

# 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

# 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

## **applicatif ou fonctionnel** (Scheme, Caml, Haskell)

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

## **objet** (Modula, Objective-C, Self, C++)

- unités réutilisables, abstraites, contrôle les effets de bord

# 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 :)

# La plateforme Java

# Le bytecode est portable

## Architecture en C

code C → compilateur → assembleur (sur disque)

## Architecture en Java

code Java → compilateur → bytecode

bytecode → machine virtuelle → assembleur (en mémoire)

La machine virtuelle (JVM) transforme le code en assembleur (JIT) lorsqu'un code est interprété souvent (hot code)

# Avantages et Inconvénients

## Pour que le C soit portable

- il n'est pas possible d'optimiser le code pour un CPU spécifique
- La compilation peut prendre plusieurs minutes

## Pour que Java soit portable

- Le compilateur ne doit pas faire d'optimisation
- Le JIT peut optimiser pour un CPU spécifique mais ne doit pas prendre trop de temps car les optimisations ont lieu à l'exécution

# OpenJDK vs Java

OpenJDK contient les sources

<http://github.com/openjdk/jdk>

Java est un ensemble de distributions

Oracle Java, RedHat Java, Amazon Java, Azul Java, SAP Java, ...

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

# Le langage Java

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

La VM ne connaît 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 langage Java

Les langages de programmation évoluent...

... car les besoins des logiciels évoluent

## Java

- OOP (Java 1 - 1995)
- Type paramétré + Enum (Java 5 - 2004)
- Lambda (Java 8 - 2014)
- Pattern matching + Record (Java 23 – 2024?)

# Le monde selon Java (Edition 2020)

# Object Oriented Programming

## En Java

- Polymorphism (à l'exécution)
  - `o.toString()` appelle la bonne implantation
- Interface (typage)
  - Pour appeler des implantations différentes d'une même méthode, il faut déclarer la méthode dans l'interface
- Classe (encapsulation)
  - Séparation de l'API publique et de l'implantation interne

# Encapsulation

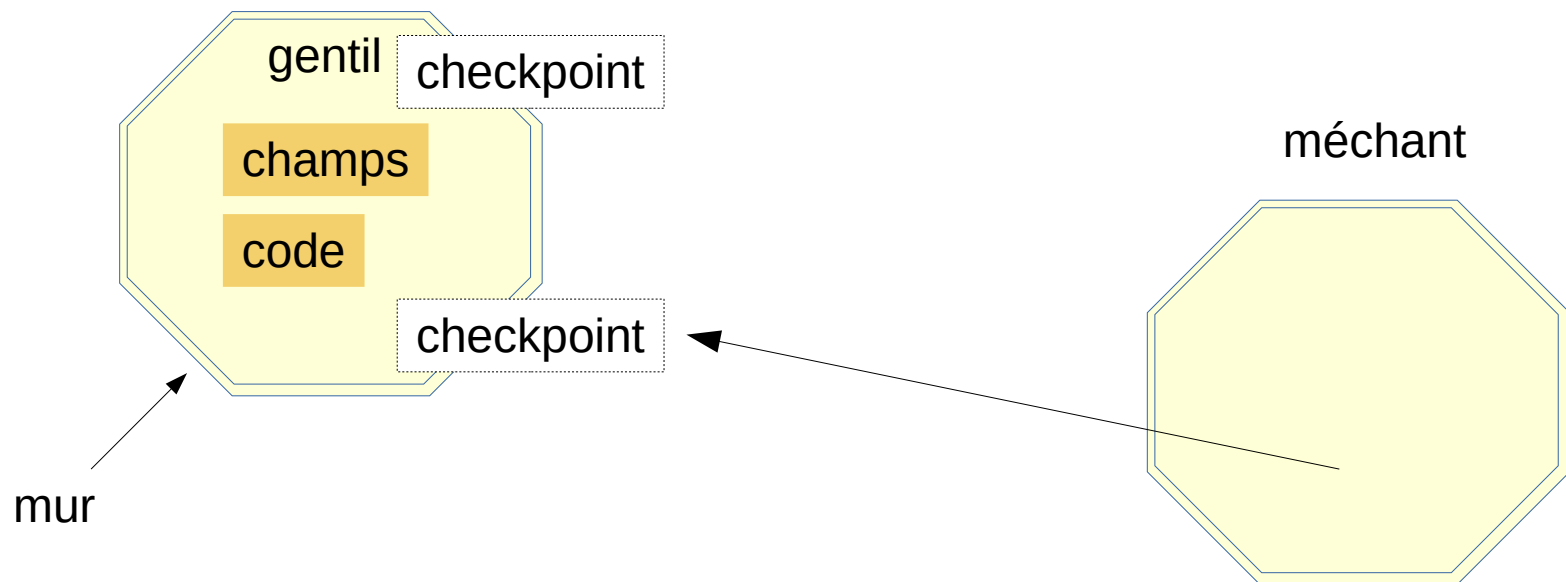
# Le monde selon Trump

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

## Définition

La seule façon de modifier l'état d'un objet est de passer par une des méthodes publiques

```
class Person {  
    private String name;  
    private List<String> pets;  
  
    public Person(String name, List<String> pets) {  
        this.name = Objects.requireNonNull(name);  
        this.pets = pets;  
    }  
}  
  
var person = new Person("John", List.of("Garfield"));
```

# Classe Non Mutable

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

les records sont non modifiables !

```
record Person(String name, List<String> pets) {  
  public Person {  
    Objects.requireNonNull(name);  
    pets = List.copyOf(pets); // copie défensive !!  
  }  
}  
  
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

```
record Person(String name, List<String> pets) {  
    // pas de copie défensive ici :(  
}
```

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

# 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

# En Java

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

# Exemple de prog par contrat

```
record Book(String author, long price) {  
  public Book {  
    Objects.requireNonNull(author);  
    if (price < 0) { throw new IAE(...); }  
  }  
  
  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
```

# Design by contract vs Unit Testing

Prog. par contrat != Tests Unitaires

La prog. par contrat insère des tests dans le code, par ex en utilisant `Objects.requireNonNull`, `Objects.checkIndex`, etc

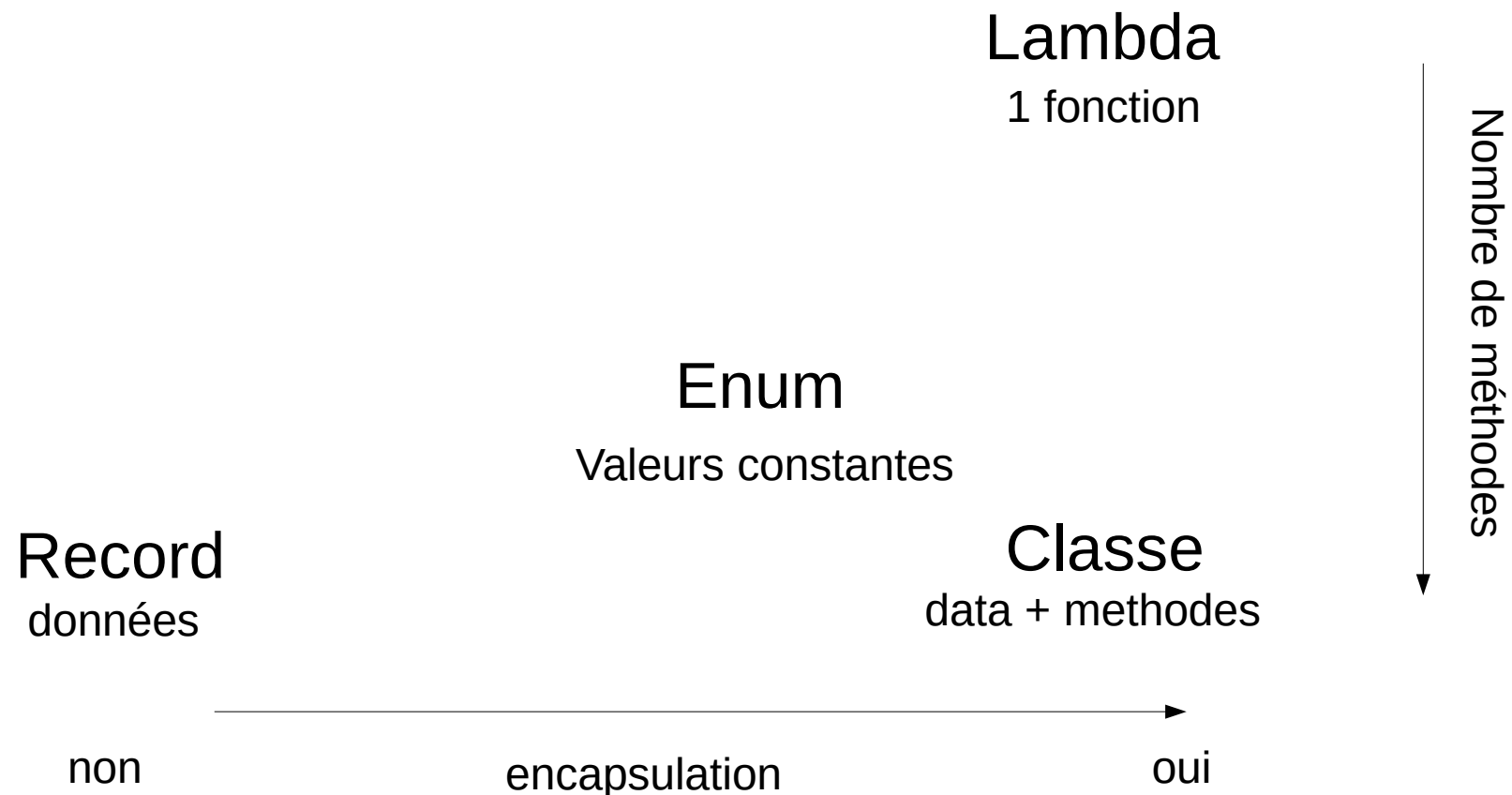
Un test unitaires teste le code de façon externe (soit l'API publique, soit l'API privée)



# Record, Lambda et Enum

# Représentations simplifiées

Les **records**, les **lambdas** et les **enums** sont des représentations simplifiées des classes



# Record comme une Classe

Un record ne permet pas l'encapsulation

```
record Point(int x, int y) { // x et y sont des composants (pas de modificateur)  
    // pas de champ supplémentaire ici !  
    public double distanceToOrigin() { return Math.sqrt(x * x + y * y); }  
}
```

est équivalent à

```
class Point {  
    private final int x; // champs forcément final  
    private final int y;  
    public Point(int x, int y) { this.x = x; this.y = y; } // constructeur canonique  
    public int x() { return x; } // accesseurs  
    public int y() { return y; }  
    + toString(), equals(Object) et hashCode() sont fournies  
    public double distanceToOrigin() { return Math.sqrt(x * x + y * y); }  
}
```

# Restrictions sur les records

## Force les bonnes pratiques

### Champs

Pas de champ explicite (que des composants)

### Constructeurs

#### Constructeur canonique

- Le noms des paramètres doit être le même que les composants
- Pas d'exceptions checkées !

Surcharge => Délégation

### Accesseurs

Peuvent être redéfinie (penser au @Override)

# Constructeur compact

Un constructeur compact est un constructeur canonique sans les assignations

```
record Person(String name, List<String> pets) {  
    Person { // pas de parenthèses !  
        Objects.requireNonNull(name);  
        pets = List.copyOf(pets);  
    }  
}
```

Les instructions

**this.name** = name et **this.pets** = pets

sont ajoutées à la fin

# Lambda comme une classe

Lambda

```
int x = 3;
```

```
IntBinaryOperator op = (a, b) → a + b + x; // la valeur de x est capturée
```

est équivalent à

```
class $lambda1$IntBinaryOperator // nom généré
```

```
implements IntBinaryOperator {
```

```
private final int x;
```

```
$lambda1$IntBinaryOperator(int x) { this.x = x; }
```

```
@Override
```

```
public int applyAsInt(int a, int b) {
```

```
return a + b + x;
```

```
}
```

```
}
```

```
int x = 3;
```

```
IntBinaryOperator op = new LambdaIntBinaryOperator(x);
```

# Restrictions sur les lambdas

## Force les bonnes pratiques

Une lambda doit avoir une interface comme type

- Elle ne peut pas implanter une classe/classe abstraite
- L'interface doit être fonctionnelle
  - Avoir 1 seule méthode abstraite  
(les méthodes statiques et par défaut sont autorisées)

Les variables capturées doivent être effectivement *final* (assignée 1 seul fois)

Java capture les valeurs pas les variables

# Enum comme une Classe

Un Enum a que des valeurs (instances) constantes

```
enum Option {  
    READ, WRITE, READ_WRITE  
    ;  
    public boolean isReadable() {  
        return switch(this) { // les constantes sont numérotées automatiquement  
            case READ, READ_WRITE → true;  
            case WRITE → false;  
        };  
    }  
}
```

est équivalent à

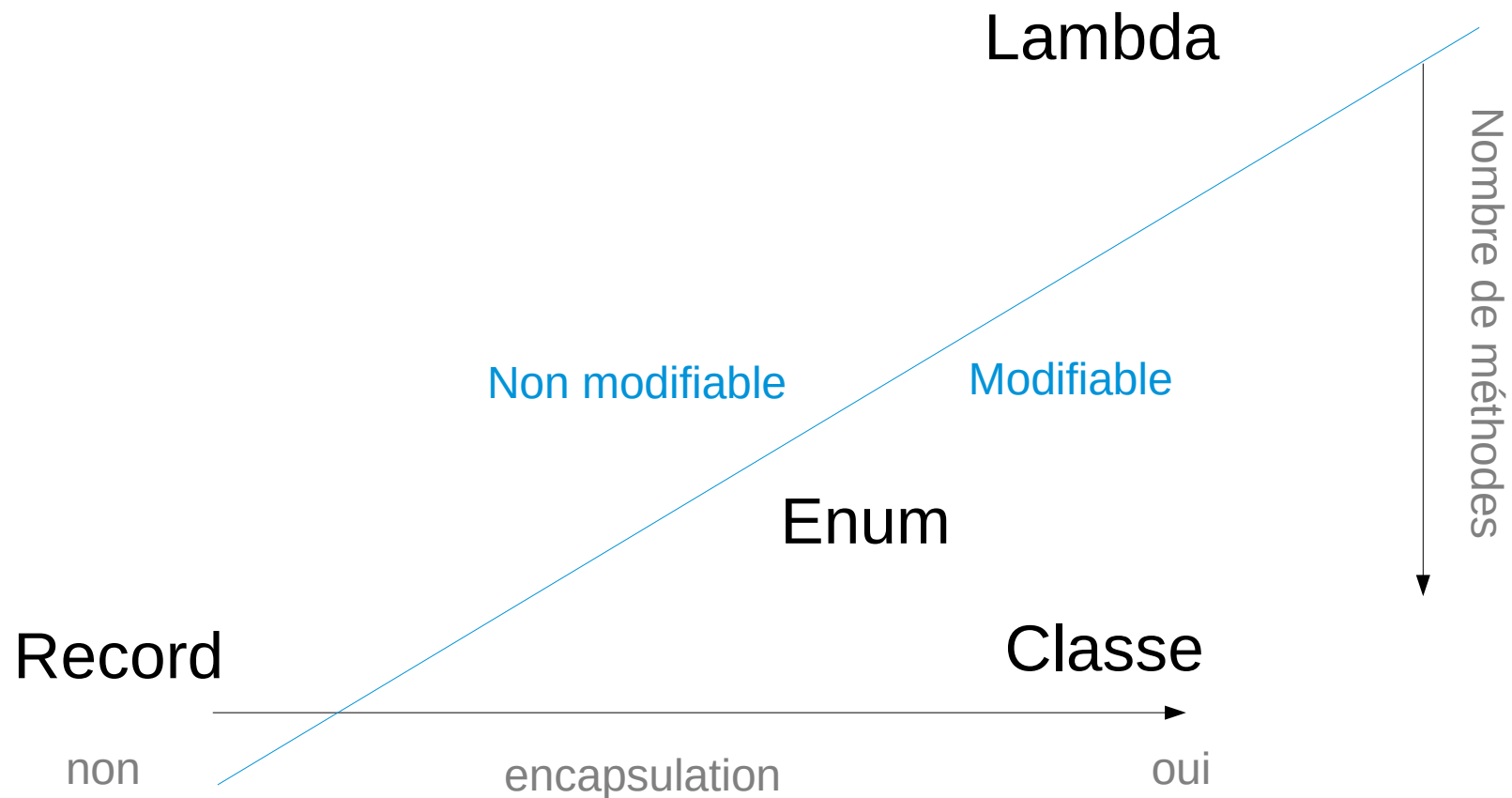
```
class Option extends java.lang.Enum {  
    private Option(String name, int ordinal) { super(name, ordinal); }  
    public static final Option READ = new Option("READ", 0);  
    public static final Option WRITE = new Option("WRITE", 1);  
    public static final Option READ_WRITE = new Option("READ_WRITE", 2);  
    public boolean isReadable() { ... }  
}
```



# Contrôle des mutations

Les données des records et lambda sont non modifiables

Les données des classes et des enums peuvent être modifiables



# Non Modifiable: Record et Lambda

Les composants d'un record sont non modifiables

```
record Point(int x, int y) {  
    public void setX(int x) { this.x = x; } // compile pas  
}  
var point = new Point(2, 3);  
System.out.println(point.x()); // non modifiable :)
```

Les variables capturées par une lambda sont non modifiables

```
int x = 3;  
IntBinaryOperator op = (a, b) → a + b + x;  
x = 5; // compile pas
```

Les variables capturées des lambdas ne sont pas accessibles de l'extérieur

# 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 réifié

Un type réifié est un type qui existe à la compilation **et** à l'exécution

Java a besoin des types réifiés à l'exécution pour les opérations

- `new X()`, `new X[]`, `instanceof X`, `cast: (X)`, `catch(X)`

Exemple de type non-réifié

Type paramétré: `List<String>`

Variable de type: `<T> void foo(T t) { ... }`

Les wildcards non bornées, e.g. `List<?>`, sont réifiés

# 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 :(



# 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  $T$ , 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 (...) {  
    i = new A();  
  } else {  
    i = new B();  
  }  
  ...  
  i.m(); // type A | B  
}
```

Exemple 1

```
m(new A());  
...  
m(new B());  
...  
void m(... i) {  
  ...  
  i.m(); // type A | B  
}
```

Exemple 2

# Interface

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

On définit une interface

```
interface Itf { void m(); }
```

On indique que A et B implémentent I

```
record A() implements Itf { public void m() { ... } }
```

```
class B implements Itf { public void m() { ... } }
```

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

# Hierarchie ouverte / fermée

Interface ouverte: Itf = A | B | ...

```
interface Itf { }
```

Interface fermée: Itf = A | B

```
sealed interface Itf permits A, B { }
```

la clause “permits” peut être inférée par le compilateur, si Itf, A et B sont déclarées dans le même fichier .java

# Interface et Typage

La notion d'interface est le concept central de Java  
classes, records et enums

```
enum Foo implements Itf { public void m() { ... } }
```

sous-interfaces

```
interface Itf2 extends Itf { void m2(); }
```

classes anonymes

```
Itf anonymous = new Itf() { public void m() { ... } };
```

lambdas

```
Itf lambda = () => System.out.println("hello");
```

Rakes\* are better than trees

*In gardening tips - Rémi Forax*

\* 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

# Délégation vs Héritage

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

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

Le code ci-dessus est **faux**, on peut par exemple utiliser `addAll()` pour 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);  
    }  
}
```



# 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 (2020) ne supportent pas l'héritage

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

# 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'utiliser l'implantation d'interface

```
sealed 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 et les sous-classes doivent être dans le **même package**

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

=> **sinon cela empêche le refactoring**

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

Exception

# Exception

Mécanisme qui évite d'utiliser le type de retour pour signaler une erreur

En C:

```
String readLine() {  
    ... lit la ligne ou envoie null  
}  
String readAuthor() {  
    String name;  
    if ((name = readLine()) == null) {  
        return null;  
    }  
    return new Author(name);  
}  
...  
Author author;  
if ((author = readAuthor()) == null) {  
    return null;  
}  
...
```

En Java:

```
String readLine() {  
    ... lit la ligne ou lève une exception  
}  
String readAuthor() {  
    return new Author(readLine());  
}  
...  
var author = readAuthor();
```

# Checked Exception

Java a deux sortes d'exceptions, les exceptions normales (RuntimeException) ...

Runtime Exception:

```
String readLine() {  
    ... throw new RuntimeException(...);  
}  
String readAuthor() {  
    return new Author(readLine());  
}  
...  
var author = readAuthor();
```

# Checked Exception (2)

Java a deux sortes d'exceptions, les exceptions normales (RuntimeException) et celle vérifiée par le compilateur (Exception)

Runtime Exception:

```
String readLine() {  
    ... throw new RuntimeException(...);  
}  
String readAuthor() {  
    return new Author(readLine());  
}  
...  
var author = readAuthor();
```

*Checked* Exception:

```
String readLine() throws IOException {  
    ... throw new IOException(...);  
}  
String readAuthor() throws IOException {  
    return new Author(readLine());  
}  
...  
try {  
    var author = readAuthor();  
} catch(IOException e) {  
    System.out.println("error " + e);  
    System.exit(1);  
}
```



# Checked Exception vs Sous-typage

Problème:

Une exception checkée fait partie de la signature d'une méthode

```
Runnable runnable = () → { /* appel qui peut lever IOException */ }; // compile pas
```

En pratique,

Si on crée sa propre exception, elle doit hériter de RuntimeException

```
class MyException extends RuntimeException { }
```

On met les exceptions checkées dans des exceptions non checkées

```
String readLine() { // si pas de throws
    try {
        // appel qui peut lever IOException
    } catch(IOException e) {
        throw new UncheckedIOException(e); // on lève une runtime exception
    }
}
```

# API standard et exceptions

L'API de Java, surtout `java.io`, `java.nio`, `java.net` utilise des exceptions pour représenter des comportements normaux

C'est pas bien !

mais on doit vivre avec :(

Par ex, `InterruptedException` ou `TimeoutException` représente un comportement normal de l'application

# Primitive vs Objet

# Tableaux

Les tableaux en Java sont des Objects

Chaque cellule est

Soit un type primitif

- byte[], int[], etc

Soit un Object

- String[], Object[], etc

Le super-type commun de tous les tableaux est  
Object pas Object[]

- on a pas accès à la longueur :(

# Tableau d'objet

Les tableaux d'objets en Java connaissent leur classe à l'exécution

- La classe du tableau est stockée dans le header

Les tableaux sont covariants (boulette !)

```
String[] array = new String[] { "hello" };  
Object[] arrayObject = array;  
System.out.println(array == arrayObject); // true
```

```
arrayObject[0] = new Point(1, 1); // ArrayStoreException
```

A chaque fois que l'on stocke un objet dans un tableau, la VM vérifie que l'objet est compatible avec la classe du tableau

# Wrappers / boxing

Comme il n'y a pas de type commun aux types primitif et aux types Object

- On utilise `java.lang.Object`
- On convertit le type primitif en objet (boxing)

Pour chaque type primitif, il existe un type objet (wrapper) qui possède un champ *value* du type du type primitif.

- Par ex. `java.lang.Integer` possède un champ *value* de type `int`.

# Les wrappers

Chaque type primitif à son wrapper, void aussi !

primitif	wrapper	box	unbox
void	java.lang.Void	-	-
boolean	java.lang.Boolean	Boolean.valueOf()	booleanValue()
byte	java.lang.Byte	Byte.valueOf()	byteValue()
char	java.lang.Character	Character.valueOf()	charValue()
short	java.lang.Short	Short.valueOf()	shortValue()
int	java.lang.Integer	Integer.valueOf()	intValue()
long	java.lang.Long	Long.valueOf()	longValue()
float	java.lang.Float	Float.valueOf()	floatValue()
double	java.lang.Double	Double.valueOf()	doubleValue()

# 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)

- Ce sera erreur dans une future version
- 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

Java 16 émet un warning !

# Module, package, classe et encapsulation

# Modificateurs de Visibilité

Java possède 4 modificateurs de visibilité

mais on en utilise que 2 ½

private et public

la visibilité de package de temps en temps

protected et la visibilité de package émule la visibilité du C++

protected rend le champs visible pour les sous-classes

- viole le principe d'encapsulation et
- suppose que la classe de base et la sous classe sont pas dans le même package ??

la visibilité de package n'est pas nécessaire entre classes internes

# Les poupées russes

Un module se définit en créant un fichier module-info.java

```
module fr.umlv.hello {  
  export fr.umlv.hello.api;  
}
```

Un package se définit en même temps qu'une classe

```
package fr.umlv.hello.api;
```

```
public class FunnelBuilder {  
  private class Into { // une classe peut avoir des classes internes  
    ...  
  }  
}
```

# Module, package, classe

Les membres d'une classe peuvent être *public* ou *private*

- *Private* veut dire pas visible par les autres classes

Les classes peuvent être *public* (ou pas *public*)

- Pas *public* veut dire pas visible par les autres packages

Les packages peuvent être *exporté* (ou pas *exporté*)

- Pas *exporté* veut dire pas visible par les autres modules

# Classe pour la Virtual Machine

# 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 Point {  
    // pointeur sur la classe +  
    // hashCode + lock + GC bits = header 64bits*  
    int x;    // int => 32bits => 4 octets  
    double y; // double => 64bits => 8 octets  
}
```

Tous les objets connaissent leur classe !

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

# Objet et mémoire

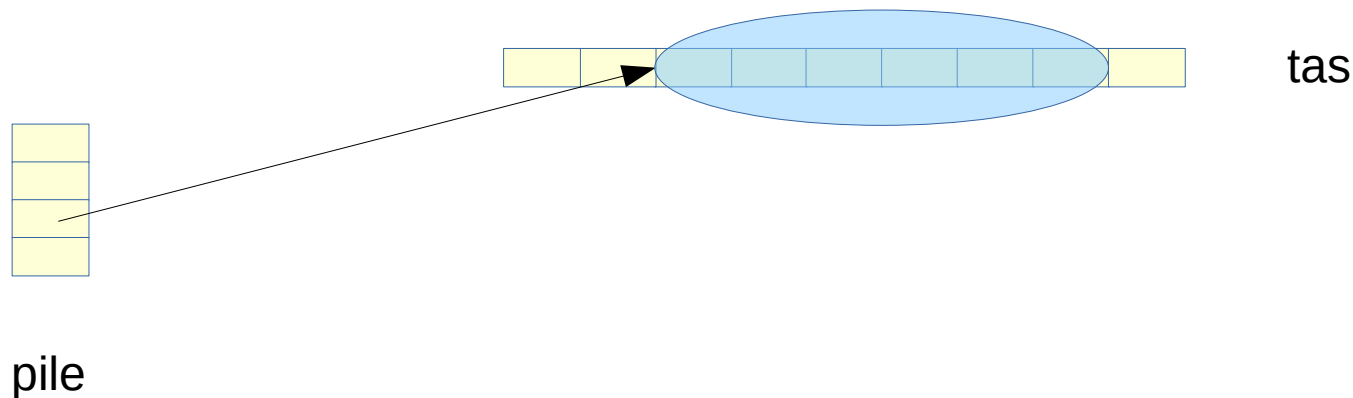
Le contenu d'un objet est

- 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



# Objet, référence et mémoire

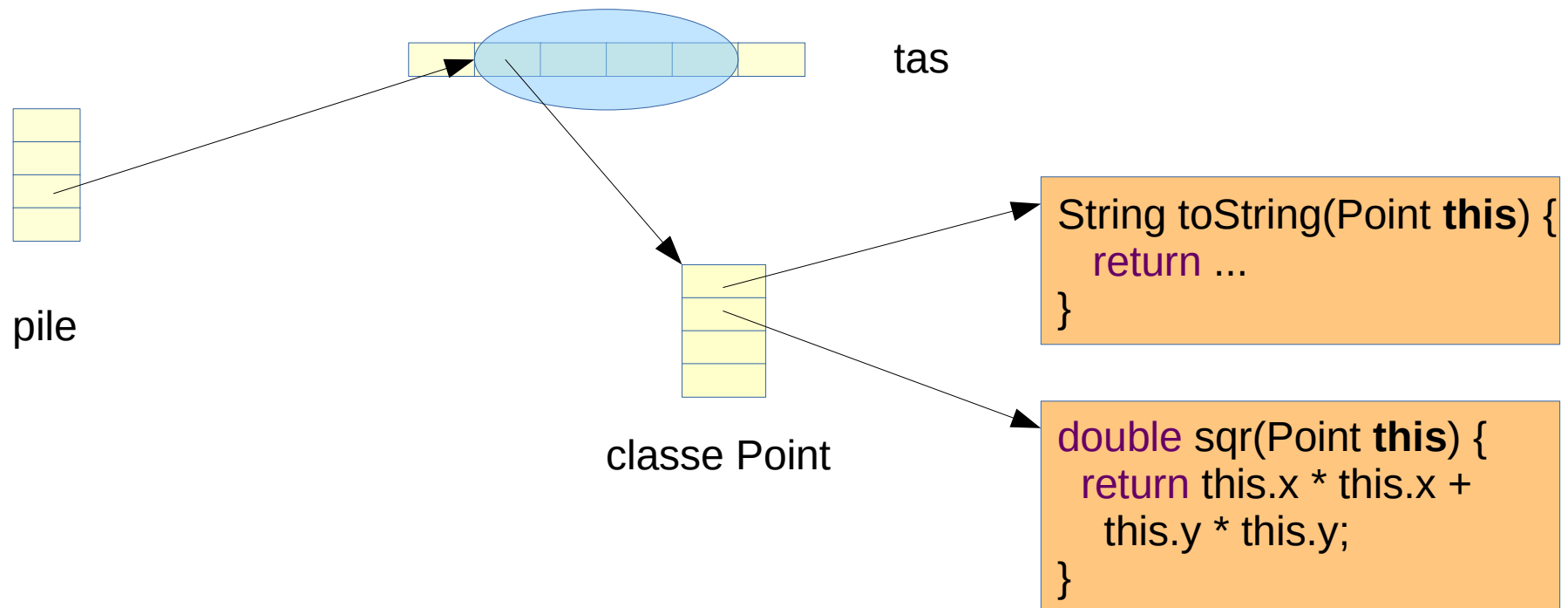
Un objet en Java correspond à une adresse mémoire non accessible dans le langage\*



\* les GCs peuvent bouger les objets  
(pourvu qu'ils mettent à jour toutes les références)

# Méthode d'objet et mémoire

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



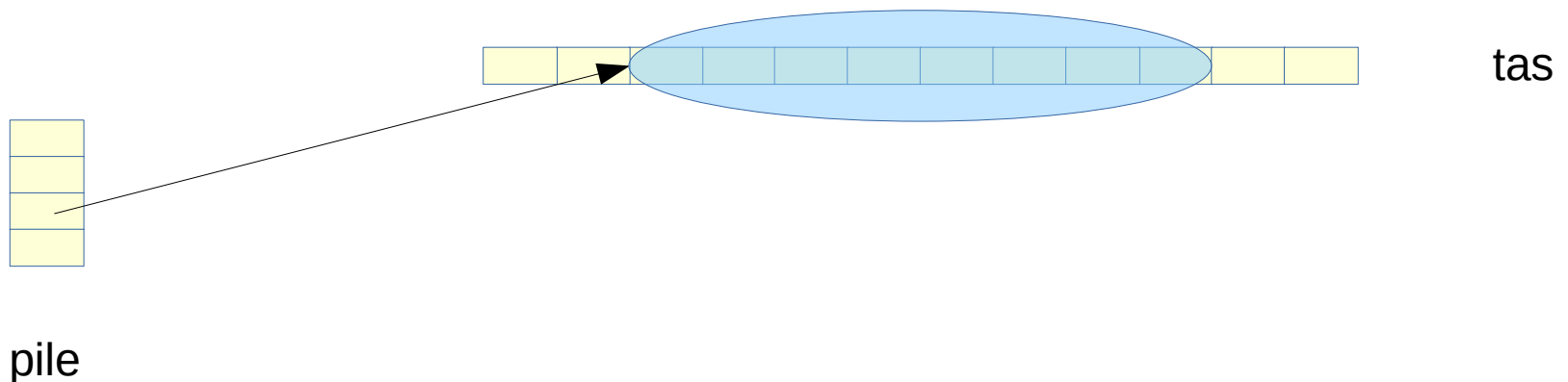
Une méthode d'instance possède un paramètre **this implicite** (que le compilateur ajoute)

# Héritage

La taille d'un objet est calculée en parcourant l'ensemble des classes parentes pour trouver les champs

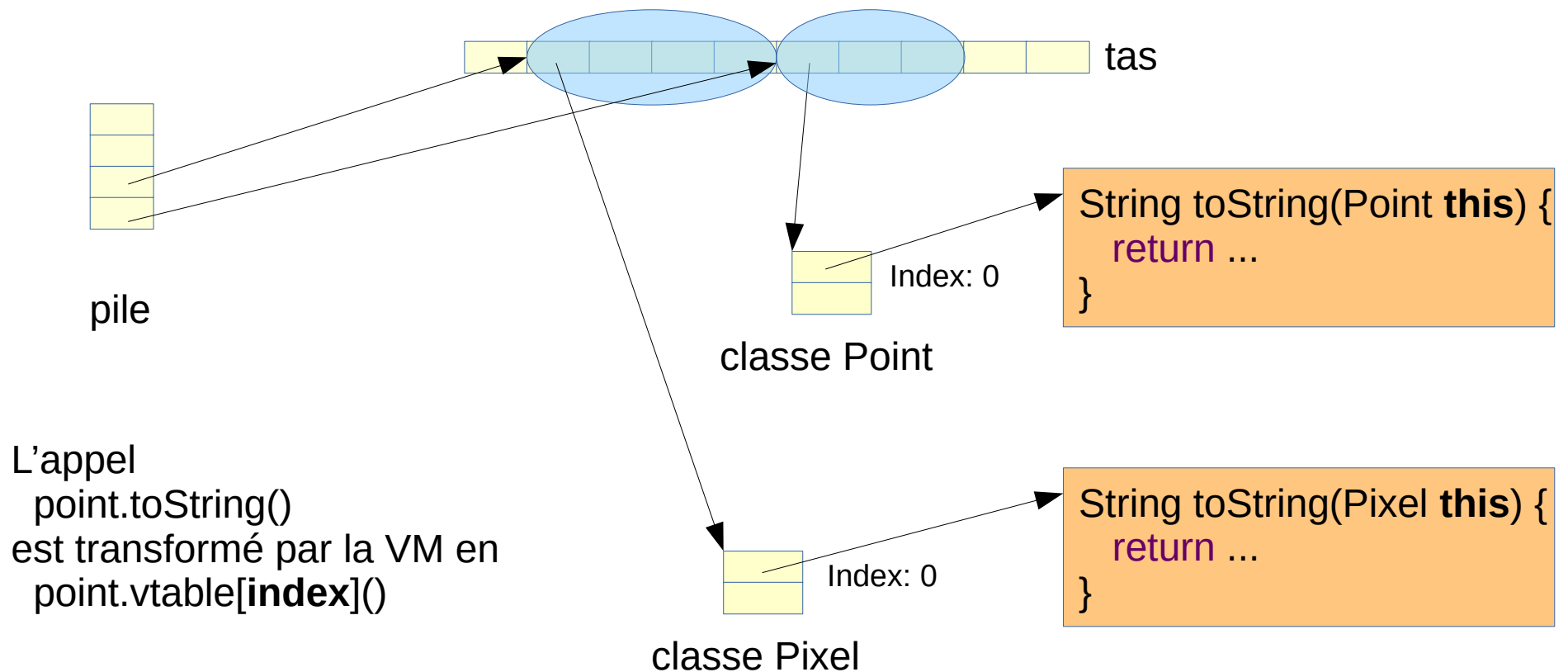
donc, c'est la même représentation en mémoire

```
class Pixel extends Point {  
    // pointeur sur la classe +  
    // hashCode + lock + GC bits = header 64bits*  
    // 12 octets de Point (x et y)  
    int color; // int => 32 bits → 4 octets  
}
```



# 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

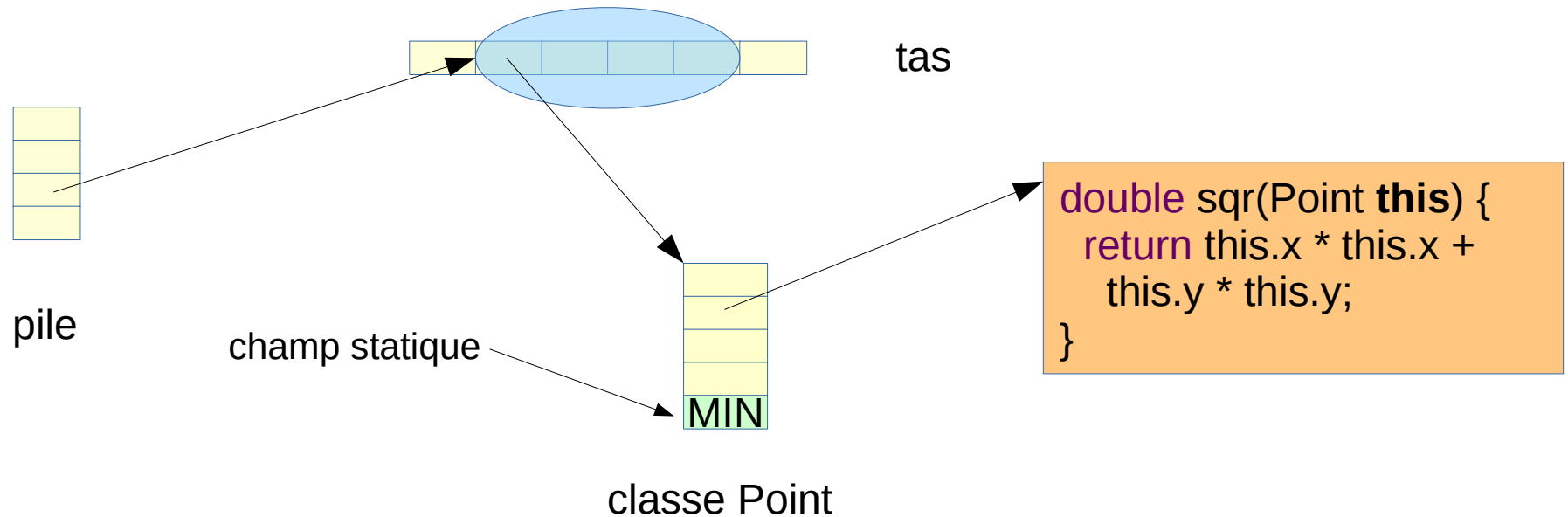
Une constante (un `#define` en C) est un champs static et final

```
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)

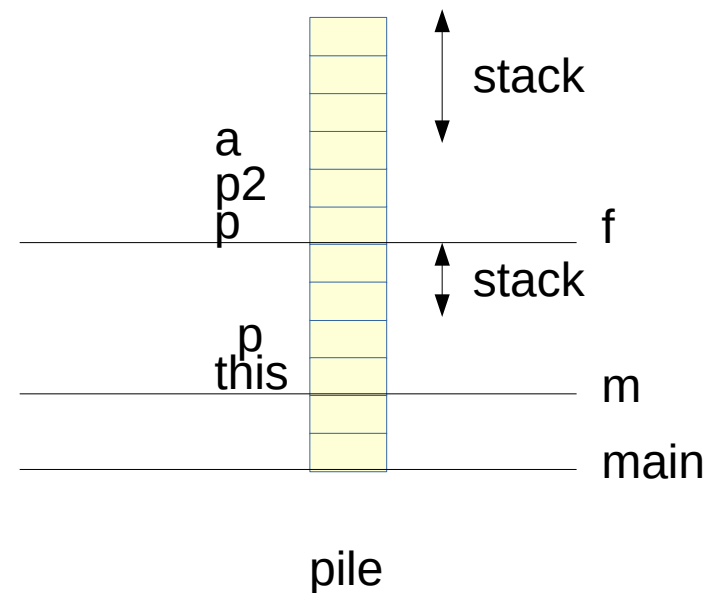
# Paramètre et Variable Locale

# Variable locale

Une variable locale (ou un paramètre) est un décalage par rapport à l'adresse de base de la zone d'activation

Le compilateur calcul la taille de la zone d'activation (taille max de la pile + taille variable locale)

```
class Foo {  
    void m(int p) {  
        f(p, 7);  
    }  
    static void f(int p, int p2) {  
        int a = 3;  
        ...  
    }  
}
```





# Variable locale vs Champ

## un champs

Le type doit être déclaré explicitement

Il est initialisé par défaut à 0, 0.0, null, false

```
class Author { String name; }
```

## une variable locale (ou un paramètre)

Le type peut être déclaré implicitement (avec **var**)

Ne peut pas être utilisé avant initialisation

```
void m() {  
    var name = "hello";  
}
```

# Variable locale vs Champ vs Tableau

Donc il y a 3 sortes de cases mémoires

Un **champ** est un décalage par rapport à l'adresse de base d'un objet dans le tas

Une **case de tableau** est un décalage par rapport à l'adresse de base d'un tableau dans le tas

Une **variable locale** est un décalage par rapport à l'adresse de base d'une activation de la méthode dans la pile