Concept de prog. en Java

Rémi Forax

Concept de prog. en Java

Classe

Primitive type vs Object

Sous-typage et Interface

Exception

Compilateur et Java Runtime

Historique (rapide)

1957-1959: Les 4 fondateurs

FORTRAN, LISP, ALGOL, COBOL

1967: Premier langage Objet

SIMULA (superset d'ALGOL): Class, héritage + GC

1970-1978: Les langages paradigmiques

C, Smalltalk, Prolog, ML, Scheme, SQL

1990-1998: 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 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 de réutilisation qui abstrait et contrôle les effets de bord

Différents styles de prog.

Chaque style de programmation n'exclue pas forcément les autres

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

- avec plus ou moins de bonheur :)

La POO en Java

Object Oriented Programming

Ensemble de principes dont le but est de developper et maintenir facilement des applications

Un objet contient

- Des données sous forme de champs
- Du code sous forme de méthodes (procédures liées à un objet)

La définition d’un objet s’appel la classe d’un objet

Un objet correspond à une instantiation de la classe

Java, Object et Valeur

Java permet de manipuler 2 sortes de valeurs

- valeur primitive (pas objet)
  - boolean, byte, short, char, int, long, float, double
- réference sur des objets (pointers dans le tas)

Pas de struct/value type comme en C, C#

Tous est pas objet comme SmallTalk, Python

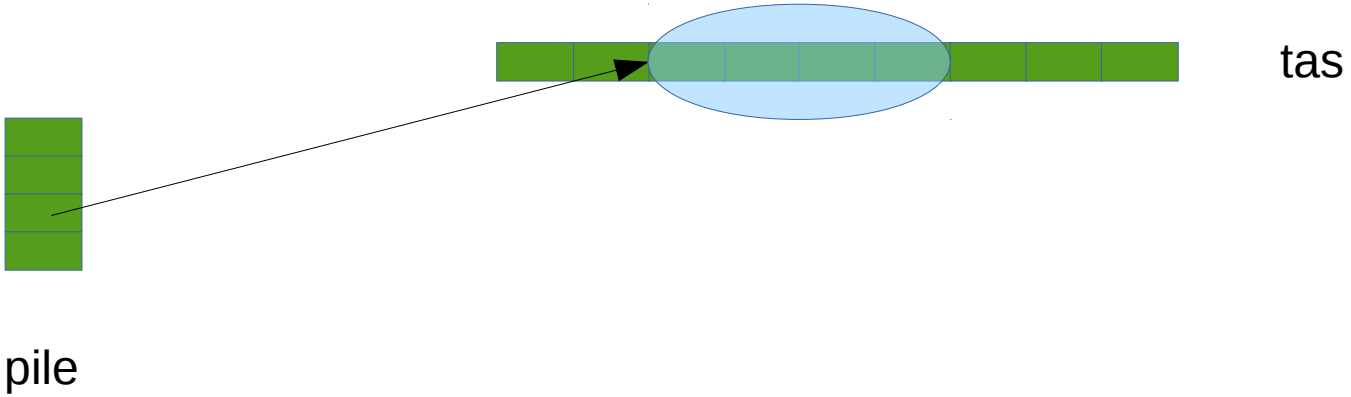
Objet et mémoire

Le contenu d’un objet est

- de taille fixe
  - On ne peut pas ajouter des champs dynamiquement comme en Python, JavaScript
- alloué dans le tas (par l’instruction **new**)
  - Pas sur la pile !
  - Accessible car 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 addresse mémoire non accessible dans le langage*



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

Classe et champs

Une classe définie des champs (ou attributs) dont le type indique comment la mémoire est structurée (formattée)

```
class Point {
  int x;
  double y;
}
```

Contrairement au C, l'ordre des champs en mémoire n'est pas garantie !

Classe et mémoire

Le type des champs permet d'avoir une idée de la taille d'un objet en mémoire

```
class Point {
  int x;          // int => 32bits => 4 octets
  double y;       // double => 64bits => 8 octet
}
```

La taille totale est au moins 12 octets
(plus gros, car header + alignement)

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

Tous les objets connaissent leur classe !

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

Classe et méthode d'instance

Une classe définie des méthodes, codes permettant de lire/écrire les champs

Une méthode d'instance est une fonction **liée** à une classe.

```
class Point {
  int x;        // int => 32bits => 4 octets
  double y; // double => 64bits => 8 octet

  double distance() { return Math.sqrt(x * x + y * y);
}
```

En Java, le code est toujours dans une méthode

Method d’instance et this

Une méthode d’instance possède un paramètre this implicite qui correspond à l’objet sur lequel on appel la méthode d’instance

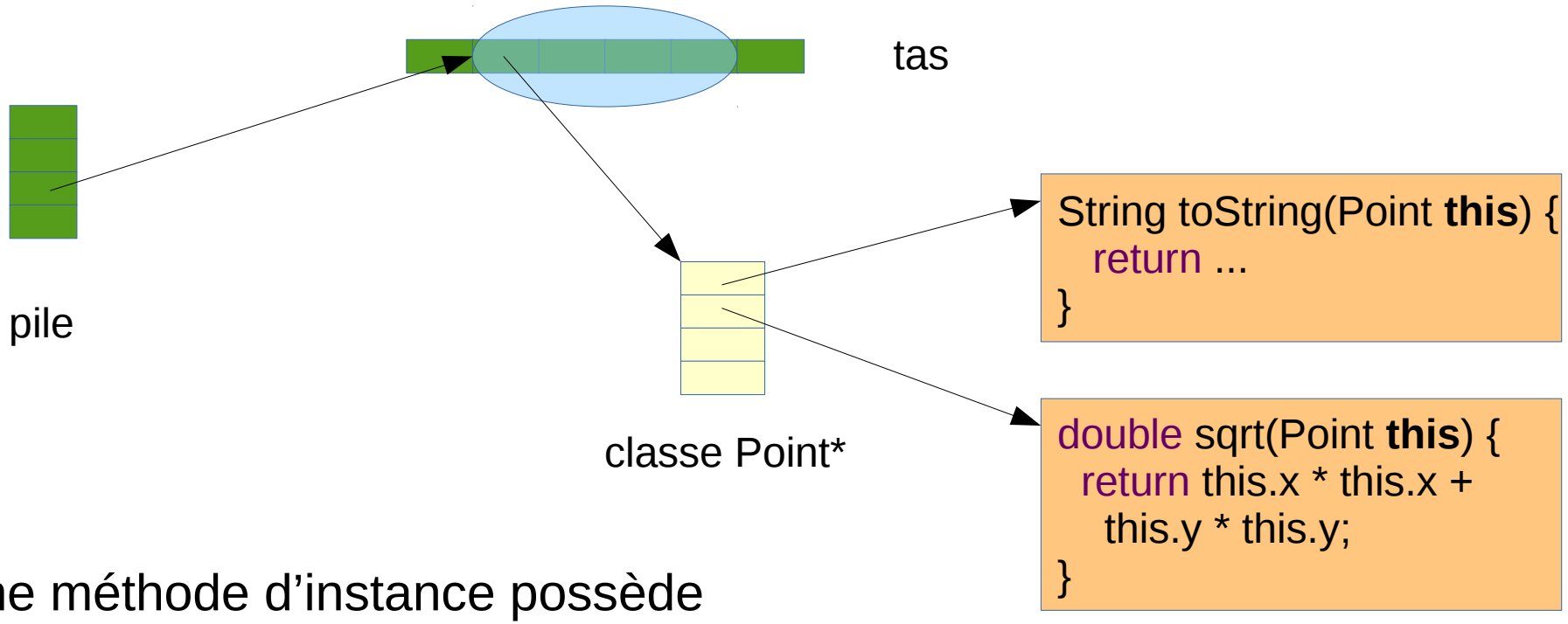
```
class Point {
  int x;        // int => 32bits => 4 octets
  double y; // double => 64bits => 8 octet

  double distance() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }


… point = …
point.distance();
```

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)

* suivant les VMs, la classe et les codes sont stockés dans le tas ou pas

Méthode statique

Une méthode statique est une méthode qui n'a pas de this implicite

Elle est appelée directement comme une fonction en C

Elle ne peut pas accéder aux champs non statiques

```
class BankAccount {
  long balance;

  void withdraw(long amount) {
    requirePositive(amount);
    ...
  }

  static boolean requirePositive(long value) {
     if (value < 0) { /* BOOM ! */ }
  }
}
```

Classe et champ statique

Il est possible de définir un champs lié à la **classe** et pas a un objet (une instance)

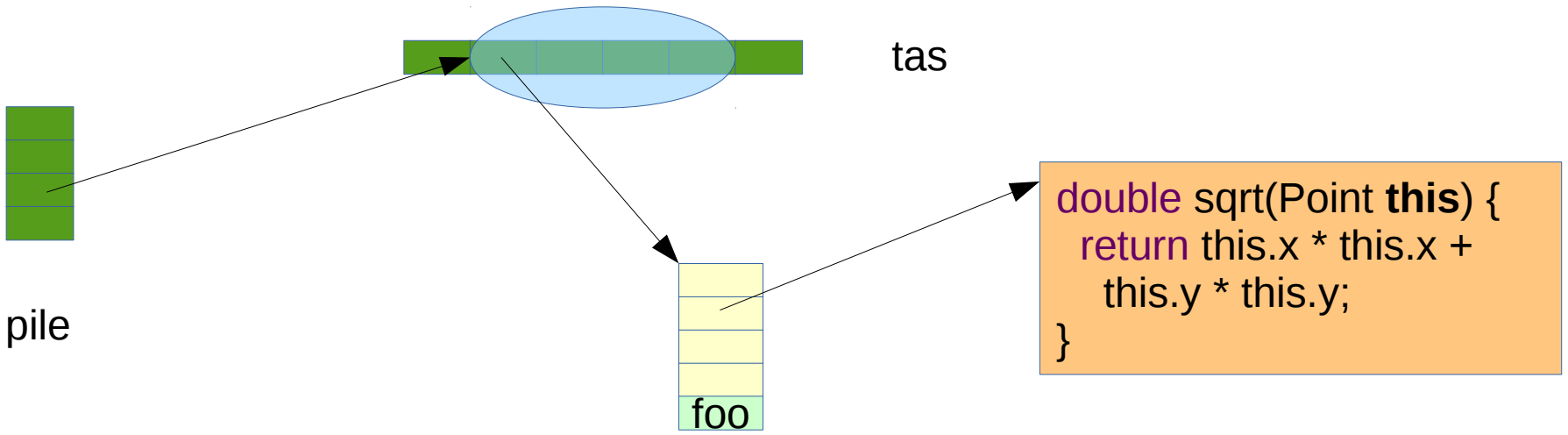
```
class Point {
  int x;
  double y;

  static int foo;
}
```

La valeur est alors **globale** (ahhh) mais rangée dans une classe

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

Champ statique et constante

Une constante, l'équivalent d'un #define en C, est un champs static et final

```
class Color {
  int red;

  Color(int red) {
    if (red < MIN || red > MAX) { /* BOOM !*/ }
    this.red = red;
  }

  static final int MIN = 0;
  static final int MAX = 255;
}
```

final veut dire initializable une seul fois

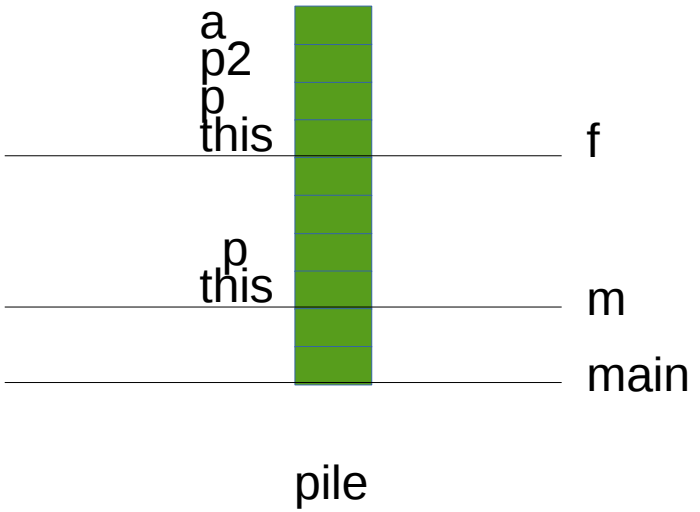
Variable locale vs Champ

Variable locale

Une variable locale (où 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);
  }
  void f(int p, int p2) {
    int a = 3;
  }
}
```



Variable locale vs Champs

Un champ est un décalage par rapport à l’adresse de base d’un objet dans le tas
(visible par tous le monde)

Une variable locale est un décalage par rapport à l’adresse de base d’une activation de la méthode dans la pile
(non visible)

En Java, Variable locale vs Champ

un champs

- Le type doit être déclaré explicitement
- est intialisé par défaut à 0, 0.0, null, false

une variable locale

- Le type peut être déclaré implicitement (avec var name = ...)
- ne peut pas être utilisé avant intialisation

```
class Utils {
  int field;   // champ initialisé à 0 par défaut

  static int randomValue() {
    var random = new Random();  // variable locale de type Random
    return random.nextInt(field);
  }
}
```

Encapsulation et Prog par contrat

Mutations

Une mutation est le fait de pouvoir changer la valeur d’une variable/d’un champ **après** initialisation

Moins il y a de mutations plus un programme
est facile à maintenir, à débugger

Mutations – 2 approches

Moins il y a de mutations plus un programme est facile à maintenir, à débugger

- Les langages dit pure interdisent les mutations
  - Actuellement, les langages “mainstream” pures sont aussi fonctionnels (Haskell, Clojure)
- Les langages objets contrôle les mutations
  - On choisi si une classe est mutable ou non
  - N’importe qui ne peut pas muter n’importe quoi

Exemple de classe non-mutable

En Java, final sur les champs et la classe permet de définir une classe non-mutable

```
final class Point {
  final int x;
  final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Principes d’encapsulation

“La seule façon de modifier l’état d’un objets est de passer par une méthode de la classe de celui-ci”

Permet de contrôler

- quel code fait des mutations ?
- que les mutations sont faites correctement

Principes d’encapsulation

En Java, on utilise des modificateurs de visibilité pour garantir l’encapsulation

- les champs sont déclarés privées
- les méthodes d’accès sont déclarées public

mais ce n’est pas suffisant !

Exemple d’encapsulation

La seule façon de modifier x et y est d’appeler translate sur Point

```
class Point {
  private int x;
  private int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void translate(int dx, int dy) {
    x += dx;
    y += dy;
  }
}
```

Utiliser des classes mutables est dangereux !

Utiliser des objets mutables est dangereux

Si Point est mutable, on casser l'encapsulation sans faire attention !

```
class Circle {
  private final Point center;

  public Point(Point center) {
    this.center = center;  // oups le code est FAUX !
  }
}

var point = new Point(1, 2);
var circle = new Circle(point);
point.translate(2, 3);  // oups, on a casser l'encapsulation
```

Utiliser des objets mutables est dangereux !

Point est mutable, il faut faire une une copie défensive

```
class Circle {
  private final Point center;

  public Point(Point center) {
    this.center = new Point(center.x, center.y);  // astuce !
  }
}

var point = new Point(1, 2);
var circle = new Circle(point);
point.translate(2, 3);  // ok !
```

Getter et Setter

Les getters et setters **sont dangereux** !!

- Les getter/setters montre trop comment les choses sont implantés
- Un getter peut publier l'état interne d'un objet
- Un setter peut permettre de changer l'état interne d'un objet de façon non contrôler

À ne pas utiliser !!!

Exemple de getter problèmatique

La seule façon de modifier x et y est d'appeler translate sur Point

```
class Circle {
  private final Point center;

  public Point(Point center) {
    this.center = center;
  }

  public Center getCenter() {
    return center;
  }
}

var circle = new Circle(new Point(1, 2));
circle.getCenter().translate(2, 3);  // oups, on a cassé l'encapsulation
```

Rêgles d’utilisation
Mutable vs Non mutable

Mutable vs Immutable

On utilise majoritairement des objets non-mutables

On peut utiliser des objets non mutable si on les stock pas dans des champs

- Utilisé un StringBuilder dans une méthode est Ok !

Si on stock un objet mutable

- Pas de getter/setter
- Copie défensive obligatoire dans le constructeur

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

Invariant, Pré-condition, Post-condition

Une classe doit définit un contrat indiquant les

- Invariants
  Relations toujours vrai lié à l'état de l'objet

Une méthode publique doit définir un contrat indiquant quelle service elle fournie à l'utilisateur ainsi que les

- Pré-conditions
  condition que les arguments doivent vérifier +
  condition que l'état de l'objet doit vérifier
- Post-conditions
  vérification de l'état d'un l'objet après l'exécution de la méthode

En Java

Pré-conditions sur les arguments

- NullPointerException, IllegalArgumentException si l'argument n'est pas valide
- IllegalStateException, si l'état de l'objet ne permet pas d'exécution la méthode

Les préconditions sont documenté dans la javadoc !!

Les post-conditions et invariant

- Le mot clé assert

Ils sont documentés dans le code

Exemple de prog par contrat

```
class Book {
  private final String author;
  private final long price;

  public Book(String author, long price) {
    this.author = Objects.requireNonNull(author);
    this.price = requiresInBetween(price, 0, Long.MAX_VALUE);
  }

  public long price(long discount) {
    requiresInBetween(discount, 0, price);
    return price – discount;
  }
}

var book = new Book("dan brown", 40);
book.price(50);  // signale à l'utilisateur le problème par une exception
```

Design by contract vs Unit Testing

Prog. par contrat != Test Unitaire

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

Un test unitaires test le code de façon externe

(soit l’API public, soit l’API privée)

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++

- La visibilité de package n'est pas nécessaire car les classes internes voient les membres privées
  - Mais avant Java 11 l'accès privé générait un bridge (cf plus tard)
- Protected rend le champs visible pour les sous-classes, ce qui viole le principe d'encapsulation

Les poupées russes

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

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

Un package se définie 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

Primitive vs Objet

Type primitif vs Object

On Java, les types primitif et les types objets sont séparés et il n’est 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

L’impact majeur sur le design du langage :(

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[]

- Donc 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, on doit vérifier 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 champs 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 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

L’identitée d’un wrapper est définie de façon bizarre

- Pour les entiers, pour les valeurs de -128 à 127 on obtient la même référence
- Pour les flottants ou les autres valeurs, on sait pas :(

```
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

Sous-typage

Classe vs Type

Une classe définie l'ensemble des opérations que l'on peut effectuer sur une valeur à l'exécution

Un type définie l'ensemble des opérations que l'on peut effectuer sur une variable à la compilation

```
URL url = new URL("http://…");
```

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) :

Sous-typage

Corollaire du principe de Liskov:
pour pouvoir ré-utiliser/partager du code, on définie un type abstrait commun à plusieurs classes que l’on pas en paramètre

Interface

Façon de définir un type abstrait en Java

Une interface définie

- Des méthodes d’instance abstraites
- Des méthodes d’instance concrètes (méthode par défaut)
- Des méthodes statiques

Une classe peut implanter plusieurs interfaces !

Une classe doit implanter l’union des méthodes abstraites de l’ensemble des ses interfaces (vérifier par le compilateur et la VM)

Exemple d’interface

Une interface ne possède pas de champs (non statique)

```
public interface StringBeautifier {
  String beautify(String s);   // abstract et public par défaut

  default boolean canBeBeautified(String s) {  // public par défaut
    return true;
  }
}

public class DefaultStringBeautifier implements StringBeautifier {
  @Override
  String beautify(String s) {
    return s.chars().mapToObj(c → "" + c).collect(Collectors.joining("*"));
  }
}
```

Méthode par défaut

Une méthode par défaut permet d’ajouter un code qui sera utiliser si la classe qui implante l’interface ne fourni pas elle même le code

Si il y a plusieurs méthodes à choisir, la méthode par défaut du sous-type est choisie

Il n’est pas possible de fournir d’mplantation des méthodes de java.lang.Object par défaut.

Interface Fonctionnel

Une interface avec une seule méthode abstraite est une interface fonctionnel

Une interface fonctionnel permet de typer une lambda ou une méthode référence

- StringBeautifier sb = s → s.toUpperCase();
- StringBeautifier sb = String::toUpperCase;

@FunctionalInterface demande au compilateur de vérifier qu'il n'y a qu'une seul méthode abstraite

Maximiser le partage de code

Une méthode publique doit préférer prendre en paramètre un type abstrait à un type concret pour aider à la réutilisation

- Par ex. prendre une java.util.List en paramètre au lieu de java.util.ArrayList

Classe vs Interface dans l'API

Quand utiliser une classe ou une interface ?

```
public class Group {
  private final ArrayList<Person> persons;
  …
```

visible de l'extérieur

Pas visible de l'extérieur

```
  public List<Person> findAllByName(String name) {
    ArrayList<Person> list = …  // ou utiliser var
    …
    return list;
  }
}
```

Pas visible de l'extérieur

Le type de retour doit être une interface pour permettre des modifications de l'implantation de la méthode à postériori

Kindom of Nouns

“In the Kingdom of Javaland, where King Java rules with a silicon fist, people aren't allowed to think the way you and I do. In Javaland, you see, nouns are very important, by order of the King himself. Nouns are the most important citizens in the Kingdom.”

Steve Yegge

Nommage en POO

Une méthode est nommée par un verbe d'action, une classe est nommée par un nom

- Si le programme a un trop gros ratios de noms, il devient incompréhensible

Un nom comme DefaultStringBeautifier apporte rien par rapport à StringBeautifier

- Il faut nommer uniquement les choses importantes !

Raffinement dans le nommage

Nommé une implantation n’est pas nécessaire si elle n’apporte rien par rapport à une interface

Il existe 3 façons d’implanter une interface (par ordre de coût cognitif)

- Une lambda (méthode anonyme)
- Une classe anonyme
- Une classe nommée (pas forcément public)

Préférer les interfaces fonctionnels !

Nommage coté utilisation

import static permet de ne pas nommé une classe lorsque l'on appel une méthode statique

```
import static java.util.stream.Collectors.joining;
…
s.chars().mapToObj(c → "" + c).collect(joining("*"))
```

var permet de ne pas spécifier le nom du type d'une variable locale

```
var builder = new StringBuilder();
```

Reprenons StringBeautifier

DefaultStringBeautifier n’a pas besoin d’être nommée et peut être implanté sous forme de lambda.

```
@FunctionalInterface
public interface StringBeautifier {
  String beautify(String s);

  default boolean canBeBeautified(String s) {
    return true;
  }

  static StringBeautifier create() {
    return s → s.chars().mapToObj(c →"" + c).collect(joining("*"));
  }
}
```

Attention !

On ne créé pas d’interface si on ne veut pas voir deux classes de la même façon

- On déclare une interface commune à Carré et Rectangle **car** mon programme affiche indifférement des carrés et des rectangles

On essaye pas de mettre l’ensemble de toutes les classes du programme dans la même hierarchie basée sur une interface !

Héritage

Héritage

Permet de définir une implantation en indiquant ce qui change par rapport à la superclass

- Il faut ré-implanter toutes les méthodes nécessaire
  sinon on casse le contrat

Pas maintenable car cela veut dire que la superclass ne peut pas changer

- Bonne idée en 1980, moins maintenant
- Les langages créé depuis 2010 (Rust ou Go) ne supporte pas l'héritage

Casse le principe d'encapsulation

Si il y a héritage, il est assez fréquent de voir des champs déclarés protected car les sous-classes en ont besoin

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 faut, on peut utliser addAll pour ajouter des étudiants (viole le contrat)

```
public final class StudentList {
  private final ArrayList<Student> students = new ArrayList<>();

  public void add(Student student) {
    Objects.requireNonNull(student);
    students.add(student);
  }
}
```

Classe Abstraite

Peut-on utiliser une classe abstraite ?

- Jamais à la place d'une interface.
- Dans ce cas, on contrôle la super-classe et la sous-classe, donc l'héritage est moins embétant.
- Une classe abstraite ne doit **pas** être **publique** car cela revient à publier les détails d'implantation.

Une classe abstraite sert à partager du code

- Une méthode sert aussi à partager du code donc il n'est pas forcément nécessaire de créer une classe abstraite
- Il faut partager des champs et du code (assez important) pour que cela soit intéressant d'utiliser une classe abstraite.

Exception

Exception

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

Comme 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) et celle vérifiée par le compilateur (Exception)

En Java:

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

En Java avec une checked exception:

```
String readLine() throws IOException {
  … lit la ligne ou lève une exception
}
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, les méthodes de l'interface doivent aussi déclarer l'exception, ce qui n'est pas possible si l'interface existe déjà !

En pratique,

On ne créé pas d'exception checkée (qui hérite d'Exception), mais on hérite de RuntimeException

On met les éxceptions checkées dans des exceptions non checkées

```
String readLine() {  // pas throws ici
  try {
    // appel qui peut lever IOException
  } catch(IOException e) {
    throw new UncheckedIOException(e);
  }
}
```

RuntimeException

Mécanisme qui permet d’éviter d’utiliser le type de retour pour signaler une erreur

- Ne doit être utiliser que si il y a une erreur
  - Integer.parseInt() ne devrait pas utiliser d’exception

- try/catch ne doit servir que si on reprend sur l’erreur
  - Logger l’erreur (printStackTrace ou autre) n’est pas reprendre sur l’erreur

Compilateur et Java Runtime

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)

en Java, la machine virtuelle transforme le code en assembleur (JIT) si le code est interpreté suffisamment souvent

Avantage et Inconvénient

Pour que le C soit portable

- il n’est pas possible d’optimiser le code pour un CPU spécifique
- La compilation peut prendre plusiuers 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 l’optimisation à lieu à l’exécution

Versions notables de Java

Java 1.0 (1995)

Java 1.2 (Collections)

Java 1.5 (Generics)

Java 1.8 (Lambda)

Java 11 (Module)

Open Source depuis Java 1.7

- Les sources sont sur openjdk.java.net
- Actuellement, une nouvelle version tous les 6 mois

OpenJDK vs Java

OpenJDK contient les sources

Java est un ensemble de distributions

- Oracle Java
- RedHat Java
- Azul Java
- SAP Java

Une distribution doit passer le Test Compatibility Kit 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 pas le seul langage qui fonctionne sur la plateforme Java

Groovy, Scala, Kotlin ou Clojure 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 blocs d’initialisations
- Les types paramétrées
- Les classes internes et les enums

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