n'importe quel code
- Libéré par un ramasse miette (Garbage Collector) de façon non prédictible

## Classe et header

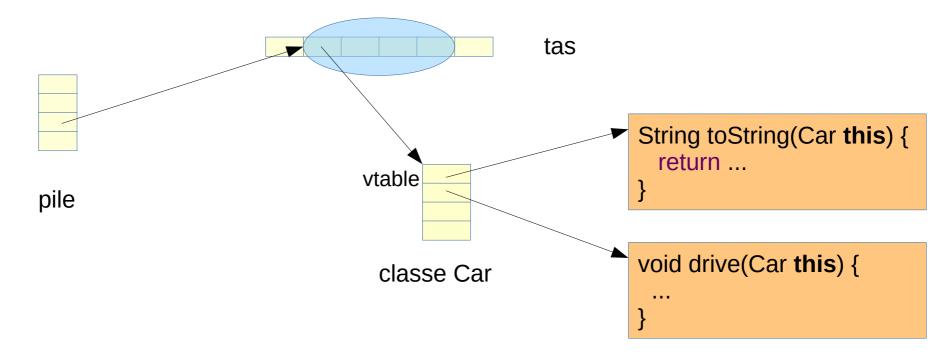
En plus des champs définies par l'utilisateur, un objet possède un entête contenant la classe ainsi que d'autre info (hashCode, le moniteur, l'état lors d'un GC, etc)

Tous les objets connaissent leur classe!

\* la vrai taille du header dépend de l'architecture et de la VM

# Méthode d'objet et mémoire

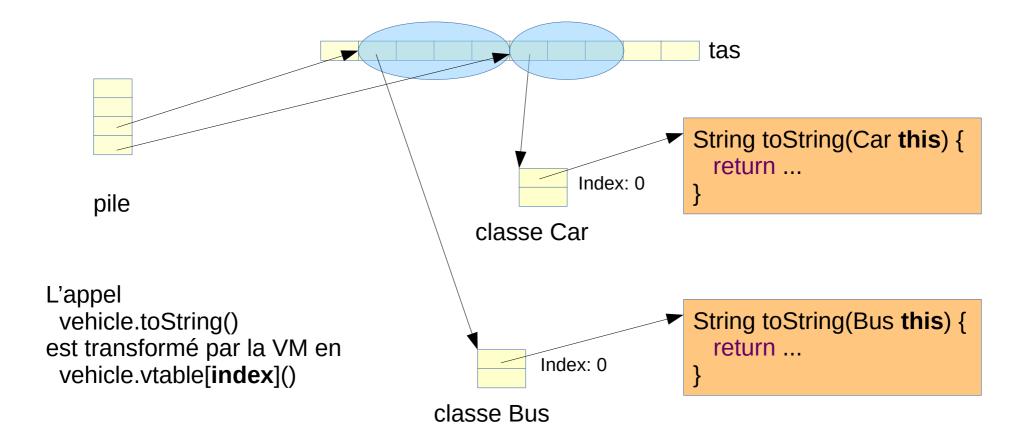
En mémoire, une méthode d'instance est un pointeur de fonction stockée dans la classe



Une méthode d'instance possède un paramètre **this implicite** (que le compilo ajoute)

## Appel Polymorphe

Si il y a redéfinition (*override*) entre deux méthodes alors les pointeurs de fonctions correspondant sont au même index dans la vtable



# Champ statique et constante

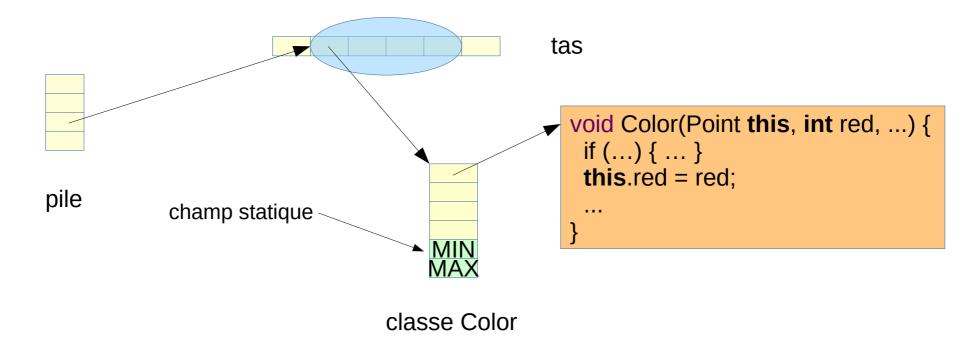
Une constante (un #define en C) est un champs static et final

```
record Color(int red, int green, int blue) {
   Color {
    if (red < MIN || red > MAX) { throw new IAE(...); }
    ...
   }
   static final int MIN = 0;
   static final int MAX = 255;
}
```

final veut dire initialisable une seul fois

# Champ statique et mémoire

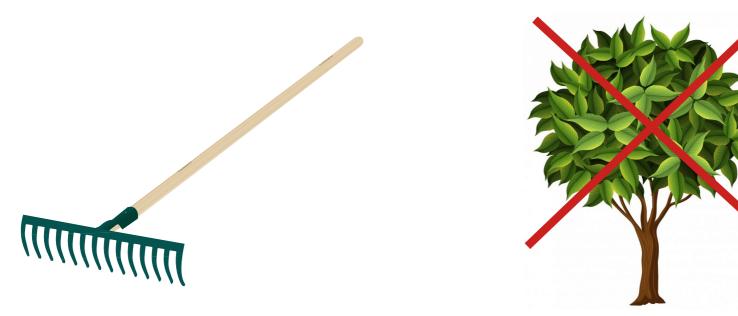
Un champ statique est une case mémoire de la classe



La valeur d'un champ statique est indépendante de la référence sur un objet (les valeurs sont stockées après la vtable)

## Rakes\* are better than trees

In gardening tips - Rémi Forax



\* rateaux

# Héritage

## L'héritage implique

- Le sous-typage (comme avec une interface)
- La copies des champs et des méthodes (pointeurs)
- La redéfinition (override) des méthodes

#### **Problèmes**

La copie des membres implique un couplage fort entre la classe de base et la sous-classe

Problème: si on met à jour la classe de base et pas les sous-classes

Problème: si la classe de base est non-mutable mais une sous-classe est mutable

# POO!= Héritage

Java a évolué sur le sujet

Historiquement (1995), l'héritage est considéré comme une bonne chose (moins de code à écrire)

Mais en pratique (2004), l'héritage rend le code difficilement maintenable

Les enums, les annotations (2004) et les records (2021) ne supportent pas l'héritage

Les langages *mainstream* créés depuis 2010, Rust et Go, ne supporte pas l'héritage non plus.

# Délégation vs Héritage

Il est plus simple de partager du code par délégation que par héritage

```
public class StudentList extends ArrayList<Student> { // ahhh
  public void add(Student student) {
    Objects.requireNonNull(student);
    super.add(student);
  }
}
```

Le code ci-dessus est faux, addAll() permet d'ajouter un étudiant null

```
public final class StudentList {
  private final ArrayList<Student> students = new ArrayList<>();
  public void add(Student student) {
    Objects.requireNonNull(student);
    students.add(student);
  }
}
```

## Préférer les interfaces

prefer rakes to trees

```
Au lieu d'utiliser l'héritage
  class Book { final String title; final String author; ... }
  class VideoGame extends Book { final int pegi; ... }
Il est plus pratique d'implanter une interface
   interface Product { String title(); String author(); }
   record Book(String title, String author)
     implements Product { }
  record VideoGame(String title, String author, int pegi)
     implements Product { }
```

## Classe Abstraite

Peut-on utiliser une classe abstraite?

Jamais à la place d'une interface

La classe abstraite ne doit **pas** être **utilisée** comme **un type**.

Oui, pour partager du code mais

La abstraite ne doit **pas** être **publique** 

=> sinon cela empèche le refactoring

la classe abstraite et les sous-classes doivent être dans le même package

## Methodes par défaut

Les méthodes par défaut permettent aussi de partager du code

Les méthodes des interfaces sont **abstract** par défaut Le mot clé **default** est le dual de **abstract** 

```
interface Product {
  abstract String title();
  default boolean titleStartsWith(String prefix) {
    return title().startWith(prefix);
  }
}
record Book(String title, String author)
  implements Product { }
```

## Programmation Orienté Data Types fermés et Pattern Matching

## Liaison tardive vs instanceof

#### On peut écrire le même code

- soit en utilisant la liaison tardive
- soit en utilisant une cascade de instanceof

```
interface Vehicle {
  void drive();
}
record Car() implements Vehicle {
  public void drive() { /* 1 */ }
}
record Bus() implements Vehicle {
  public void drive() { /* 2 */ }
}
```

```
interface Vehicle { }
record Car() implements Vehicle { }
record Bus() implements Vehicle { }

public void drive(Vehicle vehicle) {
  if (vehicle instanceof Car car) {
    /* 1 */
  } else if (vehicle instanceof Bus bus) {
    /* 2 */
  }
  throw new ISE(...);
}
```

## Instanceof est pas maintenable

Le pattern matching vérifie **l'exhaustivité** du switch si l'interface est scéllée (sealed)

```
interface Vehicle { }
record Car() implements Vehicle { }
record Bus() implements Vehicle { }
public void drive(Vehicle vehicle) {
 if (vehicle instanceof Car car) {
  /* 1 */
 } else if (vehicle instanceof Bus bus) {
  /* 2 */
 throw new ISE(...);
```

```
sealed interface Vehicle
  permits Car, Bus { }
record Car() implements Vehicle { }
record Bus() implements Vehicle { }
public void drive(Vehicle vehicle) {
  switch(vehicle) {
   case Car car -> /* 1 */
   case Bus bus -> /* 2 */
   // pas de default!
  }
}
MIEUX
```

# Liaison tardive vs pattern matching (Wadler's Expression Problem)

#### On peut

- soit étendre en ajoutant des sous-types (liaison tardive)
- soit des nouvelles opérations (pattern matching) mais pas les deux

```
interface Vehicle {
  void drive();
}
record Car() implements Vehicle {
  public void drive() { /* 1 */ }
}
record Bus() implements Vehicle {
  public void drive() { /* 2 */ }
}
```

```
sealed interface Vehicle
  permits Car, Bus { }
record Car() implements Vehicle { }
record Bus() implements Vehicle { }
public static void drive(Vehicle vehicle) {
  switch(vehicle) {
    case Car car -> /* 1 /
    case Bus bus -> /* 2 */
  }
}
```

# Programmation Orienté Donnée

Les données sont plus importantes que le code Si les données changent, le compilateur doit indiquer où faire les changements

#### On declare des types fermés

- records, enums, classes *final*, interfaces *sealed* 

#### On utilise le pattern-matching

- le switch est exhaustif (pas de default)
- record pattern plutôt que type pattern
  - on vérifie la forme des données

## Patterns acceptable par un case

Le type pattern (on match le type ou un sous-type)
 Type variable
 Le record pattern (on décompose les composants d'un record)
 Type(pattern\_component1, pattern\_component2)
 Le var pattern (on infère le type)

#### Exemple

var variable

```
- switch(vehicle) {
   case Car(var weight) - > ... // record pattern + var pattern
   case Bus bus - > ... // type pattern
}
```

# Case null et guard

- null est pas accepté à part si il y a un "case null"
  - case null
- Guard, on peut spécifier un guard avec "when" après un "case"
  - case ... when x == 2

#### Exemple

# Instanceof + switch expression

Instanceof peut utiliser les mêmes pattern que le switch

```
Vehicle vehicle = ...
if (vehicle instanceof Car(var seats, var weight)) {
  // on peut utiliser seats et weight ici !
}
```

Rappel: un switch peut renvoyer une valeur

```
var text = switch(object) {
  case String s -> s;
  case Object o -> {
    yield o.toString(); // on utilise yield pour indiquer la valeur
  }
    // d'un bloc
};
```

# Les patterns en preview

```
    Any pattern (on match toute les valeurs)

    • case -> ...

    Variable pas utilisé (aussi dans les lambdas, catch, try-with-resources)

    case String _ - > ...

- Primitive pattern (match si on peut caster sans perte)
    case long value - >
Exemple
  switch(object) {
    case Point(long x, int _) - > // primitive pattern + variable unused
    case ->
                                      // any pattern
```

## En résumé

### Le langage Java

- OOP (Encapsulation, interface, liaison tardive)
- FP (Non-mutable, lambda, interface fonctionnelle)
- DOP (interface scellée, pattern matching)

## Applications vs libraries

 Dans une application, les interfaces sont sealed, dans une librarie les interfaces sont non-sealed.