

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 paradigmatiques

C, Smalltalk, APL, Prolog, ML, Scheme, SQL

1990-2000: Les langages multi-paradigmes

Python, CLOS, Ruby, Java, JavaScript, PHP, ...

Différents styles de prog.

Un style de programmation n'exclue pas forcément les autres

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

- avec plus ou moins de bonheur :)

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

Java

Prog impérative

- Code séquentiel, Boucle, Collection (mutable)

Prog fonctionnelle

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

Prog déclarative

- Stream, Regex

Prog Orienté Object (API)

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

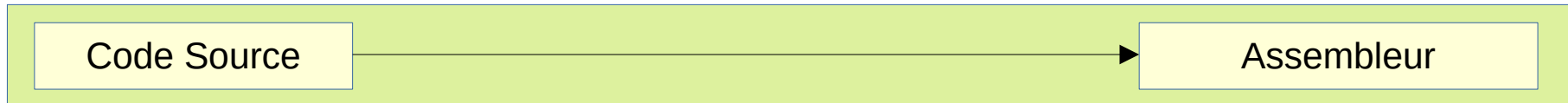
Prog Orienté Data (application)

- Record, Sealed interface, pattern matching

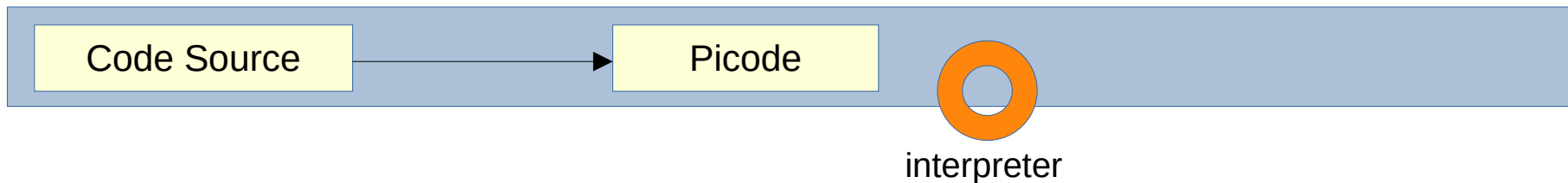
La plateforme Java

Le bytecode de Java est portable

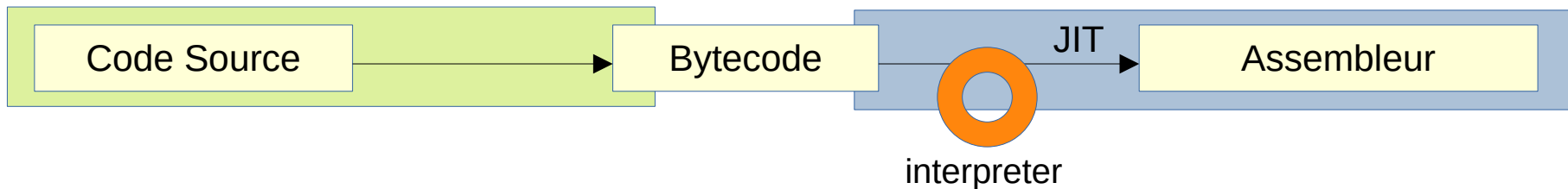
Modèle du C (1970)



Modèle de Python (1990)



Modèle de Java (1995)



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 plateforme: *Java Virtual Machine Specification (JVMS)*

<https://docs.oracle.com/javase/specs/>

Java est pas le seul langage qui fonctionne sur la plateforme Java

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

Evolution du language Java

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

Java

- POO (Java 1.0 - 1995)
- Type paramétré + Enum (Java 5 - 2004)
- Lambda + Stream (Java 8 - 2014)
- Record + Pattern matching (Java 17/21 – 2023)
- Value class (Java 28 preview - 2027)

Le monde selon Java (Edition 2026)

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.method() appelle la bonne implantation à l'exécution

Encapsulation

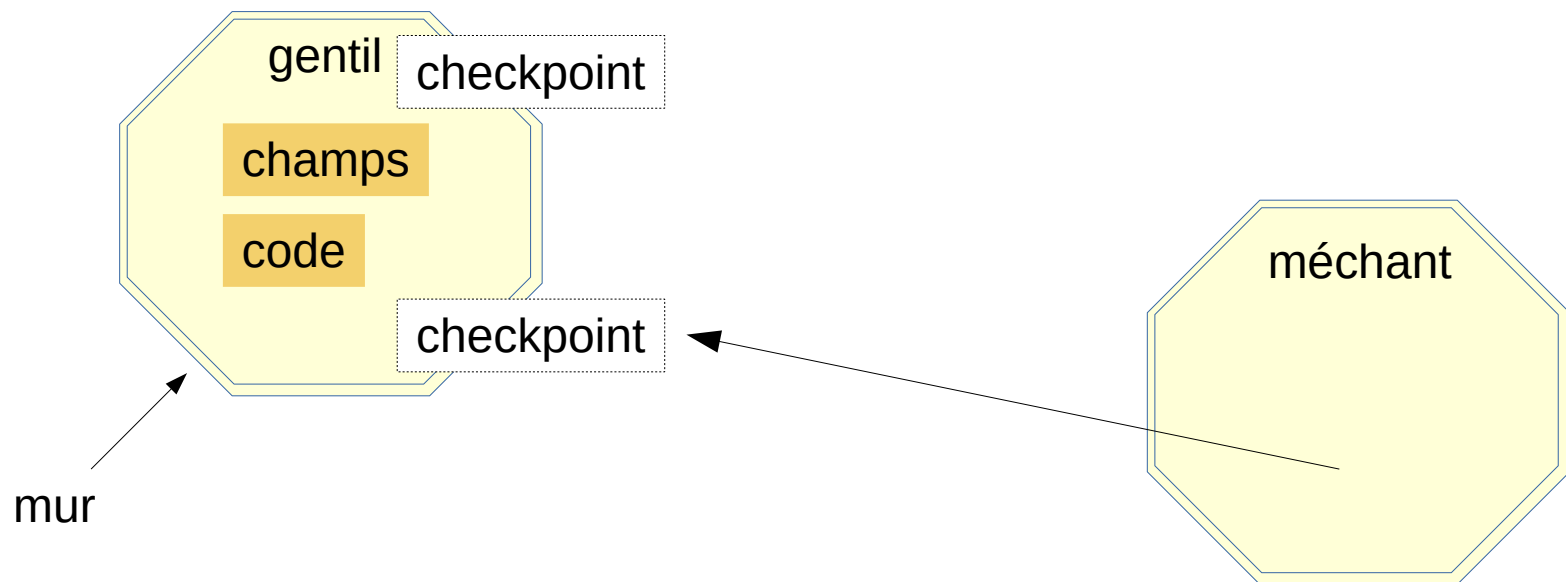
Le monde selon Trump

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 (checkpoints)



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

Intérieur vs Extérieur

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

État intérieur

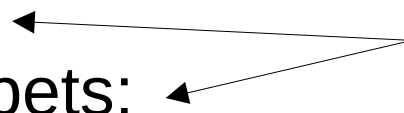
```
    public Person(String name, List<String> pets) {  
        this.name = name;  
        this.pets = pets;  
    }  
}
```

Vient de l'extérieur,
pas confiance



Classe non modifiable ??

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



Les champs doivent être initialisé à la construction

Sinon on permet l'héritage !

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

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

Constructeur Flexible

On initialise les champs AVANT l'appel au super()

```
final class Person {  
    private final String name;  
    private final List<String> pets;  
    public Person(String name, List<String> pets) {  
        this.name = name;  
        this.pets = pets;  
        super(); // à partir d'ici on accède à "this"  
    }  
    ...  
}
```

L'appel à super() doit être la dernière instruction (depuis Java 25)

Constructeur Flexible

(vs initialisation des champs)

Le compilateur copie les initialisations de champs avec '=' APRES l'appel à super() !

```
final class Person {  
    private String name = "hello"; // initialisation, ahhhh  
    public Person() {  
        super();  
        name = "hello"; // le compilateur ajoute !  
    }  
    ...  
}
```

Donc pas d'initialisation des champs avec '=' !

Encapsulation

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

Diagram annotations:

- État intérieur (points to the private fields)
- vient de l'extérieur (points to the public constructor)

```
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 ...

```
final class Person {  
    private final String name;  
    private final List<String> pets;  
    public Person(String name, List<String> pets) {  
        this.name = name;  
        this.pets = pets; // oops mistake  
        super();  
    }  
}  
  
var pets = new ArrayList<String>(); // mutable list  
var person = new Person("John", pets);  
pets.add("Garfield"); // violation !
```

Violation (2) de l'encapsulation

Autre problème, publier des objets mutables internes

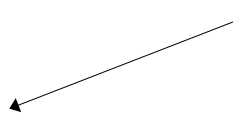
```
final class Person {  
    private final String name;  
    private final ArrayList<String> pets;  
    public Person(String name) {  
        this.name = name;  
        this.pets = new ArrayList<>(); // list mutable  
        super();  
    }  
    public void add(String pet) {  
        pets.add(pet); // oops, on peut ajouter null  
    }  
    public List<String> pets() {  
        return pets; // n'importe qui peut muter la liste  
    }  
}  
  
var person = new Person("John", pets);  
person.pets().add("Garfield"); // violation !
```

Copie défensive

Il faut faire une copie défensive des données !

```
final 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  
        super();  
    }  
    public List<String> pets() { // ok  
        return pets;  
    }  
}  
  
var person = new Person("John", List.of("Garfield"));  
person.pets().add("Garfield"); // exception !
```

List peut être modifiable



Copie défensive (2)

Il faut faire une copie défensive des données !

```
final class Person {  
    private final String name;  
    private final List<String> pets;  
    public Person(String name) {  
        this.name = name;  
        this.pets = new ArrayList<>();  
        super();  
    }  
    public List<String> pets() {  
        return Collections.unmodifiableList(pets); // defensive copy  
    }  
}  
  
var person = new Person("John", List.of("Garfield"));  
person.pets().add("Garfield"); // exception !
```

Pas d'encapsulation == vue

On peut designer une classe qui ne respecte pas le principe d'encapsulation

- On appelle cela une **vue** (*view*)
- Cela évite de recopier les données mais
 - cela doit être documentée absolument
 - car les mutations font des effets de bord :(

Les vues dans le JDK

- `Arrays.asList(T...)`, `List.subList()`,
`Map.keySet()/values()/entrySet()`,
`Collections.unmodifiableList(list)`

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 Bloch

Exemple de prog par contrat

```
final class Book {  
    ...  
    public Book(String author, long price) {  
        Objects.requireNonNull(author);  
        if (price < 0) { throw new IAE(...); }  
        super();  
    }  
  
    public long priceWithDiscount(long discount) {  
        if (discount < 0) { throw new IAE(...); }  
        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 langage 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 peut 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 adresse 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 (représentation opaque)

Type primitif vs Object

La séparation existe pour des questions de performance

Grosse bêtise !

=> 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 appelle *Wrapper.valueOf()* et *Wrapper.xxxValue()*.

Identité

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

- L'identité d'un wrapper (adresse 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

Change en Java 28 !

Programmation Orienté Object

Sous-typage, Interface et Liaison tardive

Principe de Liskov

If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of S , the behavior of P is unchanged when o_1 is substituted for o_2 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 m(...) {  
    ...  
    if (...) {  
        v = new Fox();  
    } else {  
        v = new Lion();  
    }  
    ...  
    v.noise(); // type Fox | Lion  
}
```

Exemple 1

```
m(new Fox());  
...  
m(new Lion());  
...  
void m(... v) {  
    ...  
    v.noise(); // type Fox | Lion  
}
```

Exemple 2

Interface

La syntaxe `A | B` n'existe pas en Java

On définit une interface

```
interface Animal { void noise(); }
```

On indique que A et B implémentent I

```
record Fox() implements Animal {  
  public void noise() { ... }  
}
```

```
record Lion() implements Animal {  
  public void noise() { ... }  
}
```

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 allNoises(List<Animal> animals) {  
    for(Animal animal: animals) {  
        animal.noise(); // à la compilation Animal::noise()  
                        // à l'exécution appel Fox::noise()  
                        //                puis appel Lion::noise()  
    }  
}
```

```
List<Animal> list = List.of(new Fox(), new Lion());  
allNoises(list);
```

Programmation Orienté Data

Switch, types fermés et pattern matching

Ancient switch / Nouveau switch

(Java 17)

Switch avec ':', par compatibilité avec le C

```
switch(value) {  
  case 12: // fallthrough  
  case 14:  
    ...  
    break;  
  default:  
    ...  
}
```

Switch avec ->, virgule entre les valeurs, pas de fallthrough

```
switch(value) {  
  case 12, 14 -> { // on met des accolades si on a plusieurs instructions  
    ...  
  }  
  default -> ...  
}
```

Switch expression (Java 17)

En plus du switch comme instruction, le switch peut aussi renvoyer une valeur

```
enum WeekDay { MONDAY, TUESDAY, ... }

var remoteWork = switch(weekDay) {
  case MONDAY, FRIDAY -> true;
  case TUESDAY, WEDNESDAY, THURSDAY {
    ...
    yield false; // on sort du switch
  }
};
```

Le switch doit être **exhaustif** (faire tous les cas)

Problème switch instruction et enum

Le switch instruction sur un enum ne demande pas l'exhaustivité :(

- Ajouté en Java 1.5, ne peut pas être changé sans casser la compatibilité
- Par ex:

```
switch(color) {  
    case RED -> ...  
    case GREEN -> ...  
    case BLUE -> ...  
} // le compilateur ajoute un "default" ici, ahhhh !
```

- Pour avoir un switch exhaustif, on ajoute
case null -> throw null; (cf plus tard)

Pattern Matching et switch

(Java 21 a deux switch)

On peut faire des switchs sur

- Un nombre de cas finis ($\leq$ *pattern matching*)
 - Interface scellée, enum
 - On **ne** doit **pas** écrire de *default*
 - Le compilateur doit planter si on se trompe
- Un nombre de cas infinis (ou très grand)
 - String ou entier
 - On doit écrire un *default*.
 - On ne peut pas lister tous les cas

Les patterns

Le switch sur les objets à différentes syntaxes pour les “cases”

- Le *type pattern* “Type variable” permet de matcher un type (pareil que instanceof)
- Le *record pattern* “Type(...)” permet de matcher les composants d’un record
- Nested Patterns (contreversés)
 - *var pattern* “var variable” permet de ne pas répéter le type
 - *unnamed pattern* “_” reconnaît n’importe quoi

Utilisation du *record pattern*

```
sealed interface Payment permits VisaCard, Bitcoin {}  
record VisaCard(long serial, String owner) implements Payment {}  
record Bitcoin(Object wallet, String owner) implements Payment {}
```

Le compilateur détecte les changements au niveau des données

- Si nouveau sous-type => ajouter un *case*
- Si nouveau composant d'un record => changer le *case*

```
switch(payment) {  
  case BitCoin(Object wallet, String owner) - > ...  
  case VisaCard(long serial, String _) -> ...  
}
```

Null et case partiel

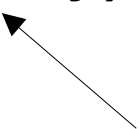
Par défaut `switch(null)` plante avec une NPE

- On peut ajouter un “case null”
- On peut combiner le “case null” avec *default*
case null, default -> ...

“when” permet de définir des “case” partiels

```
switch(point) {  
  case Point(int x, int y) when x == y -> ...  
  case Point(int x, int y) -> ...  
}
```

Il faut aussi un case “général”



Programmation Orienté Donnée

Les données sont plus importantes que le code

Si les données changent, le compilateur doit indiquer où faire les changements

On modélise les données avec des types fermés

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

On utilise le pattern-matching

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

Liaison tardive vs instanceof

On peut écrire le même code

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

```
interface Animal {  
    void noise();  
}  
record Fox() implements Animal {  
    public void noise() { /* 1 */ }  
}  
record Lion() implements Animal {  
    public void noise() { /* 2 */ }  
}
```

```
interface Animal {}  
record Fox() implements Animal {}  
record Lion() implements Animal {}  
public void noise(Animal animal) {  
    if (animal instanceof Fox fox) {  
        /* 1 */  
    } else if (animal instanceof Lion lion) {  
        /* 2 */  
    }  
    throw new IAE(...);  
}
```

↑
BEURK

Polymorphisme vs Pattern Matching

Instanceof est pas maintenable

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

```
interface Animal { }
record Fox() implements Animal { }
record Lion() implements Animal { }

public void noise(Animal animal) {
    if (animal instanceof Fox fox) {
        /* 1 */
    } else if (animal instanceof Lion lion) {
        /* 2 */
    }
    throw new IAE(...);
}
```

↑
BEURK

```
sealed interface Animal
    permits Fox, Lion { }
record Fox() implements Animal { }
record Lion() implements Animal { }

public void noise(Animal animal) {
    switch(animal) {
        case Fox fox -> /* 1 */
        case Lion lion -> /* 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 Animal {  
  void noise();  
}  
record Fox() implements Animal {  
  public void noise() { /* 1 */ }  
}  
record Lion() implements Animal {  
  public void noise() { /* 2 */ }  
}
```

On permet d'ajouter des sous-types

```
sealed interface Animal  
  permits Fox, Lion { }  
record Fox() implements Animal { }  
record Lion() implements Animal { }  
public void noise(Animal animal) {  
  switch(animal) {  
    case Fox fox -> /* 1 */  
    case Lion lion -> /* 2 */  
    // pas de default !  
  }  
}
```

On permet d'ajouter des opérations

Rakes* are better than trees
(l'héritage c'est nul)



* rateaux

Héritage

L'héritage implique

- Le sous-typage (comme avec une interface)
- La copie 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 (l'API n'est pas figée, classe forcément mutable)

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

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

Délégation vs Héritage

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

```
public class Car extends ArrayList<Passenger> {  
    // my public API get all the methods of ArrayList, ahhhh  
}
```

On utilise la délégation pour contrôler l'API

```
public final class Car {  
    private final ArrayList<Passenger> passengers;  
  
    public Car() {  
        passengers = new ArrayList<>();  
        super();  
    }  
  
    public void add(Passenger passenger) {  
        Objects.requireNonNull(passenger);  
        passengers.add(passenger);  
    }  
}
```

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 classe abstraite ne doit **pas** être **publique**

=> **sinon cela empêche le refactoring**

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

Methodes par défaut

Permet d'ajouter une méthode abstraite quand l'interface a déjà été publiée

Les méthodes des interfaces sont **abstract** par défaut

Le mot clé **default** est le dual de **abstract**

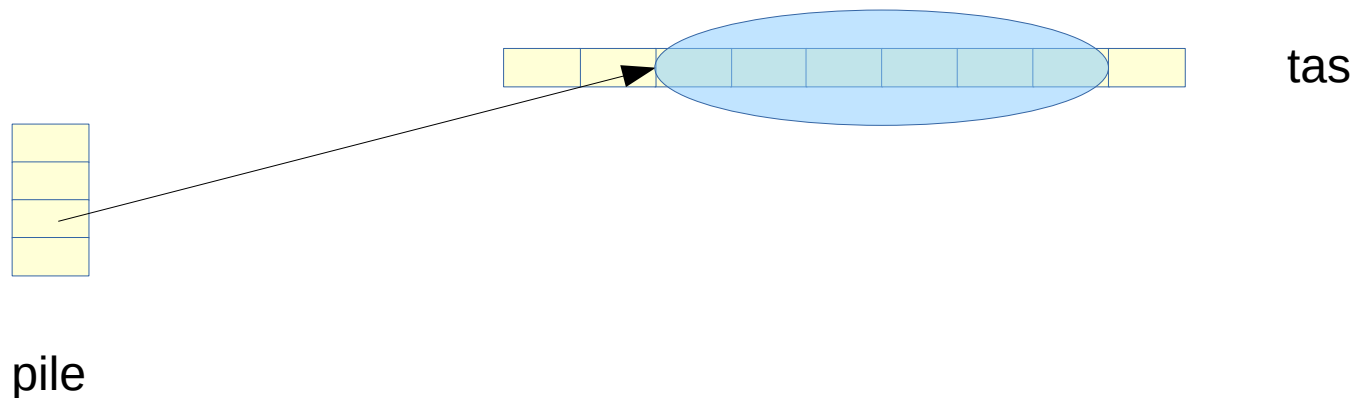
```
interface Product {  
    abstract String title();  
    default boolean titleStartsWith(String prefix) {  
        return title().startsWith(prefix);  
    }  
}  
  
record Book(String title, String author)  
    implements Product { }
```

Objets et Classe à l'exécution

(pour la Virtual Machine)

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)

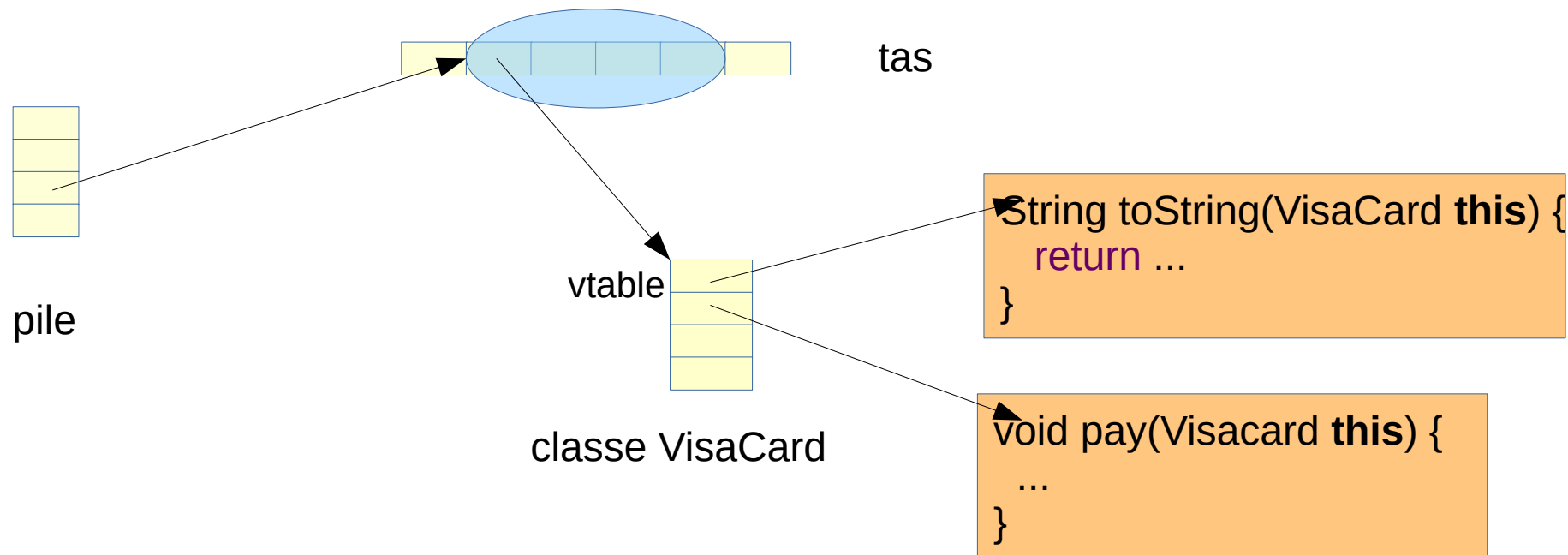
```
class VisaCard {  
    // pointeur sur la classe +  
    // hashCode + lock + GC bits = header 96bits*  
    long serial;    // long => 64bits  
    String owner;  // 32bits or 64bits  
}
```

Tous les objets connaissent leur classe !

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

Méthode d'objet et mémoire

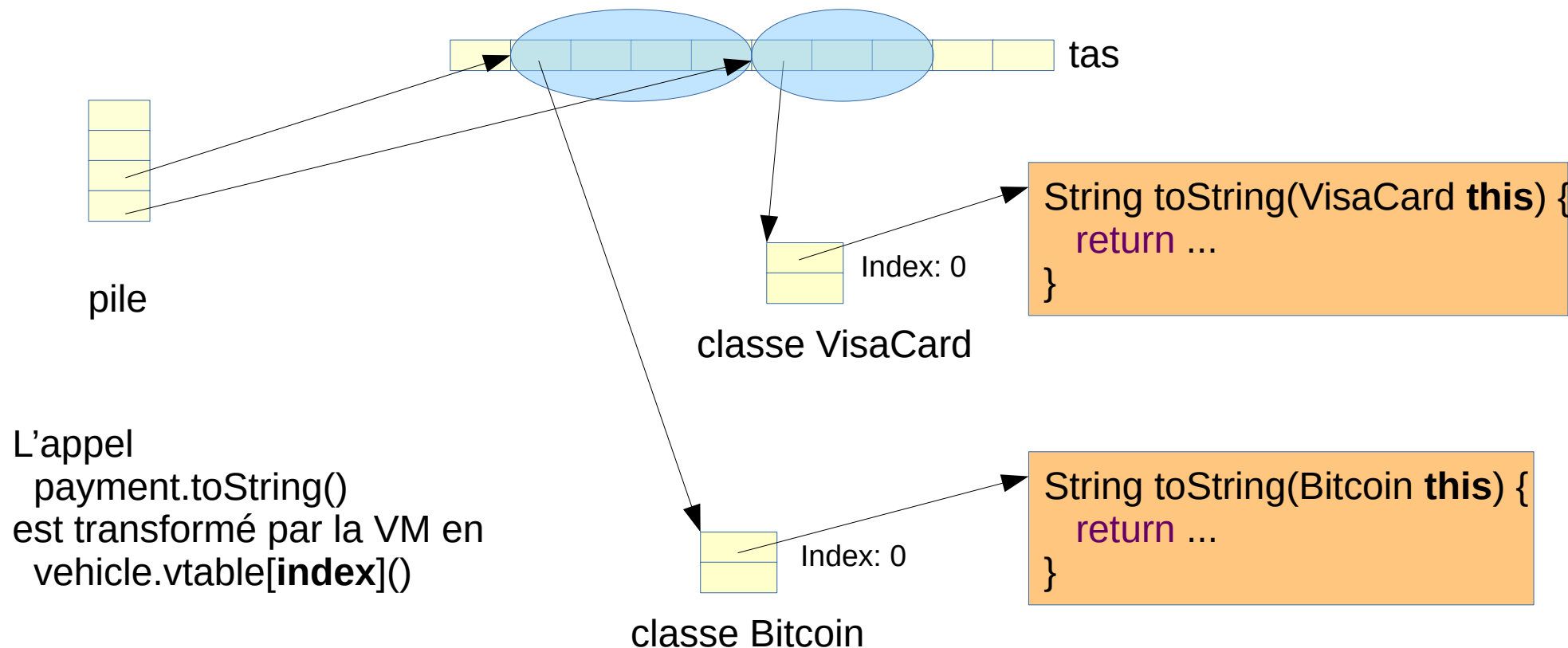
En mémoire, une méthode d'instance est un pointeur de fonction stockée dans la classe



Une méthode d'instance possède un paramètre **this implicite** (que le compilateur 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

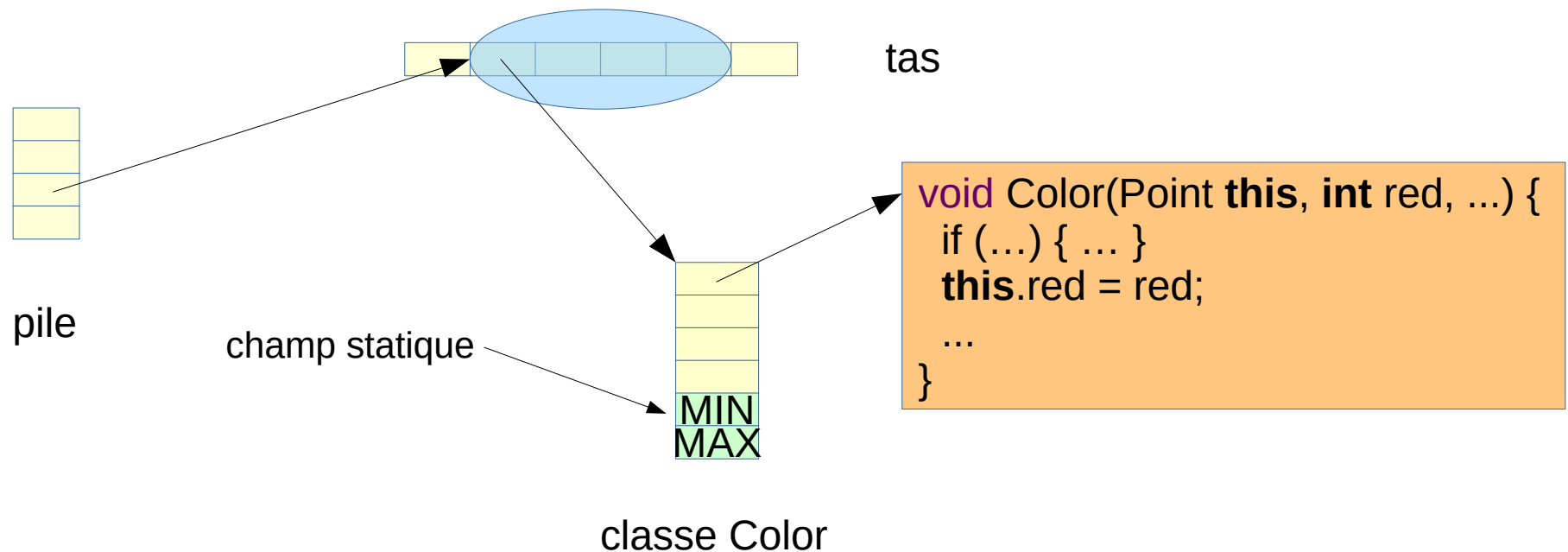
Une constante (un `#define` en C) est un champs **static** et **final**

```
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)

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 librairie les interfaces peuvent être *non-sealed*.