Programmation avancée en Java

Licence Informatique

2024 – 2025

amU Faculté des sciences Aix Marseille Université

Partie 1

Rappels

Organisation du cours

Volume horaire :

- 10 séances de cours d'1h30 (15h au total)
- 11 séances de TP de 4h (44h au total)

Évaluation :

- un examen terminal en janvier (12 points)
- contrôle continu, dont partiel (4 points)
- travaux pratiques (4 points)

Objectifs du cours

- Perfectionner ses compétences en Java.
- Apprendre à concevoir avec des objets :
  - décomposer un projet en classes,
  - choisir les propriétés et méthodes des classes,
  - concevoir les interactions entre les classes.
- Apprendre à programmer proprement, à utiliser des outils de développement professionnels.

Déroulement des cours

Fonctionnement en classe inversée :

- on prépare le cours en lisant les transparents avant le cours, et on répond au test d'avant cours,
- en cours, on discute des points difficiles dans le cours, sous la forme de quizz ou d'exercices,
- en TP, vous commencerez par une série d'exercices indépendants (entre 1h et 2h), puis vous travaillerez sur un problème plus long.

Ne fonctionne que si vous jouez le jeu !

Structures des programmes

Qu'est-ce que programmer ?

1. Analyser les besoins
2. Spécifier les comportements du programme
3. Choisir et éventuellement concevoir les solutions techniques
4. Implémenter le programme
5. Vérifier que le programme a le comportement spécifié (tester)
6. Déployer le programme dans son environnement
7. Maintenir le programme (corriger les bugs, ajouter des fonctionnalités)

Programmation orienté objet

- Décomposer le programme en unité (les objets),
- Chaque unité a une unique responsabilité,
- Les unités sont indépendantes,
- Les unités communiquent entre elles par messages (appels de méthodes)

Organisation d'un programme Java

- tâches réalisées par des objets (objet = unité agissante),
- les objets ayant les mêmes comportements sont définis par une classe (classe = comment l'objet agit),
- les interfaces permettent d'abstraire les comportements des objets (interface = quoi fait l'objet),
- la classe `Main` contient une méthode `main` qui crée les premiers objets et initie la réalisation des tâches à accomplir par le programme.

Un objet

Un objet est défini par :

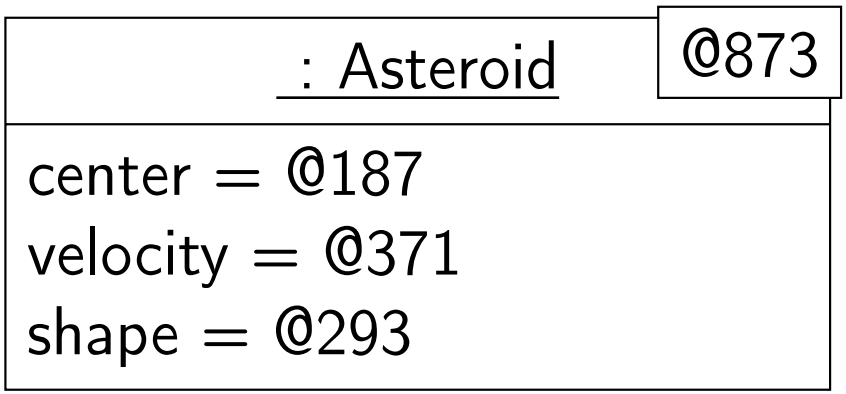
- ses méthodes (ce qu'il fait),
- ses propriétés (des valeurs internes à l'objet, qui lui servent à accomplir sa tâche).

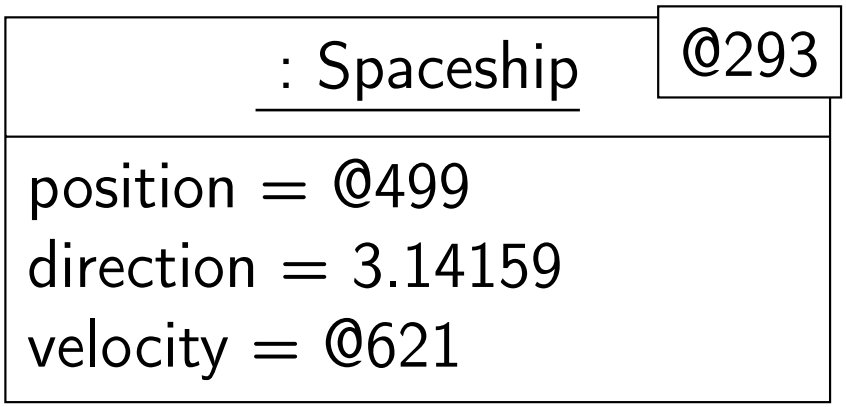
Les propriétés et méthodes peuvent être :

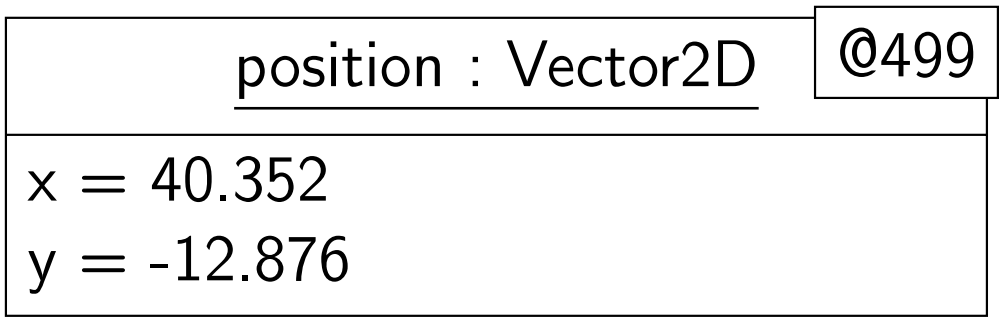
- publique (accessible à tous),
- ou privée (seulement à lui-même).

Diagrammes d'objets

Les diagrammes d'objets permettent de représenter l'état du programme à un instant de son exécution.







Une classe

Une classe définit :

- une façon de construire et initialiser un objet,
- le comportement des méthodes de cet objet,
- les propriétés de cet objet.

Remarques :

- Les classes ne sont pas nécessaires pour définir des objets, mais c'est la façon standard.
- on peut construire autant d'objets que voulus d'une même classe.

Une interface

Une interface définit :

- une liste de méthodes pouvant être réalisés par des objets.

Remarques :

- tout objet peut implémenter zéro, une ou plusieurs interfaces,
- les interfaces sont des types : elles peuvent être utilisés pour préciser la nature d'un paramètre d'une méthode, *etc.*

Relations objet vs classe vs interface

- un objet instancie une classe (et une seule),
- une classe peut implémenter une ou plusieurs interfaces,
- une interface peut étendre une autre interface

- une interface peut être implémentée par plusieurs classes,
- une classe peut être instanciée en plusieurs objets,
- une classe définie des propriétés et méthodes,
- un objet possède des propriétés et méthodes.

Un projet en Java

Un projet est constitué de :

- *packages*, qui regroupent plusieurs fichiers dans un même *espace de nom*, constitués de...
- fichiers, correspondant chacun à une classe ou une interface, et comprenant chacun...
- d'une déclaration de classe ou d'interface.

```
public class MyClassName { ... }

public interface MyInterfaceName { ... }
```

Une classe en Java

Une classe contient (dans sa déclaration) :

- des déclarations de propriétés (avec éventuellement une initialisation),
- des déclarations de méthodes et de constructeurs, chacune contenant un bloc d'instructions.

Une interface contient :

- des déclarations de méthodes (sans implémentation).

Déclaration de propriétés

Exemples sans initialisation (initialisés par défaut) :

```
private int count;
private final Color backgroundColor;
private final String name;
```

Exemples avec initialisation :

```
private int count = 0;
private final Color backColor = new Color(0.5,0.2,0);
private final String name = "John Doe";
```

Déclaration d'une méthode

```
visibilité + type de retour + nom + (paramètres) + {
  bloc d'instructions
}
```

Exemples :

```
public int getCount() { ... }

public void setCount(int newCount) { ... }

public void launchMissile(GPSCoordinate coord) { ... }

private Vector3D getAcceleration() { ... }
```

amU Faculté des sciences Aix Marseille Université

Écriture des méthodes

Corps d'une méthode

Le corps d'une méthode ou d'une classe est composé :

- d'instructions, terminées par ;,
- de structures de contrôle, permettant de décider quelles instructions exécuter.

Remarques :

- Sans structure de contrôles, les instructions sont exécutées une par une, dans le sens de lecture.
- On écrit une instruction ou une structure de contrôle par ligne.
- On utilise l'indentation pour mettre en valeur les structures de contrôle.

Composants d'une méthode

Il nous faut donc savoir :

- utiliser les 5 (principales) instructions disponibles,
- écrire des expressions servant de paramètres pour ces expressions,
- contrôler l'exécution des instructions selon des conditions (exprimées par des expressions booléennes) avec les structures conditionnelles et les boucles.

Nous détaillons tous ces concepts dans les prochains transparents.

Les instructions : création d'une variable

Une variable est un espace mémoire nommé pour stocker une valeur momentanément.
Notion de portée : La variable existe

- depuis le moment où l'instruction de création est exécutée,
- jusqu'à la sortie du bloc ({ ... }) où elle est définie.

La variable possède un type (primitif, classe ou interface), et ne peut contenir que des valeurs de ce type.

```
int count;
Vector2D position;
```

Les instructions : l'affectation

L'affectation est l'opération consistant à mettre une valeur dans un espace mémoire définit par la *l-value* (left-value, le terme de gauche dans l'affectation), généralement une variable.

L'affectation est soumise aux contraintes de types : la valeur stockée doit être d'un type compatible avec celui déclaré.

```
count = 12;
count = count + 4;
position = new Vector(3, -2.5);
```

La *r-value* (le terme de droite) doit être une expression.

Les left-values

La left-value d'une affectation peut être :

- une variable de la méthode, dont la portée englobe cette instruction,
- un paramètre de la méthode,
- une variable d'objet = une propriété de l'objet,
  - de l'objet exécutant cette méthode : `this.someVar`,
  - d'un autre objet : `someObject.someVar`,
- une variable de classe = propriété `static`, avec `SomeClass.someVar`,
- la cellule d'une tableau : `someArray[i]`.

Les instructions : la construction

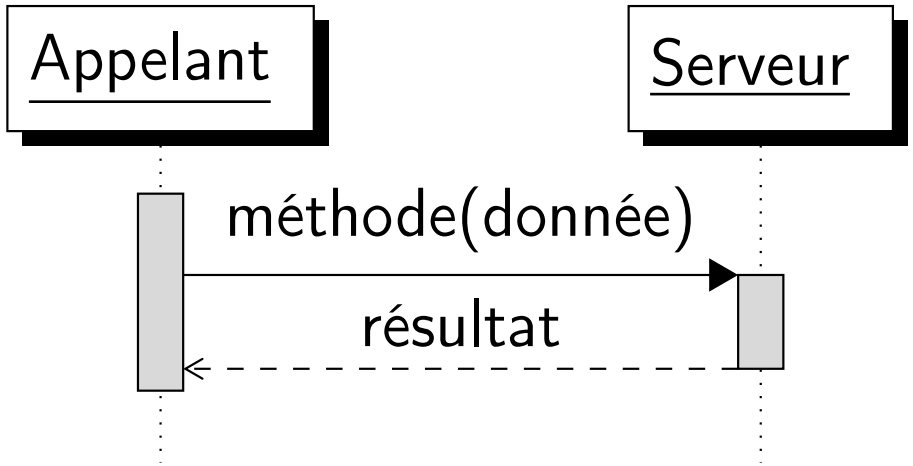
La création d'un nouvel objet se fait avec le mot-clé `new`, qui déclenche :

1. l'allocation d'un espace mémoire pour contenir le nouvel objet,
2. l'initialisation des propriétés de cet objet,
3. l'appel au constructeur de l'objet,
4. l'adresse mémoire de l'objet est retournée par l'instruction.

```
new Vector(3,-2.5);
new Spaceship();
new ArrayList<Integer>();
```

Les instructions : l'appel de méthode

L'appel de méthode permet d'envoyer un message à un autre objet, déclenchant les instructions de la méthode de cet objet. On utilise les diagrammes de séquences pour représenter les appels de méthodes. Ce sont des diagrammes temporels, montrant les interactions entre les objets.




```
server.doSomething(someData);
```

Les instructions : `return`

La méthode peut retourner une valeur lorsqu'elle a terminé son travail, en utilisant le mot-clé `return`.
Lors de l'évaluation de l'instruction `return`, l'argument du `return` (le résultat) est d'abord évalué (s'il existe), ensuite l'exécution de la méthode est interrompue. Le résultat est retourné au niveau de l'instruction ayant fait l'appel de méthode.

```
return;
return 42;
return new Vector(-5,2.333);
```

Les instructions : récapitulatif

Les instructions sont :

- la création d'une variable,
- l'affectation,
- la construction d'un objet,
- l'appel d'une méthode (de type de retour `void`),
- la terminaison et le retour de la méthode.

La création d'une variable peut-être combinée avec une affectation.

```
String name = "John Doe";
Car blueCar = new Car(Color.BLUE);
```

Les expressions

Les valeurs sont définies par des expressions :

- litéral arithmétique (0, 3.14, ...) ou booléen (`true`, `false`), chaînes de caractères (entre `"..."`),
- variables et propriétés,
- appels de méthode (ayant un type de retour autre que `void`),
- instruction `new`,
- combinaison d'expressions par des opérateurs (`+`, `*`, ...), ...

Les expressions apparaissent :

- en membre droit d'une affectation (*r-value*),
- en paramètre d'un appel de méthode, d'un constructeur,
- en argument d'une instruction `return`,
- en condition d'une structure de contrôle.

Les variables d'une expression

Quelles variables sont utilisables dans une expression ?

- Les variables de la méthode, si cette expression est dans leur portée : `someVar`.
- Les paramètres de la méthode : `someParam`.
- Les variables de l'objet (= propriétés) exécutant la méthode : `this.someVar`.
- Les variables d'objet (= propriétés) d'un autre objet, du moment qu'elles sont publiques : `someObject.someVar`.
- Les variables de classe (= propriétés statiques) de la classe, `someVar`, ou d'une autre classe si elle sont publiques : `SomeClass.someVar`.

Les structures de contrôles : if

La structure conditionnelle permet d'exécuter un bloc d'instructions que si une condition est satisfaite.

```
if (people.isGoodGuy()) {
  Sout.println("Greetings, " + people.getName() + "!");
}

if (people.isGoodGuy()) {
  Sout.println("Greetings, " + people.getName() + "!");
} else {
  Sout.println("You are not welcome here, "
               + people.getName() + ".");
}
```

Organigramme de programmation

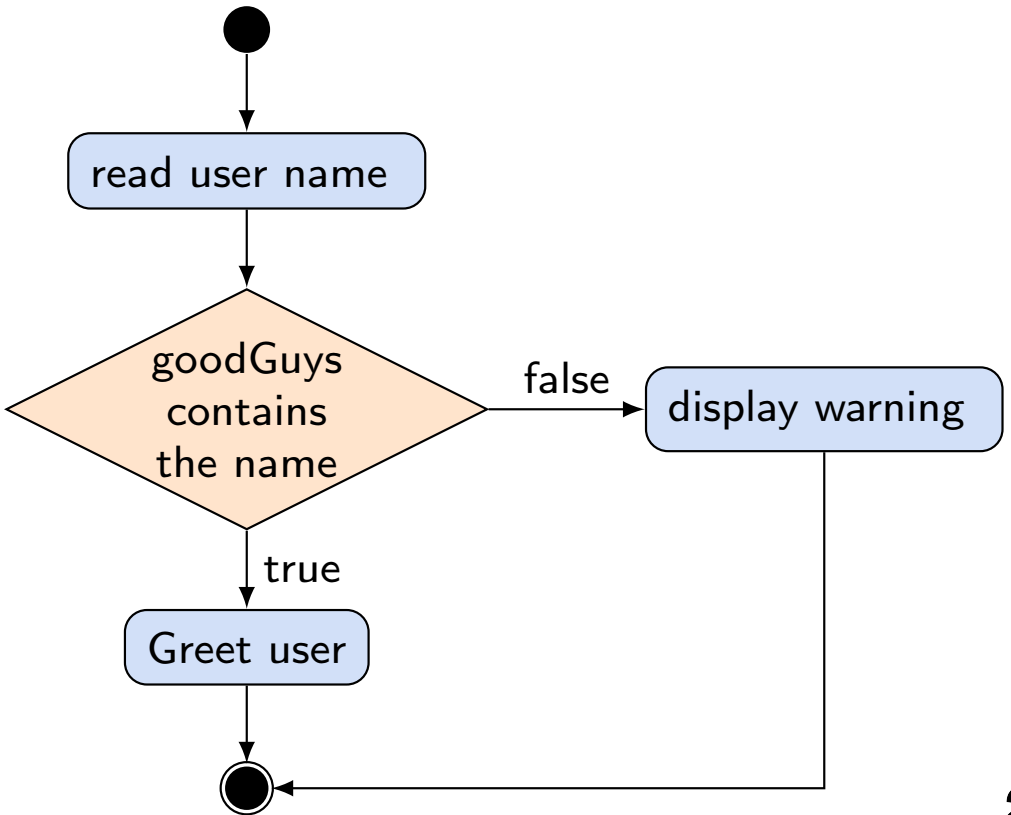
Pour représenter l'ordre d'exécution des instructions, on utilise des diagrammes appelés organigramme de programmation (*flowchart* en anglais).

- Chaque étape élémentaire est représenté par un bloc rectangulaire,
- L'ordre entre les étapes est donné en suivant les flèches,
- le disque noir indique le point de départ, les disques noirs entourées indique un point final.
- les bloc en losange indique les choix, régies par des conditions.

(version simplifiée des diagrammes d'activités en UML).

Exemple

```
String name = input.readLine();
if (goodGuys.contains(name)) {
  Sout.println("Greetings " + name + "!");
} else {
  Sout.println("You are not welcome here.");
}
```



La répétition indéfinie : `while`

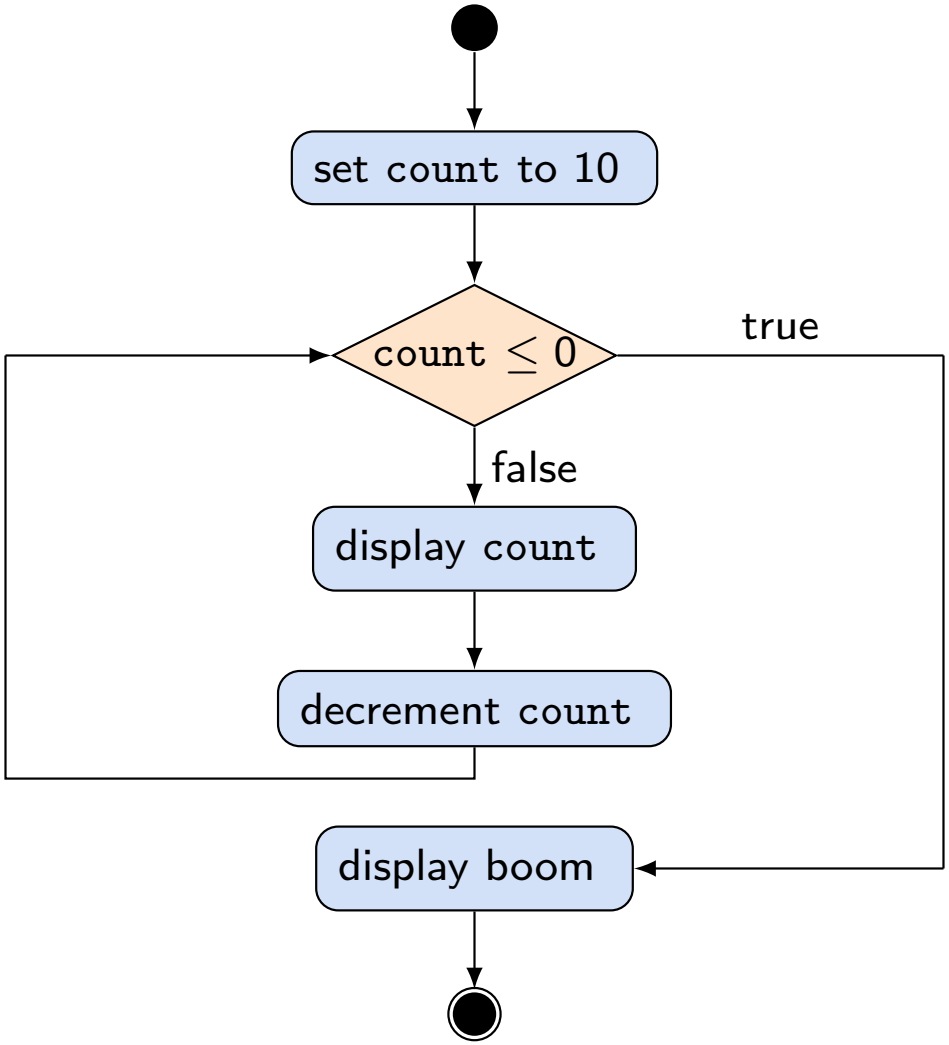
La boucle `while` permet de répéter les mêmes instructions un nombre quelconque de fois. On quitte la boucle avec l'instruction `break` ou l'instruction `return`.

```
// Le compte à rebours
int count = 10;
while (true) {
  if (count <= 0) break;
  System.out.println(count + "...");
  count = count - 1;
}
System.out.println("BOUM!!!");
```

Note : on peut avoir plusieurs instructions `break` dans la même boucle.

Organigramme : `while (true)`

Ce qui donne :



La répétition indéfinie : `while` (II)

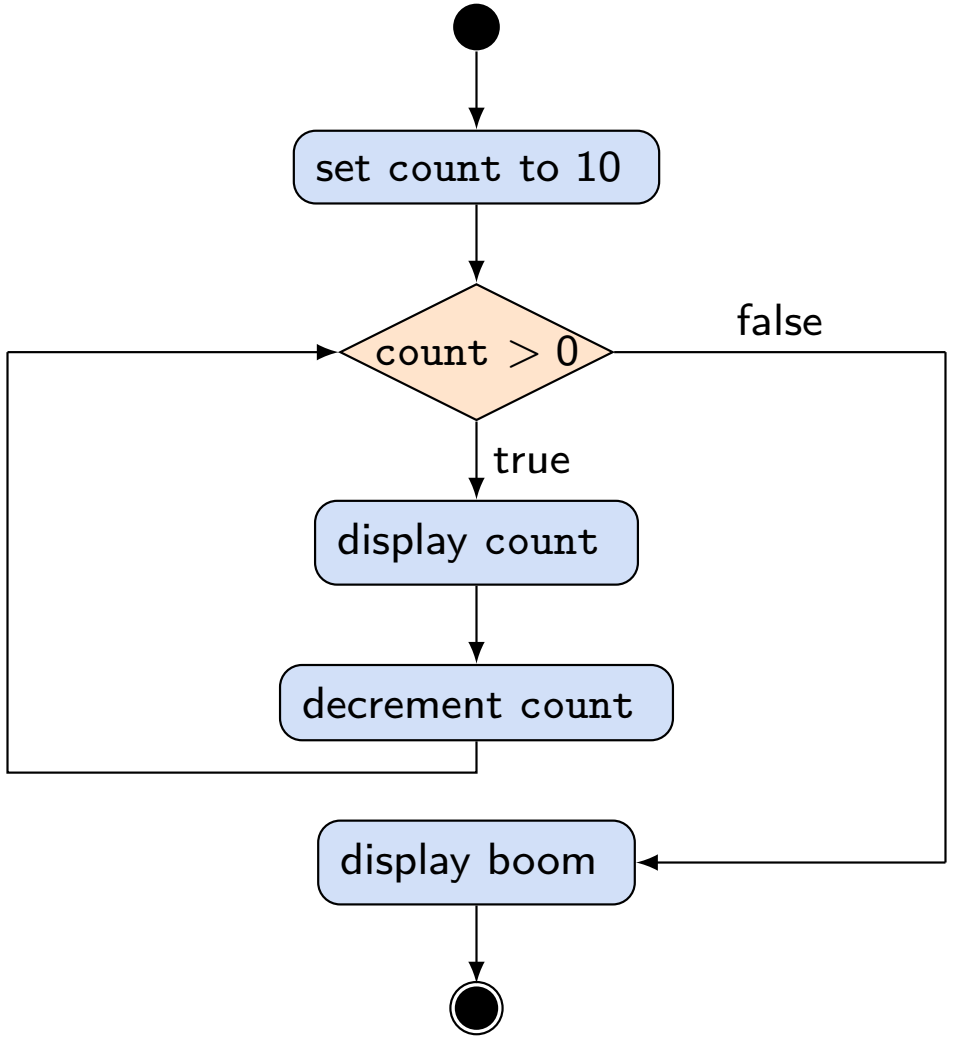
Lorsque le test de sortie est au début de la boucle `while`, on le place (inversé) directement dans la condition d'arrêt du `while` :

```
int count = 10;
while (count > 0) {
  System.out.println(count + "...");
  count = count - 1;
}
System.out.println("BOUM!!!");
```

Attention : le test n'est effectué que juste avant d'effectuer la première instruction à chaque itération !

Organigramme : `while`

Ce qui donne (notez la subtile différence) :



La répétition indéfinie : `while` (III)

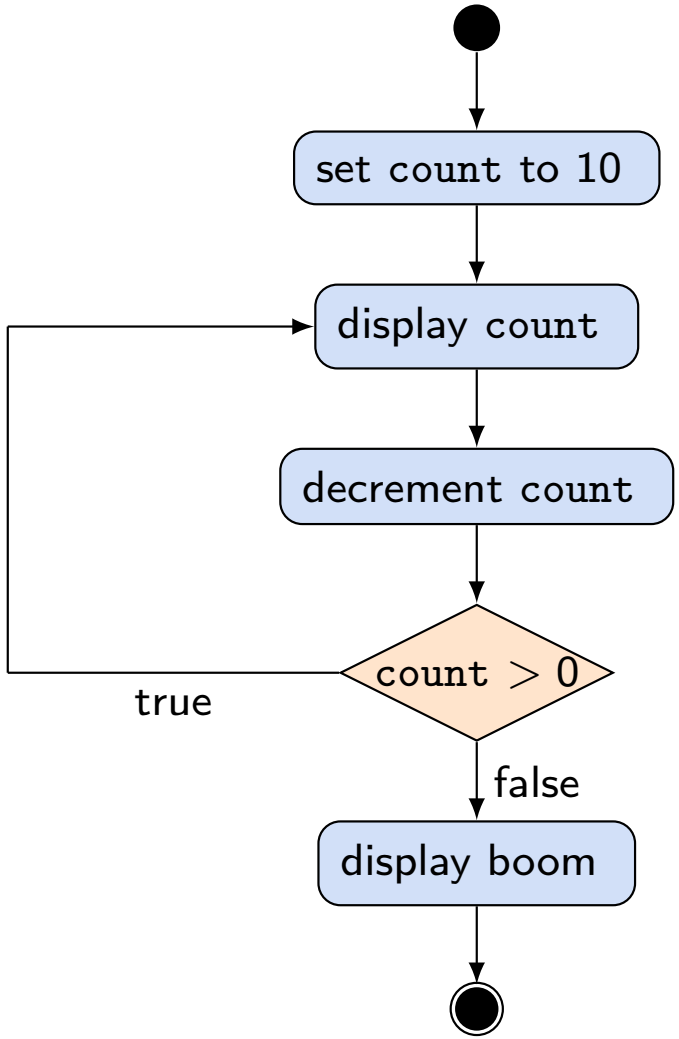
Lorsque le test de sortie est à la fin de la boucle `while`, on utilise la construction `do ... while` :

```
int count = 10;
do {
  System.out.println(count + "...");
  count = count - 1;
} while (count > 0);
System.out.println("BOUM!!!");
```

Attention : le test n'est effectué que juste après d'effectuer la dernière instruction de chaque itération !

Organigramme : do while

Ce qui donne :



Comparaison de ces 3 solutions

Exercice

Que se passe-t-il dans chaque cas si on initialise le compte à rebours à 0 ou -1 ?

En déduire que la dernière version est moins bonne que les autres.

La répétition bornée : `for`

Dans certains cas, il est plus direct d'utiliser une boucle `for`, qui isole :

- l'initialisation de la boucle,
- la condition d'arrêt,
- l'instruction de progression.

```
for (int count = 10; count > 0; count = count - 1) {
  System.out.println(count + "...");
}
System.out.println("BOUM!!!");
```

On peut utiliser `break` et `return` dans une boucle `for` aussi.

La répétition bornée : `for`

```
for (initialisation;
     condition;
     incrément)
{
  ...
}
```

=

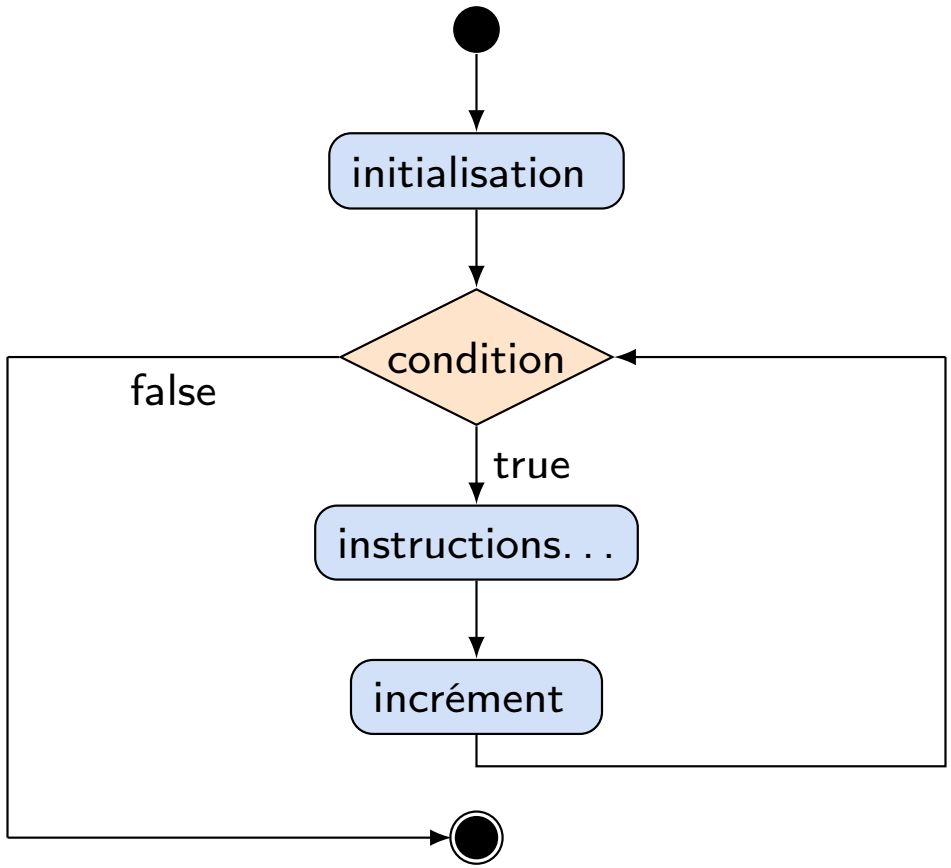
```
initialisation;
while (condition) {
  ...
  increment;
}
```

Usages typiques :

- exécuter des instructions paramétrées par un indice entier `index`, pour toutes les valeurs de `index` entre 0 et n (ou un intervalle d'entiers arbitraires),
- exécuter des instructions pour toutes les valeurs d'un tableau (*cf.* prochain cours).

Organigramme : for

```
for (initialisation; condition; incrément) {
  instructions;
}
```



L'itération : for

Pour les objets qui sont des collections, implémentant l'interface `Iterable<Elt>`, on peut utiliser la boucle `for` ainsi :

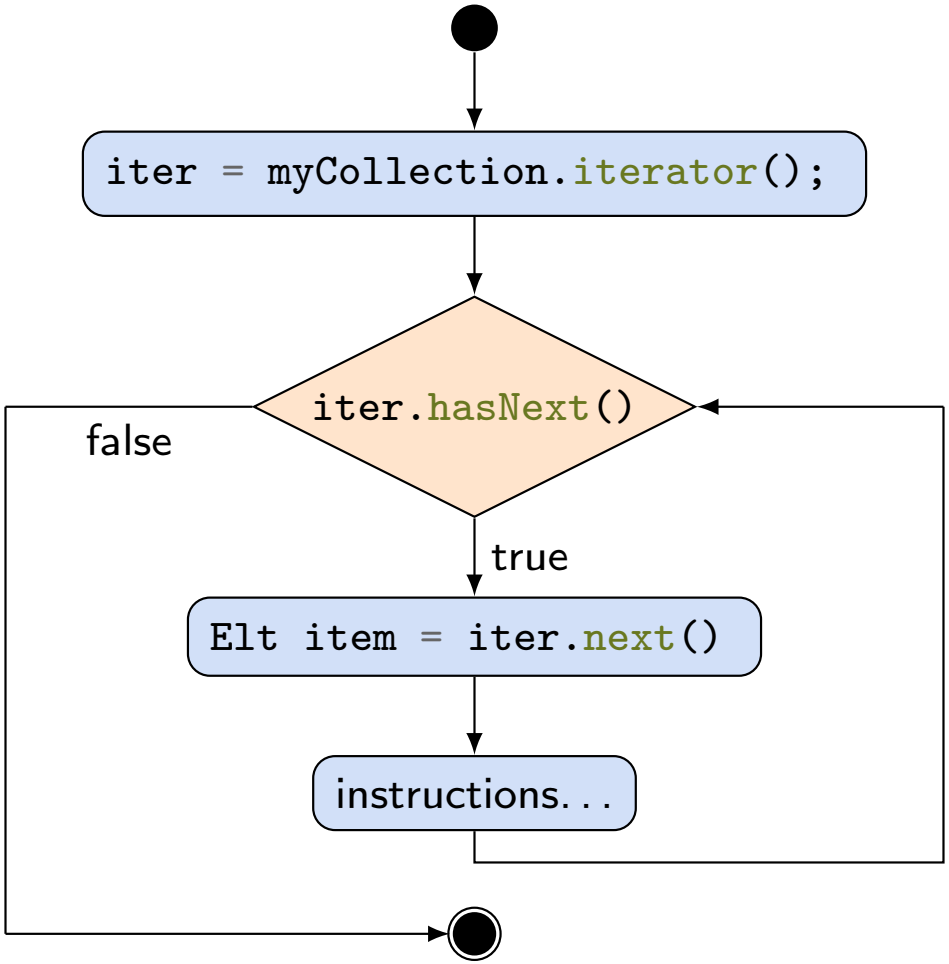
```
for (Elt item : myCollection) {
  item.doSomething();
  // ...
}
```

Abbréviation pour :

```
Iterator<Elt> iter = myCollection.iterator();
for ( ; iter.hasNext(); ) {
  Elt item = iter.next();
  item.doSomething();
}
```

Organigramme : `for` sur itérables

Ce qui donne :



Imbrication des structures de contrôle

Les structures de contrôles définissent des blocs d'instructions, entre { et } :

- possible d'utiliser d'autres structures de contrôle dans ces blocs, par imbrication,
- les blocs définissent les portées des variables, une variable ne vit que dans le bloc où elle est définie et les blocs imbriqués dedans.
- les structures de contrôle compliquent le programme : il faut éviter l'imbrication !

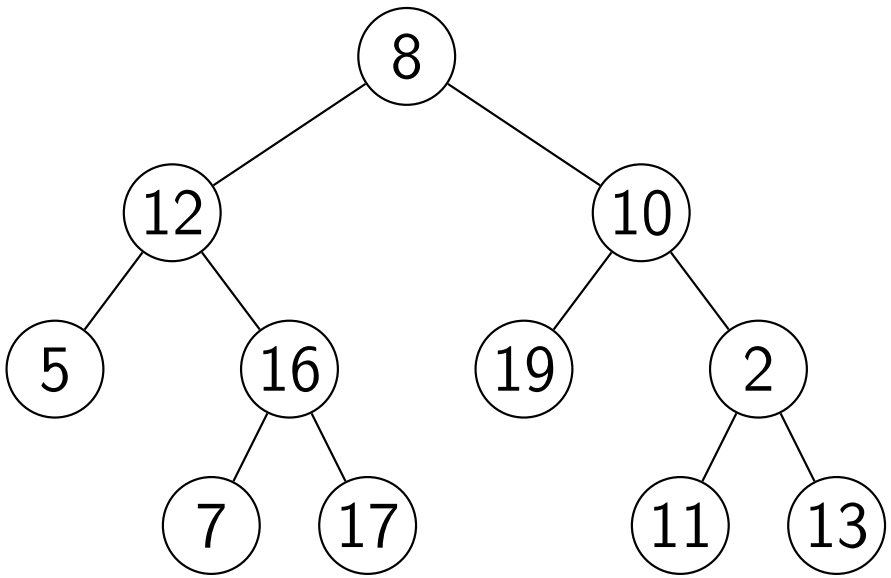
Partie 2

Réduire l'imbrication des structures de contrôle

Description d'un problème

On considère le problème suivant : étant donné un arbre enraciné, écrire les valeurs de chaque nœud, dans l'ordre de lecture, chaque niveau de l'arbre sur une ligne différente.

Exemple :



Sortie :

```
8
12  10
5  16  19  2
7  17  11  13
```

La classe des arbres

Voici la classe représentant les arbres :

```
public class Tree {
  ...

  public int getRoot() { ... }

  public List<Tree> getChildren() { ... }
}
```

Une solution en Java

L'algorithme tel que donné par ChatGPT (09/2025) :

```java
public class BreadthFirstSearch {

    public void printValues(Tree tree) {
        List<Tree> nextLevel = new ArrayList<>();
        nextLevel.add(tree);
        while (!nextLevel.isEmpty()) {
            Collection<Tree> currentLevel = nextLevel;
            nextLevel = new ArrayList<>();
            for (Tree subtree : currentLevel) {
                System.out.print(subtree.getRoot() + " ");
                for (Tree child : subtree.getChildren()) {
                    nextLevel.add(child);
                }
            }
            System.out.println();
        }
    }
}
```

Une solution en Java

Exercice

Avant de regarder la suite, demandez-vous deux minutes comment l'algorithme fonctionne.

Solution

C'est un problème classique, appelé le parcours en largeur, que vous étudierez en cours d'algorithmique.
On appelle niveau un ensemble de tous les nœuds ayant la même profondeur dans l'arbre (ce sont les nœuds qui seront écrits sur une même ligne).

1. Le premier niveau consiste en la racine seule. On commence par ce niveau.
2. tant que le niveau actuel est non-vide :
    3. écrire toutes les valeurs du niveau actuel,
    4. calculer et passer au niveau suivant, constitué de tous les enfants des nœuds du niveau actuel.

Une solution en Java

Exercice

Reconnaissez-vous l'algorithme dans l'implémentation ?

Analyse de l'exemple

L'algorithme est compliqué à lire :

- boucles imbriqués, difficile de suivre l'exécution de l'algorithme,
- trop de choses dans la même méthode,
- l'algorithme ne correspond pas directement à l'explication.

Imbrication

- L'imbrication est le fait d'utiliser des structures de contrôles (une boucle `for` ou `while`, une structure conditionnelle `if`) les unes dans les autres.
- On peut ainsi avoir une boucle dans une boucle dans une boucle...dans une méthode.
- Chaque "dans une" correspond à un degré supplémentaire d'imbrication.

Pour garder une méthode lisible, il est essentiel de ne pas avoir un degré d'imbrication élevé (idéalement 0 : pas de structure de contrôle, ou 1 : pas d'imbrication).

Comment procéder pour écrire l'algorithme

Depuis la description en langue naturelle (en français) de l'algorithme, traduire directement en syntaxe Java. Introduire des méthodes pour les parties à détailler.

1. Le premier niveau consiste en la racine seule. On commence par ce niveau.
2. tant que le niveau actuel est non-vide :
   3. écrire toutes les valeurs du niveau actuel,
   4. calculer et passer au niveau suivant, constitué de tous les enfants des nœuds du niveau actuel.

Puis recommencer pour les méthodes introduites.
Renommer variables et méthodes lorsque c'est opportun.

L'algorithme en Java simple

La méthode principale (à comparer à la description de l'algorithme).

```
public void printValues(Tree tree) {
    List<Tree> currentLevel = new ArrayList<>();
    currentLevel.add(tree);
    while (!currentLevel.isEmpty()) {
        printAllRoots(currentLevel);
        currentLevel = computeNextLevel(currentLevel);
    }
}
```

L'algorithme en Java simple

Les méthodes introduites.

```
private void printAllRoots(List<Tree> trees) {
    for (Tree tree : trees) {
        System.out.print(tree.getRoot() + " ");
    }
    System.out.println();
}

private List<Tree> computeNextLevel(List<Tree> currentLevel) {
    List<Tree> nextLevel = new ArrayList<>();
    for (Tree tree : currentLevel) {
        addAllChildren(tree,nextLevel);
    }
    return nextLevel;
}

private void addAllChildren(Tree tree, List<Tree> level) {
    for (Tree child : tree.getChildren()) {
        level.add(child);
    }
}
```

Analyse de la version simplifiée

- très proche de la description de l'algorithme,
- chaque méthode est simple à comprendre, on peut facilement vérifier qu'elles sont correctes,
- le fait d'utiliser des méthodes portant un nom détaillé améliore la lisibilité (on connait le rôle de chaque méthode),
- ce n'est pas beaucoup plus long à écrire,
- à l'exécution, les instructions réalisées sont (presque) les mêmes.

Ce qu'il faut retenir

- Commencer par une description à haut-niveau, ensuite aller vers les détails.
- Ne pas hésiter à introduire de nombreuses méthodes : cela vous force à nommer et donc à réfléchir au rôle de chaque méthode, et cela rend le programme plus facile à lire et à vérifier.
- Éviter les boucles imbriquées : elles sont difficiles à comprendre. Sortez la boucle interne dans une méthode.
- IntelliJ peut vous assister dans l'extraction de méthode.
- Suivre les principaux patrons de méthodes.

Les patrons de méthodes

Patron de méthode : itérer une opération

Faire une opération sur tout ou partie des éléments d'une collection :

```
public void iterateProcess(Collection<Elt> elements) {
  initialise();
  for (Elt element : elements) {
    if (element.isGood()) {
      process(element);
    }
  }
  return;
}
```

une boucle `for` contenant une conditionnelle `if`.

Patron de méthode : réduction

Faire un calcul agrégeant les éléments d'une collection :

```
public ReturnType aggregate(Collection<Elt> elements) {
  ReturnType value = initialise();
  for (Elt element : elements) {
    value = process(value, element);
  }
  return value;
}
```

Une boucle `for`, quelques instructions dont une affectation.

Patron de méthode : tester une collection

Tester que tous les éléments d'une liste vérifient une condition :

```
public boolean areAllGood(Collection<Elt> elements) {
  for (Elt element : elements) {
    if (!element.isGood()) {
      return false;
    }
  }
  return true;
}
```

Un `for`, un `if`, on quitte la méthode dès qu'un mauvais élément est trouvé (`return`).

Patron de méthode : boucle `while`

Répéter une opération jusqu'à la réalisation d'une condition :

```
public void repeatProcess() {
  value = initialise();
  while (isNotOver(value)) {
    instructions; // value must change!
  }
  return;
}
```

Quitter une boucle

- La condition d'une boucle est évaluée au début de la boucle (`while`, `for`) ou à sa fin (`do while`) exclusivement (revoir les organigrammes de programmation).
- Si la condition est réalisée, les instructions de la boucle sont toutes exécutées, avant que la condition ne soit à nouveau testée.
- Dans certains cas, nous souhaitons sortir de la boucle "au milieu" de ses instructions, comment faire ?
- Une possibilité est d'utiliser un `return`, mais ce n'est pas toujours suffisant.

Dans l'exemple suivant, on utilise le crible d'Ératosthène pour calculer les nombres premiers inférieurs à n.

Crible (version 1)

```java
public List<Integer> allPrimes(int max) {
    List<Integer> primes = new ArrayList<>();
    for (int n = 2; n <= max; n++) {
        boolean isPrime = true;
        for (Integer prime : primes) {
            if (n % prime == 0) {
                isPrime = false;
            }
        }
        if (isPrime) {
            primes.add(n);
        }
    }
    return primes;
}
```

Remarques sur l'algorithme

- Compliqué ! ne respecte pas les patrons de méthodes.
- Un entier n est premier si aucun entier premier $p < n$ ne le divise, la boucle `for` imbriquée teste si n est premier.
- Cela fonctionne car on trouve les nombres premiers du plus petit au plus grand.
- dès qu'on trouve que `prime` divise `n` (`if` ligne 6), on sait que n n'est pas premier, on voudrait interrompre la boucle pour gagner du temps.

L'instruction `break` permet de quitter immédiatement la boucle (la plus imbriquée). La prochaine instruction réalisée est celle suivant immédiatement la boucle venant d'être interrompue.

Crible (version 2, break)

```java
public List<Integer> allPrimes(int max) {
    List<Integer> primes = new ArrayList<>();
    for (int n = 2; n <= max; n++) {
        boolean isPrime = true;
        for (Integer prime : primes) {
            if (n % prime == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            primes.add(n);
        }
    }
    return primes;
}
```

Extraction de la méthode

En extrayant la boucle interne dans une méthode propre, le `break` devient naturellement un `return`, on retombe sur les patrons connus.

Exercice

Réalisez l'extraction de la boucle `for` testant si n est divisible par un nombre premier. Utilisez les patrons de méthodes.

(solution au prochain transparent)

Solution

```
public List<Integer> allPrimes(int max) {
    List<Integer> primes = new ArrayList<>();
    for (int n = 2; n <= max; n++) {
        if (!hasDivider(n,primes)) {
            primes.add(n);
        }
    }
    return primes;
}

private boolean hasDivider(int n, Collection<Integer> primes) {
    for (Integer prime : primes) {
        if (divides(p,prime)) return true;
    }
}

private boolean divides(int p, int prime) {
    return p % prime == 0;
}
```

Un autre exemple

Le programme suivant calcule la somme des entiers entrés par l'utilisateur. L'utilisateur doit terminer par -1 pour indiquer la fin des données.

```
public class Main {
    private final static Scanner input = new Scanner(System.in);
    public static void main(String[] args) {
        int sum = 0;
        boolean isNotOver = true;
        while (isNotOver) {
            int value = input.nextInt();
            if (value == -1) {
                isNotOver = false;
            } else {
                sum = sum + value;
            }
        }
        System.out.prinln("Sum = " + sum);
    }
}
```

Analyse du programme

- Utilisation d'une boucle `while`, pour répéter les opérations de lire un entier et le sommer.
- Dans la boucle, il faut :
  1. lire l'entier,
  2. sommer l'entier, mais pas si c'est -1.
- On pourrait tester dans la condition d'arrêt du `while` si `value /= -1`, en déclarant (et initialisant) `value` avant le `while`.

Deuxième essai

```
public class Main {
    private final static Scanner input = new Scanner(System.in);
    public static void main(String[] args) {
        int sum = 0;
        int value = input.nextInt();
        while (value /= -1) {
            sum = sum + value;
            value = input.nextInt();
        }
        System.out.prinln("Sum = " + sum);
    }
}
```

- plus simple,
- mais la lecture de l'entrée est dupliquée : violation du principe DRY, *Don't Repeat Yourself !*.

Mieux (plus facile à comprendre) :

Quitter la boucle avec un `break`.

Bonne solution

```
public class Main {
    private final static Scanner input = new Scanner(System.in);
    public static void main(String[] args) {
        int sum = 0;
        while (true) {
            int value = input.nextInt();
            if (value == -1) break;
            sum = sum + value;
        }
        System.out.prinln("Sum = " + sum);
    }
}
```

- ordre des opérations clair,
- condition de sortie mise en évidence, au moment propice,
- pas de répétition,
- pas de variables avec un rôle factice.

Instruction `continue`

Supposons qu'on veuille

- sommer uniquement les valeurs positives,
- et terminer dès que l'entrée est 0.

Les valeurs négatives doivent être ignorées.

L'instruction `continue` permet

- de terminer immédiatement l'itération actuelle de la boucle, les instructions suivantes ne sont pas exécutées,
- et de revenir au début de la boucle pour l'itération suivante : évaluation de la condition d'arrêt puis itération suivante.

Exemple avec continue

```
public class Main {
    private final static Scanner input = new Scanner(System.in);
    public static void main(String[] args) {
        int sum = 0;
        while (true) {
            int value = input.nextInt();
            if (value == 0) break;
            if (value < 0) continue;
            sum = sum + value;
        }
        System.out.prinln("Sum = " + sum);
    }
}
```

Notes :

- `break` et `continue` fonctionne pour les boucles `while`, `do while` et `for`.

Labels (anecdotique)

Dans le cas de boucles imbriquées (à éviter en général), `break` et `continue` agissent par défaut sur la boucle la plus profonde dans laquelle ils apparaissent.

Pour quitter une boucle moins profonde, on utilise des labels : on donne un nom à la boucle, et on indique ce nom dans l'instruction `break` ou `continue`.

Exemple de label

```java
public List<Integer> allPrimes(int max) {
    List<Integer> primes = new ArrayList<>();

    searchAllIntegers:
    for (int n = 2; n <= max; n++) {
        for (Integer prime : primes) {
            if (n % prime == 0) {
                continue searchAllIntegers;
            }
        }
        primes.add(n);
    }

    return primes;
}
```

Quand utiliser des boucles imbriqués ?

Il est légitime d'utiliser des boucles imbriqués pour :

- du calcul matriciel : les matrices sont en deux dimensions, une boucle par dimension,
- d'autres objets bidimensionnels : images, grille, ...

```
for (int column = 0; column < nbColumns; column++) {
  for (int row = 0; row < nbRows; row++) {
    ...
  }
}
```

Exercice

Exercice

Reprendre le premier exemple du parcours en largeur d'un arbre dans votre IDE (version compliquée). Utilisez les outils d'extraction de méthode de l'IDE pour simplifier l'algorithme.

Partie 3

Le modèle mémoire de Java

La mémoire

La programmation impérative

La programmation impérative fonctionne sur le principe suivant :

- Programme composé d'instructions,
- les instructions sont exécutées séquentiellement,
- chaque instruction produit un effet sur l'ordinateur,
- un effet est une opération en mémoire : lecture ou écriture,

Le résultat du programme est la conséquence de tous ces effets.

Objectif

Comprendre l'impact de chaque instruction sur l'état de l'ordinateur

Modèle de représentation de la mémoire

- Les processeurs modernes sont extrêmement complexes !
- Nous étudierons un modèle simplifié de la mémoire.
- Critères du modèle :
  - Petit nombre de règles et principes,
  - Cohérence avec les comportements observables des programmes.
- Applicable à la plupart des langages de programmation.

Représentation primaire de la mémoire

Premier niveau d'abstraction :

- Mémoire de l'ordinateur = tableau d'une seule colonne,
- chaque case a une adresse (indice de ligne),
- (en Java, les références sont des adresses,)
- contenu des cases : mots binaires, représentant chacun
  - une donnée primitive,
  - ou une adresse.

Note

Toutes les cases ne sont pas utilisables, mais cela est géré par le langage de programmation, en collaboration avec le système d'exploitation.

Représentation secondaire de la mémoire

Deuxième niveau d'abstraction, en trois zones :

- la zone statique, pour
  - les instructions du programme,
  - les valeurs des constantes, les littéraux,
- la pile, pour
  - les variables et les paramètres des méthodes,
  - mémoire à court terme,
- le tas, pour
  - les objets et leur propriétés, les tableaux,
  - mémoire à long terme,

On y ajoute les registres, très peu nombreux, non directement utilisables par le programmeur.

La zone des instructions

Zone statique :

- chargée au lancement, non modifiée durant l'exécution,
- contient les instructions compilées du programme, groupées par méthodes,
- contient aussi les constantes et littéraux, en particulier les chaînes de caractères,
- contient pour chaque classe, la table des méthodes (avec une référence vers la première instruction de chacune).

La zone des instructions

Les instructions sont exécutées séquentiellement :

- pointeur d'instruction (ou compteur du programme) : registre contenant l'adresse de la prochaine instruction,
- avance après chaque instruction,
- peut sauter vers une autre instruction :
  - appels de méthode, `return`,
  - structures de contrôle `if`, `while`, `for`, `switch`,
  - instructions `break`, `continue`, `yield`, `throw`.

La zone du tas

Le tas (*heap*)

- contient les objets et tableaux créés durant l'exécution du programme,
- chaque objet/tableau occupe un bloc, constitué de cases consécutives,
- les propriétés des objets ou le contenu des cases des tableaux sont directement stockés dans le bloc :
  - par leur valeur (types primitifs),
  - par leur référence (types références).
- Référence = adresse du premier emplacement du bloc.

Note

En Java, les blocs du tas existent indéfiniment.

La zone de la pile

La pile (*stack*)

- contient les variables et les paramètres de chaque méthode en cours,
- constitué d'un large bloc de cases consécutives de la mémoire (quelques milliers en Java),
- stockage temporaire, les variables disparaissent en fin de méthode,
- organisée en couches, une couche par méthode en cours d'exécution.

Note

Comportement dynamique (apparition et disparition des valeurs). Taille limitée : trop d'appels de méthodes imbriqués entraîne l'erreur StackOverflow.

Les couches de la pile

La pile est constitué de couches, une par méthode non-terminée.

- La couche active, au sommet de la pile, contient les variables et paramètres de la méthode en cours d'exécution.
- Elle contient aussi une case pour stocker `this`, la référence de l'objet dont la méthode s'exécute.
- Appel de méthode : empilement d'une nouvelle couche active. L'ancienne couche active devient inaccessible.
- Un retour de méthode supprime la couche active (ce qui termine la méthode), la couche inférieur redevient active : on restaure les variables et paramètres précédant l'appel de méthode.

L'organisation de la pile en couches permet donc la gestion des appels de méthode.

Représentation des objets dans le tas

Un objet dans le tas = un bloc de cases consécutives.

- Une case par propriété de l'objet, contenant :
  - une valeur (type primitif),
  - une référence (type référence).
- Une case pour identifier la classe de l'objet, contenant une référence vers la table des méthodes de la classe dans la zone d'instruction.

Polymorphisme

L'objet porte ses méthodes ! Ainsi l'appel d'une même méthode sur deux objets différents peut exécuter deux blocs d'instructions différents.

Le ramasse-miettes

Persistance des objets dans le tas : les objets sont créés, mais ne sont jamais supprimés.

Problème : accumulation d'objets qui encombrent la mémoire. Memory leak (fuite de mémoire).

Solution : en Java, mécanisme automatique de désallocation des objets devenus inaccessibles (inutilisables) :

le Garbage Collector (GC)

a.k.a. ramasse-miettes, ou Glaneur de Cellule. Invisible pour le programmeur.

L'effet des instructions

La déclaration de variable

Dans une méthode, la déclaration d'une variable provoque :

1. l'ajout de la variable dans la couche active de la pile,
2. l'initialisation de cette variable.

L'affectation

Résolution de l'affectation :

1. évaluation de la *left-value* $\longrightarrow$ emplacement mémoire m :
   - variable, paramètre de la méthode : $m \in$ couche active de la pile,
   - propriéte d'un objet, d'une classe, cellule d'un tableau : $m \in$ tas.
2. évaluation de la *right-value* $\longrightarrow$ valeur v :
   - une valeur (type primitif),
   - ou une référence (type référence).
3. remplacement du contenu de la case m par v.

La construction d'un objet

Résolution de la création d'un objet avec `new` :

1. allocation d'un bloc du tas,
2. initialisation des propriétés de l'objet,
3. appel du constructeur (*cf* appel de méthode),
4. retourne la référence du nouvel objet = adresse du bloc.

L'appel de méthode

Résolution d'un appel de méthode :

1. évaluation de l'objet appelant la méthode,
2. recherche de la première instruction de la méthode (en utilisant la table des méthodes de l'objet dans la zone d'instructions),
3. évaluation des arguments de la méthode,
4. sauvegarde du pointeur d'instruction dans la couche active de la pile,
5. ajout d'une couche dans la pile, avec `this` contenant la référence de l'objet, et une case par paramètre, contenant les valeurs des arguments,
6. saut du registre pointeur d'instruction vers la première instruction de la méthode.

La méthode est ensuite évaluée jusqu'au retour.

Le retour de méthode

Résolution de `return` :

1. évaluation de l'expression de la valeur de retour (dans un registre),
2. suppression de la couche supérieure de la pile,
3. restauration du pointeur d'instruction, stocké lors de l'appel de méthode, provoquant un saut vers l'instruction ayant appelé la méthode qui vient de retourner.

La valeur de retour est disponible dans son registre, pour terminer l'instruction contenant l'appel de méthode.

Les structures de contrôles

Les structures de contrôles (`if`, `while`, `for`, `switch`) fonctionnent toutes par saut, c'est-à-dire par modification du registre de pointeur d'instruction. Le saut est *controlé* par les conditions booléennes ou les valeurs des expressions.

La fin d'un bloc provoque la suppression des variables définies dans le bloc de la couche active de la pile.

Les instructions `break`, `continue`, `yield` fonctionnent par saut.
L'instruction `throw` ressemble au `return`, en plus complexe.

Récapitulatif

Modèle d'évaluation

- Modèle de la mémoire, abstraction en 3 zones :
  - zone statique,
  - pile,
  - tas.
- Modèle d'évaluation, expliquant :
  - chaque instruction,
  - les appels de méthodes, avec la pile,
  - les structures de contrôle,
  - le polymorphisme objet : la méthode exécutée est celle de l'objet, déterminée à sa création.

Intérêt du modèle

Les aspects techniques doivent être compris : important pour

- comprendre le comportement d'un programme,
- interpréter et corriger des erreurs d'un programme,
- améliorer les performances d'un programme.

D'un point de vue fondamental, liens avec l'architecture, les systèmes d'exploitation, la compilation, la théorie des langages de programmation.

Perspective

Ces connaissances sont essentielles pour devenir un programmeur compétent.

Partie 4

Tableaux, collections et itérateurs

Les collections

Collections

Pourquoi des collections d'objets ?

- pour manipuler des ensembles d'objets simultanément,
- pour ranger et accéder à des objets selon les besoins,

Une variété de collections

On ne manipule pas toutes les collections de la même façon :

- grouper des valeurs pour les traiter ensemble,
- ordonner des valeurs,
- tester l'appartenance d'éléments,
- créer une file d'attente,
- associer des valeurs entre elles,
- *etc.*

Différentes collections, plus ou moins efficaces selon les opérations.

Interface de collection

Les opérations supportées par une structure de données forment son interface.

Une interface est une *liste de déclaration de méthodes* implémentables par une classe.

Une classe qui implémente ces méthodes *implémente l'interface*.

Une interface est un contrat : si une classe implémente l'interface `List<Elt>`, elle doit avoir des méthodes `add`, `get`, `size`, ...

Les listes

L'interface `List`

On connait déjà `List` : séquence ordonnée indexable par des entiers.

```
public interface List<E> {
  boolean add(E e);
  E get(int index);
  int indexOf(E e);
  ...
}
```

Implémentations : `Vector`, `LinkedList`, `ArrayList`

Exemples de collection

Exemples de collections dans la vie courante.

- Réfrigérateur : collection d'aliments mis au frais.
- File d'attente au guichet : collection de personnes attendant dans l'ordre d'arrivée.
- Pile de T-shirts : collection de vêtements avec un ordre spatial.
- Annuaire : collection de noms et de numéros de téléphone.

Les ensembles

Réfrigérateur

Exemples d'opérations :

- prendre un aliment,
- déposer un aliment,
- compter le nombre d'aliments,
- tester la présence d'un aliment.

Structure d'ensemble (*Set*)

L'interface Set

```
public interface Set<E> {
    boolean add(E e);
    boolean contains(E e);
    boolean isEmpty();
    boolean remove(E e);
    ...
}
```

E peut être remplacé par le type d'objets de votre choix (Set<Integer>, Set<String>, Set<List<Double>>, ...).

Implémentations disponibles : TreeSet, HashSet

Type de retour

Pourquoi `add`, `remove` retournent des valeurs de vérités (`boolean`) ?

Règle : par convention, les méthodes modifiant une collection retourne

- `true` si la modification a été réalisée,
- `false` si aucune modification n'a eu lieu.

Par exemple, `false` est retournée par `set.remove(elt)` si `elt` n'était pas dans l'ensemble.

Exemple d'utilisation de `Set`

```
public Set<String> dictionary(List<String> text) {
  Set<String> words = new HashSet<>();
  for (String word : text) {
    words.add(word);
  }
  return words;
 }
```

Notez :

- l'initialisation d'un ensemble vide,
- l'opération d'ajout d'un élément.

Différences entre `List` et `Set`

- un objet `Set` contient chaque élément au plus une fois, un objet `List` peut avoir des répétitions,
- un objet `Set` peut rapidement accomplir les opérations `add`, `contains`, `remove`. Un objet `List` accomplit `contains` et `remove` très lentement.
- les éléments d'un objet `List` sont classés en séquence (0,1,2,...) et indicés, les éléments d'un objet `Set` ne sont pas classés. On ne peut pas trier un `Set`.
- `Set` utilise la méthode `equals` de ses éléments, et `HashSet` utilise la méthode `hashCode`, il faut donc les implémenter ou utiliser les méthodes par défaut.

Les files et piles

File d'attente

Exemples d'opérations :

- ajouter une personne en fin de file,
- récupérer la personne en début de file,
- tester si la file est vide,
- compter le nombre de personnes en attente.

Structure de file (*Queue*).

Les éléments sortent dans l'ordre où ils sont rentrés (ordre FIFO, *First-in First-out*).

L'interface Queue

```
public interface Queue<E> {
  boolean offer(E e);
  E poll();
  E peek();
  boolean isEmpty();
  int size();
  ...
}
```

Implémentations : `ArrayDeque`, `LinkedList`, `PriorityQueue`

Pile de T-shirts

Exemples d'opérations :

- mettre un T-shirt en haut de la pile,
- récupérer le T-shirt du haut de la pile,
- tester s'il y a un T-shirt.

Structure de pile (*Stack*).

Les éléments sortent dans l'ordre inverse où ils sont rentrés (LIFO, *Last-in First-out*).

Piles et files

- File : premier entré, premier sorti (FIFO : *first-in, first-out*), une file a deux extrémités actives, une extrémité où l'on rentre et l'autre dont l'on sort.
- Pile : dernier entré, premier sorti (LIFO : *last-in first-out*), une pile a une seule extrémité active (le haut de la pile), à laquelle on ajoute ou retire des éléments.

En Java, une seule interface commune aux deux : file à double extrémité (*deque* : abbréviation de *double-ended queue*).

L'interface Deque

```
public interface Deque<E> {
  boolean offerFirst(E e);
  E pollFirst();
  E peekFirst();
  boolean offerLast(E e);
  E pollLast();
  E peekLast();
  ...
}
```

- `poll` : retirer et retourner l'élément à l'extrémité,
- `peek` : retourner l'élément à l'extrémité, sans le retirer de la deque.

Implémentations : `ArrayDeque`, `LinkedList`

Patron de méthode des files

Traiter tous les éléments de la file, dans l'ordre :

```
public void process(Deque<E> queue) {
  while (!queue.isEmpty()) {
    E first = queue.pollFirst();
    process(first);
  }
}
```

Le traitement d'un élément peut provoquer l'ajout de nouveaux éléments dans la file.

Attention à tester que la file est non-vide avant d'extraire un élément ! (sinon vous obtiendrez `null`).

Exemple d'utilisation d'une deque

- Au rugby on peut marquer soit 3 points, soit 5 points, soit 7 points d'un coup.
- Le score d'une équipe ne peut donc pas être un seul point, ou 2 points, ou 4 points, mais peut être 6 points $(3+3)$, 8 points $(3+5)$, 9 points $(3+3+3)$, ...
- Quels sont les scores réalisables ?
- Pour chaque score n réalisable, $n+3$, $n+5$ et $n+7$ sont aussi réalisables. 0 est réalisable.

Exemple d'utilisation d'une deque

```
Set<Integer> scores = new HashSet<>();

public void computeScores() {
  Deque<Integer> feasibles = new ArrayDeque<>();
  feasibles.offerLast(0);
  while (!feasibles.isEmpty()) {
    addScore(feasibles.pollFirst(), feasibles);
  }
}

public void addScore(int n, Deque<Integer> feasibles) {
  if (n >= 100 || scores.contains(n)) { return; }
  scores.add(n);
  feasibles.offerLast(n+3);
  feasibles.offerLast(n+5);
  feasibles.offerLast(n+7);
}
```

Exemple d'utilisation d'une deque

- `offerLast` + `pollFirst` : utilisation de la deque comme une file.
- Initialisation avec `new ArrayDeque<>()`.
- La file contient les scores réalisables.
- Si un score est déjà connu, on le saute.
- Similaire à l'algorithme de parcours en largeur de graphe.

Les tables d'associations

Annuaire

Exemples d'opérations :

- ajouter un contact (nom + numéro),
- chercher le numéro d'un contact,
- retirer un contact.

Structure de table d'associations (*Map*), en Python : *dictionnaire*.

L'interface Map

```
public interface Map<Key,Value> {
  Value put(Key k, Value v);
  boolean containsKey(Key k);
  Value get(Key k);
  Value remove(Key k);
  Set<Key> keySet();
  Collection<Value> values();
  ...
}
```

De nouveau, on peut remplacer `Key` et `Value` par n'importe quels types.

Implémentations : `HashMap`, `TreeMap`

Exemple d'utilisation de `Map`

On veut compter le nombre d'appels téléphoniques effectués vers chaque numéro.

- Associer à chaque numéro le nombre d'appel.
- Lorsqu'un appel est effectué, récupérer le compte précédent, et l'augmenter de 1. Cas spécial : premier appel.
- Notez bien dans le prochain transparent : l'initialisation, l'ajout d'une association, le test d'une clé, la récupération d'une valeur.

Exemple d'utilisation de Map

```
public class PhoneCallStats {
  Map<PhoneNumber,Integer> callCounts = new HashMap<>();

  public void call(PhoneNumber number) {
    int oldCount = this.getCallCount(number);
    callCounts.put(number,oldCount+1);
  }

  public int getCallCount(PhoneNumber number) {
    return (callCounts.containsKey(number)) ?
      callCounts.get(number) :
      0;
  }
}
```

L'interface Collection

L'interface `Collection`

`Set`, `Queue`, `Deque`, `List` *étendent* l'interface `Collection` (mais pas `Map` !).

```
public interface Set<E> extends Collection<E> {
   ...
}
```

Ce qui signifie que toutes ces interfaces possèdent aussi les services définies par l'interface `Collection`.

L'interface Collection

```
public interface Collection<E> {
  boolean add(E e);
  boolean remove (E e);
  int size();
  Iterator<E> iterator();
  ...
}
```

Définition d'une collection dans votre programme :

```
Collection<Item> myItems = new ArrayList<Item>();
```

Trier une collection

La classe `Collections` (notez le 's') contient des fonctions (méthodes statiques) pour travailler sur les collections.
Exemple :

```
List<Item> myList = ...;
Collections.sort(myList);
```

Pour trier, il faut que la classe `Item` définisse une méthode `compareTo` (qu'elle implémente l'interface `Comparable<Item>`), ce qui permet d'ordonner les `Item`.

Itérer sur une collection

Comment faire une manipulation pour chaque item d'une collection ?

La boucle `for` permet de faire les mêmes instructions pour tous les éléments d'une collection.

```
Collection<Item> itemList = ...;
for (Item item : itemList) {
  ... // instructions utilisant item
}
```

Les éléments sont traités un par un. Ordre des éléments : dépend de l'implémentation de la collection.

Syntaxe du `for`

```
for (Item item : itemList) { ... }
```

- `for` : mot-clé de répétition,
- `(...)` : les paramètres de la répétition :
  - `Item` : le type des éléments de la collection,
  - `item` : un identifiant pour une nouvelle variable désignant l'élément de la collection, qui change à chaque itération,
  - “`:`” : se lit “*in*” (“contenu dans”),
  - `itemList` : la collection sur laquelle itérer.
- `{...}` : les instructions à répéter pour chaque élément de la collection.

En résumé

nom	interface	création
liste	`List`	`new ArrayList<>()`
pile/file	`Deque`	`new ArrayDeque<>()`
ensemble	`Set`	`new HashSet<>()`
table d'associations	`Map`	`new HashMap<>()`

Les noms des interfaces sont accompagnés du type des éléments : `List<Integer>`, `Set<Node>`, `Map<Node, Point2D>`, `...`

Entre `<...>`, on utilise `Integer`, `Double`, `Boolean` plutôt que `int`, `double`, `boolean`.

Les tableaux

Les tableaux

Un tableau est un espace mémoire pouvant contenir plusieurs objets simultanément. Les objets sont stockés dans des places (les cases) distinctes et indicées de 0 à `length - 1`, où `length` est la longueur du tableau.

- Un tableau est considéré comme un objet, il a des méthodes.
- La taille d'un tableau est fixe et décidée lorsque le tableau est créé.
- Les objets "contenus" par un tableau sont de même type (homogénéité).

Que contient un tableau ?

Comme pour les variables et propriétés :

- un objet est stocké dans une case par sa référence,
- un tableau est stocké dans une case par sa référence,
- une valeur d'un type primitif est stockée directement (la case contient sa valeur).

Ainsi, affecter un objet dans une case du tableau ne fait qu'y mettre sa référence, l'objet n'est (heureusement) pas dupliqué !

Exemple de syntaxe

```java
public int sumOfSquares(int n) {
  // déclaration, création (new)
  int[] tab = new int[n];
  // accès à la longueur (propriété du tableau)
  for (int index = 0; index < tab.length; index++) {
    tab[index] = index * index; // affectation à une case
  }
  int squareSum = 0;
  // itération sur tous les éléments
  for (int square : tab) {
    squareSum = squareSum + square;
  }
  System.out.println(squareSum);
}
```

Déclaration d'un tableau

Déclaration :

- Type d'un tableau : `int[]`, `Double[]`, `Book[]`, ...les crochets sont propres aux tableaux.
- Le type ne porte pas l'indication de la longueur du tableau.
- Nom d'un tableau : même convention que les variables/propriétés.

Initialisation :

- `new` comme pour les objets (marque l'allocation d'un bloc de mémoire pour contenir le tableau),
- puis indication du type des éléments contenu du tableau,
- puis la longueur entre crochet.

```
int[] integers = new int[100];
Book[] library = new Book[100 * 1000];
```

Initialisation d'un tableau

Après la création, les cases du tableau sont initialisées :

- à 0 pour les types numériques,
- à `false` pour le type `boolean`,
- au caractère nul pour le type `char`,
- à `null` pour les types références d'objets.

Conseil : essayez autant que possible d'initialiser les valeurs du tableau juste après sa création, c'est une habitude qui évite bien des erreurs.

Initialisation explicite

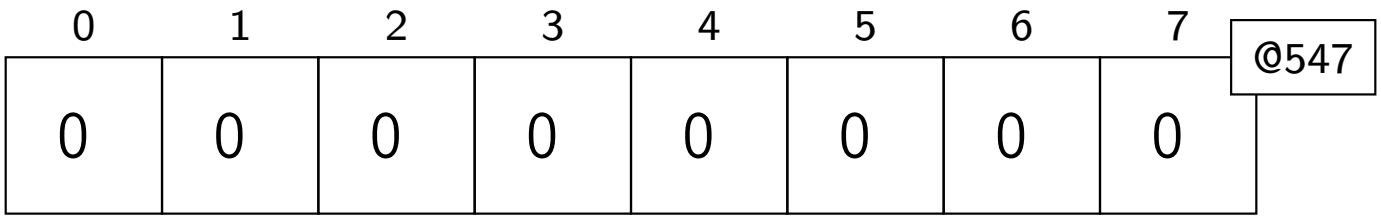
On peut aussi initialiser un tableau en donnant la liste explicite de valeurs stockées dans le tableau.

```
int[] someInts = { 1; 5; 3; 7; 2; 8; 4; 6 };
String[] someNames = { "Alice", "Bob", "Carol", "Dave" };
```

Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

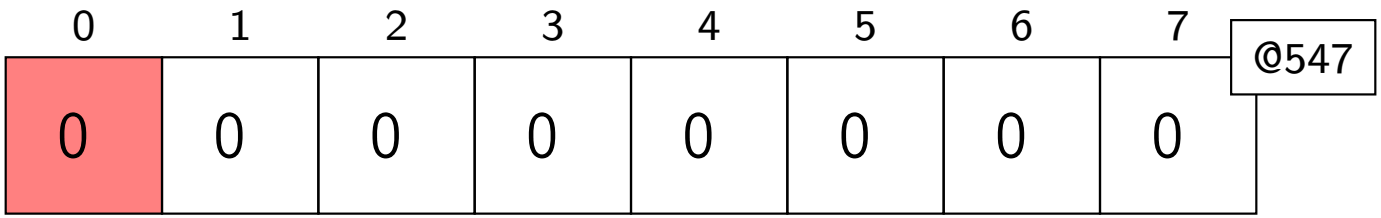
table = @547



Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

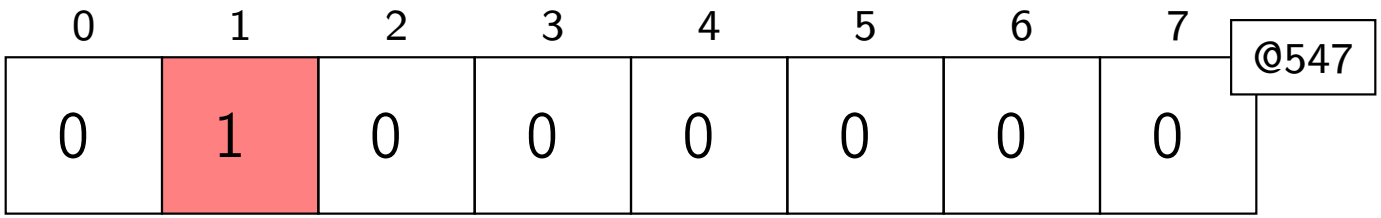
table = @547



Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

`table = @547`

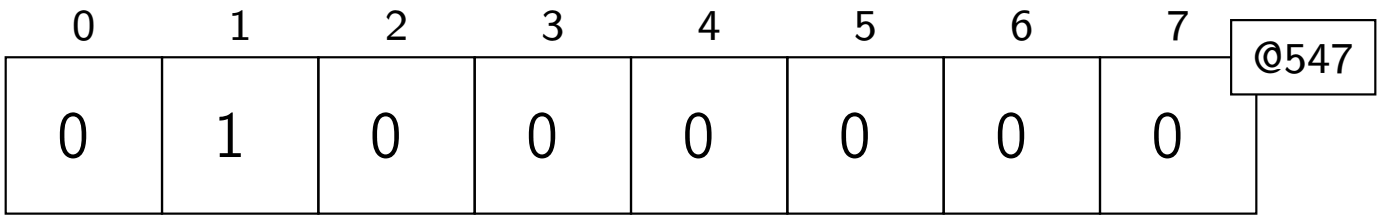


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 2

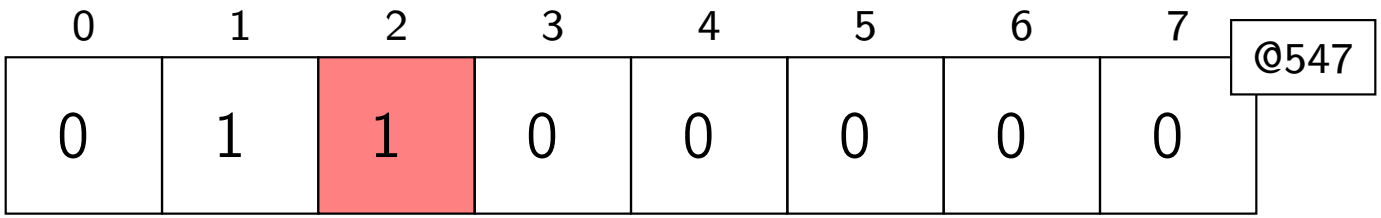


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 2

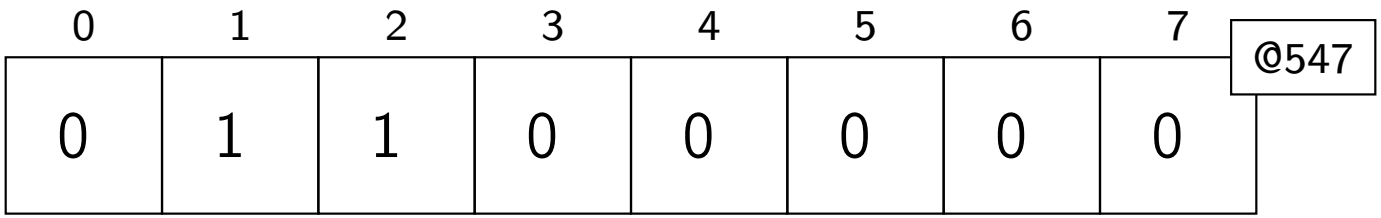


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 3

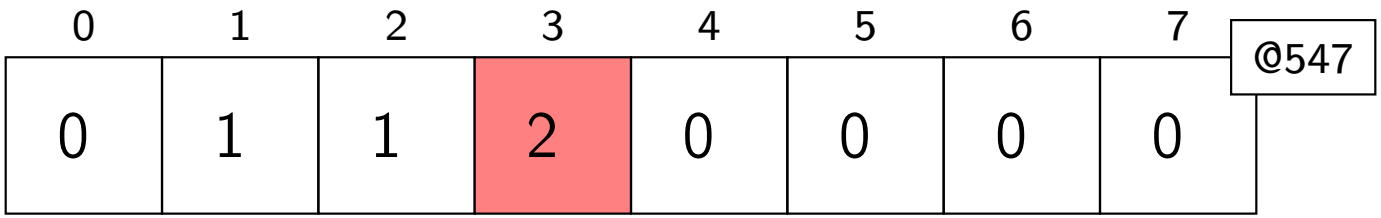


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 3

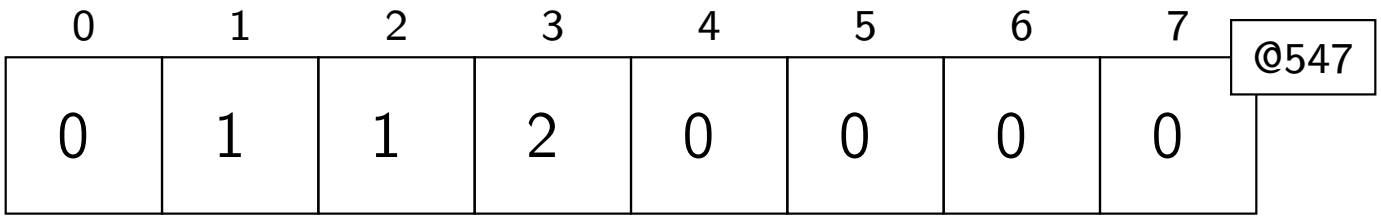


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 4

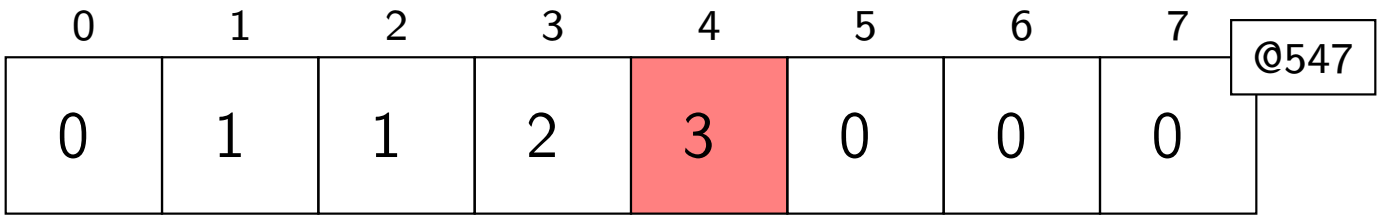


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

`table = @547`

`i = 4`

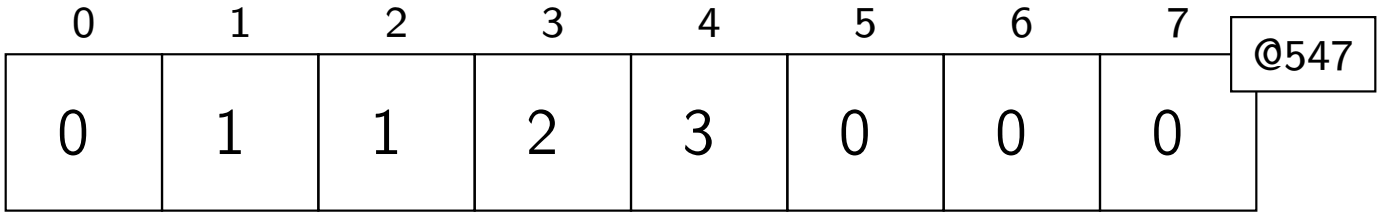


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 5

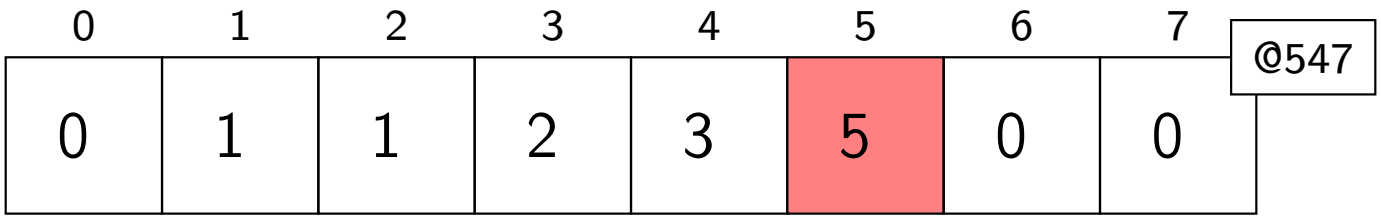


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 5

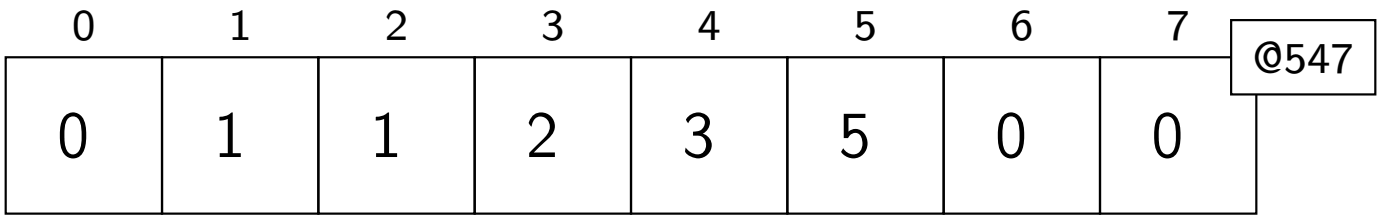


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 6

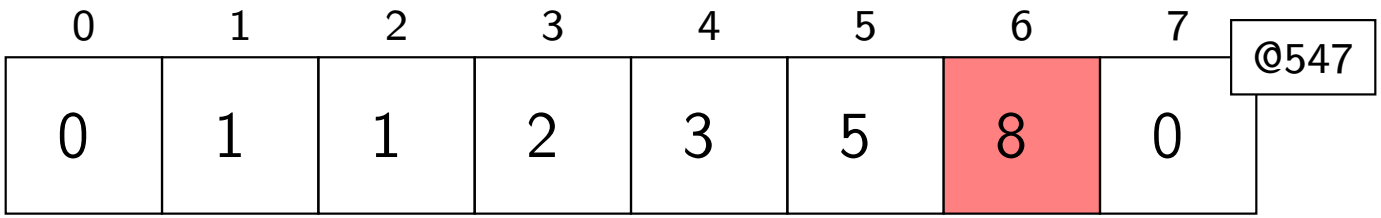


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

`table = @547`

`i = 6`

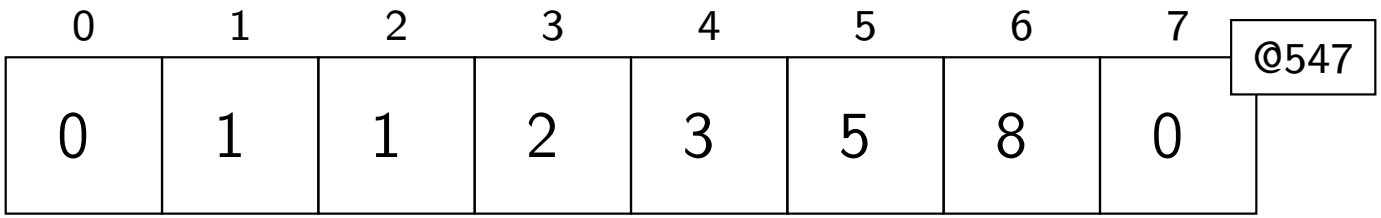


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

i = 7

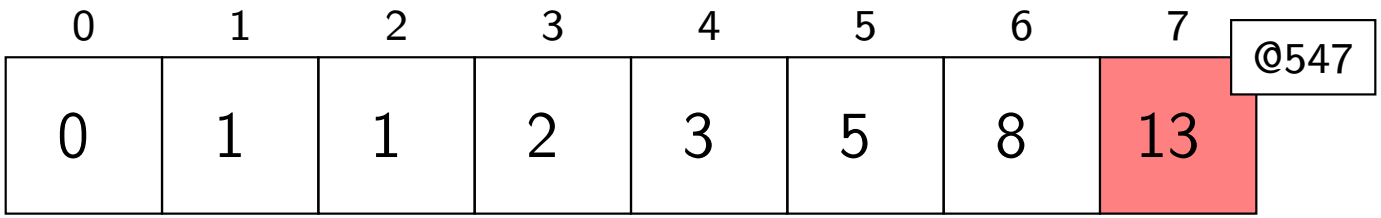


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

`table = @547`

`i = 7`

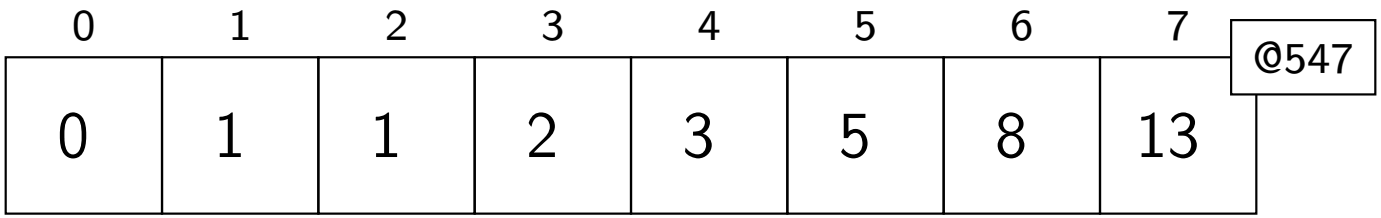


Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547

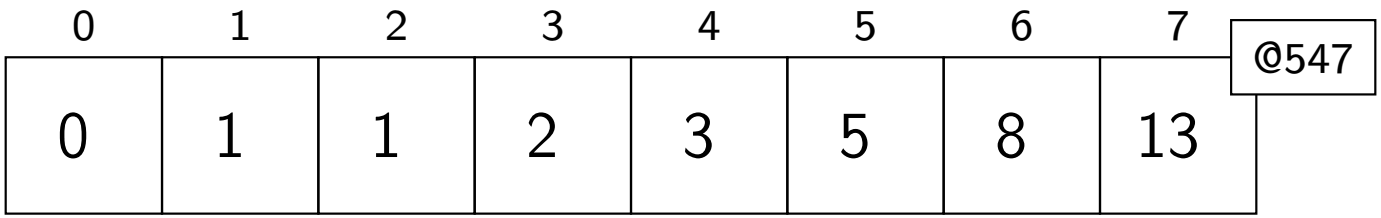
i = 8



Exemple d'initialisation

```
int[] table = new int[8];
table[0] = 0;
table[1] = 1;
for (int i = 2; i < table.length; i++) {
  table[i] = table[i-1] + table[i-2];
}
```

table = @547



Accès et modification d'un tableau

La syntaxe *crochetée* (nom du tableau, puis indice de la case entre crochet) permet d'accèder au contenu du tableau.

```
int[] array = { 10, 11, 12, 13 };
int sum = array[0] + array[1]; // sum == 21
int twelve = array[sum - 19]; // avec une expression
```

L'indice doit avoir le type `int`. De plus, les cases sont des *l-values*, on peut modifier le contenu d'une case par affectation :

```
array[2] = -1;
array[3] = array[3] + 20; // array[3] == 33
sum = sum + array[2] + array[3]; // sum == 53
```

Parcours d'un tableau

Pour parcourir (traiter) tous les éléments d'un tableau :

```
// si l'indice de la case est nécessaire
for (int cell = 0; cell < table.length; i++) {
  process(cell,table[cell]);
}

// si l'indice est inutile
for (Elt elt : table) {
  process(elt);
}
```

On peut bien sûr utiliser `break`, `return`, `continue` normalement.

Avantages et désavantages

Avantages :

- accès très rapide à n'importe quelle case,
- simplicité,
- facile à copier,
- facile à parcourir avec une boucle `for`.

Inconvénients :

- La taille est fixe, et choisie lors de sa création.
- Bas niveau : les opérations faciles à réaliser sont souvent trop simples par rapport aux besoins.
- L'accès à une case hors du tableau provoque une exception `ArrayIndexOutOfBoundsException` :

```
int[] table = new int[100];
table[100] = 42; // BOUM !!!
```

amU Faculté des sciences Aix Marseille Université

Algorithme : recherche d'un élément

Tâche : trouver l'indice où apparait une valeur dans le tableau.

```
public int find(Elt e, Elt[] table) {
  for (int cell = 0; cell < table.length; i++) {
    if (table[cell] == e) {
      return cell;
    }
  }
  return -1;
}
```

Algorithme : recherche du minimum

Tâche : trouver l'indice de l'élément minimum du tableau.

```
public int getArgMin(int[] table) {
  int argMin = 0;
  for (int cell = 1; cell < table.length; cell++) {
    if (table[cell] < table[argMin]) {
      argMin = cell;
    }
  }
  return argMin;
}
```

Algorithme : tri d'un tableau

Tâche : Trier le tableau en ordre croissant.

```
public void sort(int[] table) {
  sort(table, 0, table.length - 1);
}
public void sort(int[] table, int fromIndex, int toIndex) {
  if (toIndex - fromIndex < 1) return;
  int pivot = partition(table, fromIndex, toIndex);
  sort(table, fromIndex, pivot-1);
  sort(table, pivot+1, toIndex);
}
private int partition(int[] table, int low, int high) {
  int pivot = low;
  low = low + 1;
  while (true) {
    low = firstBiggerThanPivot(table, pivot, low, high);
    high = lastSmallerThanPivot(table, pivot, low, high);
    if (low > high) break;
    swap(table,low,high);
  }
  swap(table, pivot, low - 1);
  return low - 1;
}
```

Algorithme : tri d'un tableau (II)

Méthodes auxiliaires :

```
public int firstBiggerThanPivot(int[] table, int pivot,
                                int cell, int high) {
  while (cell <= high && table[cell] <= table[pivot]) {
    cell++;
  }
  return cell;
}

public int lastSmallerThanPivot(int[] table, int pivot,
                                int low, int cell) {
  while (cell >= low && table[cell] > table[pivot]) {
    cell--;
  }
  return cell;
}

public void swap(int[] table, int cell1, int cell2) {
  int buffer = table[cell1];
  table[cell1] = table[cell2];
  table[cell2] = buffer;
}
```

Exercice

Exercice

Créer un projet java et recopier cet algorithme. Dans `swap` ajouter une instruction pour afficher le contenu du tableau (faites une méthode pour l'affichage). Dans la deuxième méthode `sort`, ajouter une instruction au début pour écrire les valeurs des paramètres.
Utiliser le programme pour trier un tableau d'une vingtaine de valeurs. En analysant l'affichage, essayer de comprendre le fonctionnement de l'algorithme.

Tri d'un tableau

Écrire un algorithme de tri d'un tableau est difficile, et il est facile de se tromper !

On utilise donc un algorithme déjà écrit :

```
int[] values = { 6, 3, 5, 1, 2, 8, 4, 7 };
Arrays.sort(values);

String[] names = { "Alice", "Dave", "Bob", "Charlie" };
Arrays.sort(names);
```

Alternative plus rapide pour les gros tableaux :
`Arrays.parallelSort(table)` (algorithme parallèle)

La classe Arrays

La classe Arrays de la librairie standard contient diverses méthodes statiques pour manipuler les tableaux :

```
int[] values = { 1, 4, 7, 14, 23 }
int[] moreValues = { 2, 3, 5, 7, 11, 13, 17, 19 }

Arrays.equals(values, moreValues); // false
Arrays.binarySearch(values, 14); // 3 car values[3] == 14
Arrays.fill(values, 5); // values == { 5, 5, 5, 5, 5 }
```

Arrays.binarySearch fonctionne uniquement sur les tableaux triés !

Copie d'un tableau

`Arrays` contient aussi des méthodes très efficaces pour réaliser des copies d'un tableau.

```
int[] values = { 1, 2, 3, 4, 5, 6, 7, 8 };

int[] res1 = Arrays.copyOf(values, 5);
// res1 == { 1, 2, 3, 4, 5 }

int[] res2 = Arrays.copyOf(values, 10);
// res2 == { 1, 2, ..., 8, 0, 0 }
// complète (pad) avec des 0

int[] res3 = Arrays.copyOfRange(values, 2, 6);
// res3 == { 3, 4, 5, 6 }
// 2e argument : premier indice copié
// 3e argument : premier indice non-copié
```

Égalité de deux tableaux

Pour tester l'égalité de deux tableaux :

- la méthode `.equals` du tableau et `==` testent l'égalité de référence (c'est le même tableau, même adresse en mémoire),
- `Arrays.equals` teste que les valeurs contenues dans chaque case sont égales par leurs méthodes `.equals`.
- `Arrays.deepEquals` teste que les valeurs contenues dans chaque case sont égales par leurs méthodes `.equals` ou par `Arrays.deepEquals` pour les tableaux.

`Arrays.equals` et `Arrays.deepEquals` diffèrent donc pour les tableaux de tableaux.

Tableaux bidimensionnels

Un tableau de tableaux est un tableau à plusieurs dimensions. Typiquement, on évitera d'avoir plus de deux dimensions.

Un tableau bidimensionnel est un tableau dont chaque case contient une référence à un autre tableau.

- `int` : type des entiers,
- `int[]` : type des tableaux d'entiers,
- `int[][]` : type des tableaux de tableaux d'entiers.

Déclaration d'un tableau bidimensionnel

- `new int[3][4];` 3 lignes, 4 colonnes.
- `new int[3][];` 3 lignes, toutes nulles. Il faut alors initialiser chaque ligne.

```
int[][] array = int[3][4];
array.length; // 3
for (int row = 0; row < array.length; row++) {
  for (int column = 0; column < array[row]; column++) {
    array[row][column] = row * column;
  }
}

int[][] table = int[3][]; // 3 lignes non-définies
table[0]; // null
table[0][1]; // erreur : NullPointerException
```

Tableaux non-rectangulaires

Les lignes peuvent être initialisées avec des longueurs distinctes.

```
double[][] triangularTable = new double[4][];
for (int row = 0; row < triangularTable.length; row++) {
  triangularTable[row] = new double[row+1];
  for (int column = 0;
       column < triangularTable[row].length;
       column++)
  {
    triangularTable[row][column] = 10 * row + column;
  }
}
```

Quand utiliser les tableaux ?

Cas d'usages :

- pour utiliser une librairie qui exige des tableaux,
- pour implémenter des structures de données (listes, files, ...),
- pour optimiser la performance de certains programmes,
- pour du calcul numérique, vectoriel ou matriciel.

En général, préférer les collections :

- plus simple,
- plus de fonctionnalités,
- hormis quelques opérations, plus efficace.

Itérables

Exemple : Itérable

De nombreux objets sont itérables :

- les collections, les tableaux,
- les mots d'un texte,
- une séquence définie par une relation de récurrence $u_{n+1} = f(u_n)$,
- les pixels d'une image,
- ...

Besoin d'un mécanisme pour unifier le traitement.

Boucle `for` des itérables

```
Iterable<Pixel> image = ...;
for (Pixel pix : image) {
  doSomething(pix);
}
```

- Syntaxe : `for (EltType elt : provider) { ... }`,
- `provider` doit implémenter `Iterable<EltType>`,
- `break`, `continue` et `return` autorisés.

L'interface `Iterable`

```
public interface Iterable<E> {
  Iterator<E> iterator();
}
```

- Être itérable, c'est avoir la capacité de créer un objet chargé de l'itération (`Iterator`).

L'interface `Iterator`

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  ...
}
```

- `hasNext` permet de savoir s'il existe au moins un élément supplémentaire,
- `next` permet de récupérer le prochain élément.
- l'`iterator` est l'objet responsable de fournir les éléments un par un.

`Iterable` vs `Iterator`

- `Iterable` possède des éléments qui peuvent être examinés un par un (peut être itéré).
- `Iterator` fournit des éléments un par un (itère).

Généralement :

- Suffixe `able` : indique une capacité.
- Suffixe `ator` : indique une responsabilité.

Itérateur d'entiers

```
public class Range implements Iterator<Integer> {
  private int next = 1;
  private final int max;
  public Range(int min, int max) {
    this.next = min;
    this.max = max;
  }
  public boolean hasNext() { return next <= max; }
  public Integer next() {
    if (!hasNext()) throw new NoSuchElementException();
    return next++;
  }
}
```

Itérateurs d'entiers (utilisation, I)

```
Iterator<Integer> range = new Range(1,10);
range.hasNext(); // true
range.next(); // 1
range.next(); // 2
range.hasNext(); // true
range.next(); // 3
for (int i = 4; i < 10; i++) { range.next(); }
range.next(); // 10
range.hasNext(); // false
```

Itérateurs d'entiers (utilisation, II)

```
Iterator<Integer> range = new Range(1,10);
int sum = 0;
while (range.hasNext()) {
  int next = range.next();
  sum = sum + next;
  System.out.println(next);
}
```

Le fragment ci-dessous est faux (Range n'est pas itérable) !

```
for (Integer i : new Range(1,10);) { // NON !
  System.out.println(i);
}
```

Intervalle itérable (I)

```
public class Interval implements Iterable<Integer> {
  public final int start;
  public final int stop;
  public interval(int start, int stop) {
    this.start = start;
    this.stop = stop;
  }
  Iterator<Integer> iterator() {
    return new Range(start,stop);
  }
}
```

Intervalle itérable (II)

Utilisation :

```
Interval interval = new Interval(1,10);
int sum = 0;
for (Integer i : interval) {
  sum = sum + i;
  System.out.println(i);
}
```

Raccourci pour :

```
Iterator<Integer> iterator = interval.iterator();
while (iterator.hasNext()) {
  int i = iterator.next();
  sum = sum + i;
  System.out.println(i);
}
```

Exercice

Exercice

Faire un monoïde des intervalles pour l'union, l'union de plusieurs intervalles est le plus petit intervalle les contenant tous. On commencera par ajouter un opérateur binaire d'union dans la classe des intervalles.

Exercice

Exercice

Écrire une classe pour représenter les suites définies par récurrence de la forme $u_0 = c$, $u_{n+1} = f(u_n)$. On souhaite qu'une telle suite soit itérable, l'itération retournant les termes de la suite dans l'ordre. On utilisera l'interface `DoubleUnaryOperator` pour représenter la fonction f.

(Pour simplifier vous pouvez commencer en restreignant aux suites dont la relation de récurrence est de la forme $u_{n+1} = au_n + b$.)

```
public interface DoubleUnaryOperator {
  double applyAsDouble(double x);
}
```

Suite (I)

```
public class Sequence implements Iterable<Double> {
  private final DoubleUnaryOperator f;
  private final double initialValue;

  public Sequence(DoubleUnaryOperator f, double x0) {
    this.f = f; this.initialValue = x0;
  }

  public Double term(int i) {
    for (Double d : this) {
      if (i == 0) return d;
      i--;
    }
    return 0;
  }

  public Iterator<Double> iterator() {
     return SequenceIterator(f,initialValue);
  }
}
```

Suite (II)

```
public class SequenceIterator implements Iterator<Double> {
  private final DoubleUnaryOperator f;
  private double nextValue;

  public SequenceIterator(DoubleUnaryOperator f, double x0) {
    this.f = f;
    this.nextValue = x0;
  }

  public boolean hasNext() { return true; }

  public Double next() {
    double current = nextValue;
    nextValue = f.applyAsDouble(nextValue);
    return current;
  }
}
```

Suite (III)

Utilisation :

```
public double sq(double x) { return x * x; }

public double sqrt(double target) {
  Sequence newtonRoot =
   new Sequence(x -> 0.5 * (x + target / x),
                target);
  for (Double possibleRoot : newtonRoot) {
    if (Math.abs(sq(possibleRoot) - target) < epsilon)
      return possibleRoot
  }
}

sqrt(139328.5387); // 373.26738231460837
Sequence powersOfTwo = new Sequence(x -> 2. * x, 1);
powersOfTwo.term(16); // 65536.0
```

Pourquoi séparer `Iterable` et `Iterator`

Que se passe-t-il si l'intervalle gère lui-même l'itération ?

```
Interval range = new Interval(1,10);
for (Integer i : range) {
  for (Integer j : range) {
    context.fillEllipse(10. * i, 10. * j, 8, 8);
  }
}
```

- L'objet `range` possèderait un seul état pour gérer les deux itérations,
- Les deux boucles `for` interfèreraient !
- les `Iterator` permettent de gérer l'état de l'itération.

Partie 5

Représentation des données

Énumérations

Énumération

Les énumérations sont des classes ayant un nombre fini et prédéterminé d'instances (connu au moment de programmer), permettant de représenter des données, comme par exemple :

- les jours de la semaine, les mois de l'année,
- les couleurs d'un jeu de cartes,
- l'état d'un système (actif, dormant, en attente).

Pourquoi utiliser des enums

On pourrait utiliser un codage entier pour représenter les différentes valeurs possibles (0 = lundi, 1 = mardi,...).

Avantage d'utiliser une énumération :

- plus lisible (`0` *vs.* `MONDAY`),
- plus sûr : `8` n'est pas un jour de la semaine, le compilateur java vérifie que le type `Day` est bien respecté,
- meilleure documentation : le type `Day` est plus précis que le type `int`

Exemple de syntaxe

On définit une énumération dans un fichier dédié (comme pour une classe normale) :

```
public enum Suit {
  SPADES, HEARTS, DIAMONDS, CLUBS;
}
```

- Il n'y a que 4 objets de la classe `Suit` : `Suit.SPADES`, `Suit.HEARTS`, ...
- on peut tester l'égalité des `Suit` avec `==`,
- on peut utiliser la méthode statique `Suit.values()` pour itérer sur toutes les options dans une boucle `for`.

```
for (Suit suit : Suit.values()) {
  sout.println("Ace of " + suit.toString());
}
```

enum et switch

La structure de contrôle `switch` accepte les `enum` :

```
public boolean isRed(Suit suit) {
  switch (suit) {
    case Suit.HEARTS :
    case Suit.DIAMONDS : return true;
    default : return false;
  }
}
```

Les options de l'`enum` sont considérées comme des littéraux.

Méthodes par défaut

- Les `enum` ne sont pas instanciables ! Les instances existent et sont utilisables immédiatement (`Suit.HEART`, ...),
- les instances possèdent des méthodes (par exemple, `compareTo`, `toString`, ...) qui peuvent être redéfinies (`@Override`).

```
public enum Suit {
  SPADES, HEARTS, DIAMONDS, CLUBS;

  @Override
  public String toString() {
    switch (this) {
      case SPADES : return "spades";
      case HEARTS : return "hearts";
      case DIAMONDS : return "diamonds";
      case CLUBS : default : return "clubs";
    }
  }
```

Méthodes et constructeurs

- On peut ajouter des méthodes et des propriétés comme dans n'importe quelle classe.
- L'initialisation des propriétés peut être faite par un constructeur privé (un `enum` n'est pas instanciable, donc n'a pas de constructeur public).
- Les arguments à passer au constructeur sont donnés à la déclaration de chaque option (*cf.* prochain transparent).
- L'instanciation des options se fait lorsque la classe est chargée (typiquement au début de l'exécution du programme).

Un exemple plus complet

```
public enum Suit {
  SPADES ("spades",'♠'),
  HEARTS ("hearts", '♡'),
  DIAMONDS ("diamonds", '♢'),
  CLUBS ("clubs", '♣');

  private final String name ;
  private final char symbol;

  private Suit(String name, char symbol) {
    this.name = name;
    this.symbol = symbol;
  }

  @Override
  public String toString() {
    return name;
  }

  public char getSymbol() {
    return symbol;
  }
}
```

Exercice

Exercice

Écrire un `enum` pour les mois de l'année, avec une méthode retournant le nombre de jour du mois (on considèrera que l'année n'est pas bissextile).

- Écrire une solution avec un `switch`.
- Écrire une solution avec une propriété `nbDays` initialisée par un constructeur.

Enregistrement

Enregistrement

Les enregistrements sont des classes dont les instances ont pour unique responsabilité d'agréger plusieurs informations en une seule valeur. Par exemple, pour représenter :

- les données biométriques d'un individu (nom, âge, poids, ...),
- ou les coordonnées d'une grille (ligne et colonne),
- ou les métadonnées d'un fichier de musique,...

Sans enregistrement

Sans enregistrement, on peut utiliser une classe :

```
public final class BiometricData {
  private final String name;
  private final int height; // in cm
  private final double weight; // in kg
  public BiometricData(String name,
                       int height,
                       double weight) {
    this.name = name;
    this.height = height;
    this.weight = weight;
  }
  public String name() { return name; }
  public int height() { return height; }
  public double weight() { return weight; }
}
```

Avec enregistrement

Les enregistrements simplifient l'écriture de telles classes au minimum :

```
record BiometricData(String name,
                     int height,
                     double weight) {}
```

Cette ligne est équivalente à toute la classe précédente ! (et même un peu plus : `equals`, `toString`, `hashCode`)

Syntaxe générale :

```
record NomDeClasse(descripteur) { ... }
```

(disponible depuis Java 14, mars 2020.)

Définition d'un enregistrement

Le descripteur d'enregistrement est la liste de paramètres (type + identifiant) déclarées à la définition de l'enregistrement.
Les enregistrements sont des classes avec :

- pour chaque paramètre du descripteur :
  - une propriété privée finale de type et d'identifiant identiques au paramètre, appelée composant de l'enregistrement,
  - d'un accesseur public, de même identifiant que le paramètre,
- un constructeur public, le constructeur canonique, ayant les mêmes paramètres que le descripteur,
- des méthodes `hashCode`, `equals`, `toString`, basées sur les propriétés définies par les paramètres.

Déclarations supplémentaires

En plus de ces déclarations par défaut, on peut ajouter dans l'enregistrement :

- une redéfinition du constructeur (le constructeur par défaut copie les arguments dans les propriétés associées), ou des accesseurs,
- des méthodes arbitraires,
- et même des interfaces ou classes internes (peu utile),
- des propriétés statiques et/ou un initialiseur statique (par exemple pour définir des constantes).

Par contre, on ne peut pas ajouter de propriétés d'instance (non-statique).

Exemple (début)

```
record Vector2D(double x, double y) {
  // propriétés x et y, accesseurs x() et y(),
  // constructeur,...

  // propriétés statiques
  public static final Vector2D NULL = new Vector2D(0,0);
  public static final Vector2D[] roots6 = new Vector2D[6];

  // initialiseur statique
  static {
    for (int i = 0; i < 6; i++) {
      root6[i] = polar(1, i * Math.PI / 6);
    }
  }

  ...
```

Exemple (fin)

```
...
  // méthode statique
  public static Vector2D polar(double norm,
                               double angle) {
    return new Vector2D(norm * Math.cos(angle),
                        norm * Math.sin(angle));
  }

  // méthode d'objet
  public Vector2D add(Vector2D addend) {
    return new Vector2D(this.x + addend.x,
                        this.y + addend.y);
  }
}
```

Pourquoi redéfinir le constructeur ?

- Vérifier la validité des valeurs des composants,
- Normaliser les valeurs,
- Réaliser des copies des valeurs de type référence, pour ne pas être perturbé par leur modification ultérieure dans une autre partie du programme.

Redéfinir les accesseurs est rarement pertinent.

Exemple

```
record Rational(int denom, int num) {
  // constructeur redéfinit
  public Rational(int denom, int num) {
    // Validité des valeurs :
    if (num == 0) { throw new ArithmeticException(); }
    // Normalisation :
    int g = gcd(Math.abs(denom), Math.abs(num));
    int sign = ((num >= 0) == (denom > 0)) ? 1 : -1;
    this.denom = sign * Math.abs(denom) / g;
    this.num = Math.abs(num) /g;
  }
  private static int gcd(int a, int b) {
    while (b > 0) {
      remain = a % b;
      a = b;  b = r;
    }
    return a;
  }
}
```

Génériques, interfaces

Les enregistrements peuvent être génériques :

```
record Pair<A,B>(A firstComponent, B secondComponent) {}
```

Les enregistrements peuvent implémenter des interfaces :

```
record Coord(int x, int y) implements Comparable<Coord> {
  public int compareTo(Coord p) {
    return this.x < p.x ? -1:
           this.x > p.x ? 1:
           p.y - this.y;
    }
}
```

À noter :

Les enregistrements :

- peuvent être déclarés avec une visibilité (`public`, `private`,...),
- sont toujours déclarés final, ce qui signifie qu'ils ne peuvent pas être étendus en d'autres classes.
- étendent systématiquement la classe `java.lang.Record`.

Quand utiliser des enregistrements ?

- Si la classe a pour seul rôle d'agréger des données, sans fonctionnalité associée.
- Seulement si la classe n'a pas de raison d'être étendue plus tard.
- Seulement si les objets de la classe sont immuables.
- Les enregistrements sont particulièrement utiles en tant que classes internes, lorsqu'une méthode doit retourner plusieurs informations

Exemple d'utilisation interne

```
public class Matrix {
  ...
  record Coord(int row, int col) {};

  public Coord findZero() {
    for (int row = 0; row < nbRows; row++) {
      for (int col = 0; col < nbCols; col++) {
        if (matrix[row][col] == 0) {
          return new Coord(row,col);
        }
      }
    }
    return null;
  }
```

Conclusion

Les enregistrements sont une simple facilité syntaxique :

- ils permettent de décrire certaines classes avec consision,
- ils sont totalement redondants, on peut utiliser des classes *normales* à la place,
- mais leur concision offre une meilleure lisibilité (autant ou plus d'information en moins de ligne),
- enfin, ils constituent une formalisation de la notion d'agrégation de données (l'équivalent du produit cartésien en mathématiques) et sont donc un bon outil pour la modélisation.

Interfaces scellées

Problèmatique

- Les énumérations permettent de représenter des valeurs d'un ensemble fini (et petit).
- Les enregistrements permettent de représenter l'agrégation de plusieurs valeurs de types prédéfinis, toujours les mêmes.
- Une classe générale est aussi limitée à possèder un ensemble fixe de propriétés.

Solutions adaptées pour représenter des données uniformes, mais comment représenter des données plus hétérogènes ?

Situation

On souhaite représenter des billets de train :

- les billets standards indiquent simplement le trajet (gare de départ et destination),
- les billets avec réservation indiquent en plus une date et un horaire de départ, ainsi qu'un numéro de place,
- les billets illimités permettent de se déplacer sur tout le réseau ferroviaire entre deux dates.

Analyse et solution naïve

Selon le type de billet, on ne veut pas stocker les mêmes informations. Fausse solution : prévoir tous les possibles.

- Une propriété pour chaque information possible,
- une énumération décrivant les trois cas possibles (standard, réservation, illimité),
- une propriété contenant une valeur de l'énumération pour savoir quelles sont les informations actives.

(cf. transparent suivant)

Mauvaise solution

```
public class TrainTicket {
  private Enum Kind { STANDARD, BOOKING, UNLIMITED; }

  private final Kind kind;
  private final Station departure;
  private final Station destination;
  private final DateTime departureTime;
  private final DateTime arrivalTime;
  private final String seat;
  private final Date startingDay;
  private final Date endingDay;

  ...
}
```

Analyse de la mauvaise solution

On s'en sort, avec assez d'effort, cela fonctionne. Mais :

- inefficace en mémoire,
- complexe, ne respecte pas le principe de responsabilité unique des classes (chaque classe ne doit faire qu'une seule chose),
- encourage les erreurs : possibilité de représenter des valeurs qui n'ont pas de sens dans l'application (par exemple un abonnement avec une siège réservé),
- difficile à maintenir à cause de la complexité.

Bonne solution

Utiliser une interface scellée, permettant de représenter une alternative entre plusieurs possibilités :

- une classe pour chaque possibilité : standard, avec réservation, illimité,
- une interface scellée englobant les trois possibilités sous un seul type.

Réalisation en Java

```
public sealed interface TrainTicket
  permits StandardTicket, BookingTicket, UnlimitedTicket
  {}

record StandardTicket(Station departure,
                      Station destination) {}

record BookingTicket(
  Station departure, Station destination,
  String seat,
  DateTime departureTime, DateTime arrivalTime
) {}

record UnlimitedTicket(Date startingDay,
                       Date endingDay) {}
```

Usage

```
TrainTicket aubagneMarseille =
  new Standard(aubagneStation, marseilleStCharlesStation);
TrainTicket parisMarseille =
  new BookingTicket(
    parisGareDeLyonStation,
    marseilleSaintCharlesStation,
    "Voiture 8 place 42",
    DateTime.of(2024,Month.OCTOBER,15,14,0),
    DateTime.of(2024,Month.OCTOBER,15,17,15)
  );
 System.out.println(parisMarseille.toString());
```

Quelques remarques

- `TrainTicket` étant une interface, tout valeur de type `TrainTicket` est une des implémentations de `TrainTicket`.
- Le scellement n'autorise comme implémentation que celles listées dans la clause `permits` : toute valeur de type `TrainTicket` est donc instance de `StandardTicket`, `BookingTicket`, ou `UnlimitedTicket`.

Quelques remarques

- Les trois implémentations sont des enregistrements, mais on aurait aussi bien pu utiliser des classes standards ou des énumérations.
- Les trois implémentations sont dans le même fichier `TrainTicket.java`, mais on peut aussi leur attribuer un fichier à chacune.
- `TrainTicket` peut contenir des méthodes par défaut, valables pour toutes les implémentations, et chaque implémentation peut avoir des méthodes qui lui sont propres.

Contraintes sur les classes permises.

Toutes les classes permises doivent être définies !
On peut utiliser :

- des enregistrements,
- des classes non-extensibles, déclarées avec `final` :

```
public final class StandardTicket
    implements TrainTicket {...}
```

- des interfaces scellées, créant une hiérarchie de types,
- des classes/interfaces non-scellées, déclarées avec `non-sealed`, pouvant elle-même avoir une nombre arbitraire d'extensions/implémentations.

```
public non-sealed class StandardTicket
    implements TrainTicket {...}
```

Difficulté

On souhaite écrire une méthode qui, étant donné un ticket de train et une gare, détermine si c'est bien la gare de départ du ticket.

```
public static boolean isDeparture(TrainTicket ticket,
                                  Station station) {
  return ticket.departure().equals(station); // erreur !
}
```

Ne compile pas, car l'interface `TrainTicket` n'a pas de méthode `departure()` !
Seuls `StandardTicket` et `BookingTicket` ont une telle méthode.

Utilisation du `switch`

La structure de contrôle `switch` permet de retrouver parmi les classes permises, laquelle correspond à une instance :

```
public static boolean isDeparture(TrainTicket ticket,
                                  Station station) {
  switch (ticket) {
  case StandardTicket stdTicket :
    return stdTicket.departure().equals(station);
  case BookingTicket bkgTicket :
    return bkgTicket.departure().equals(station);
  case UnlimitedTicket ultdTicket :
    return true; // any station is a possible departure
  }
}
```

Utilisation du `switch`

On peut aussi utiliser la forme expressive du `switch` :

```
public static boolean isDeparture(TrainTicket ticket,
                                  Station station) {
  return switch (ticket) {
    case StandardTicket stdTicket ->
      stdTicket.departure().equals(station);
    case BookingTicket bkgTicket ->
      bkgTicket.departure().equals(station);
    case UnlimitedTicket ultdTicket ->
      true; // any station is a possible departure
  };
}
```

Prenez garde aux différences syntaxiques entre les deux versions !

Conclusion

On peut former des données complexes :

- par agrégation de données plus simples (classes standards, enregistrement),
- par alternative entre données plus simple (énumération, interfaces scellées).

On accède à ces données :

- par des accesseurs dans les agrégations,
- par le `switch` dans les alternatives.

C'est le fait de sceller l'interface qui permet l'usage du `switch`, le nombre de cas étant alors déterminé.

Le sélecteur `switch`

Le sélecteur instructionnel sur les entiers

Structure de contrôle `switch`

La structure de contrôle `switch` permet de choisir entre plusieurs alternatives, selon une valeur entière :

```
switch (value) {
  case 1 : return "one";
  case 2 : return "two";
  case 3 : return "three";
  case 4 : return "four";
  default : return "many";
}
```

Règles du `switch`

```
switch (expression) {
  ...
  case 42 :
  ...
}
```

- L'expression sélecteur évaluée par le `switch`, figurant entre parenthèses, doit être de type entier (ou caractère).
- Les `case` doivent être suivis d'un littéral obligatoirement (un entier écrit en chiffre, comme `42`), jamais par une expression arbitraire (comme `2 * x + 1`).
- L'expression sélecteur est évaluée, puis l'instruction suivant le `case` correspondant à sa valeur est exécutée,
- puis toutes les instructions suivantes, une par une (même après d'autres `case`).

Règles du `switch`

```
switch (expression) {
  ...
  case 42 :
  ...
  default :
  ...
}
```

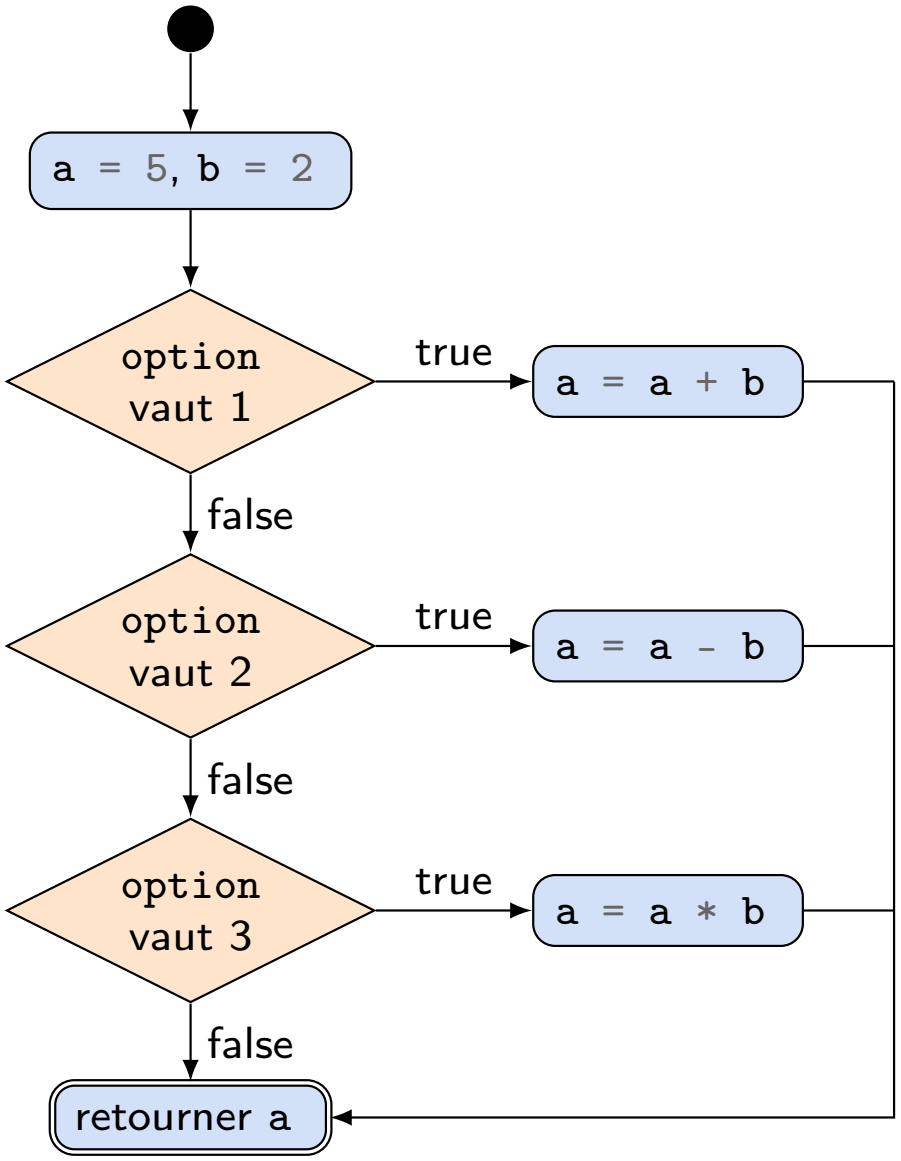
- On utilise `break` (ou `return`) pour quitter le bloc du `switch`.
- Si aucun `case` ne correspond à la valeur évaluée, `default` est choisi s'il existe, sinon l'exécution quitte le bloc du `switch`.

Exemple switch

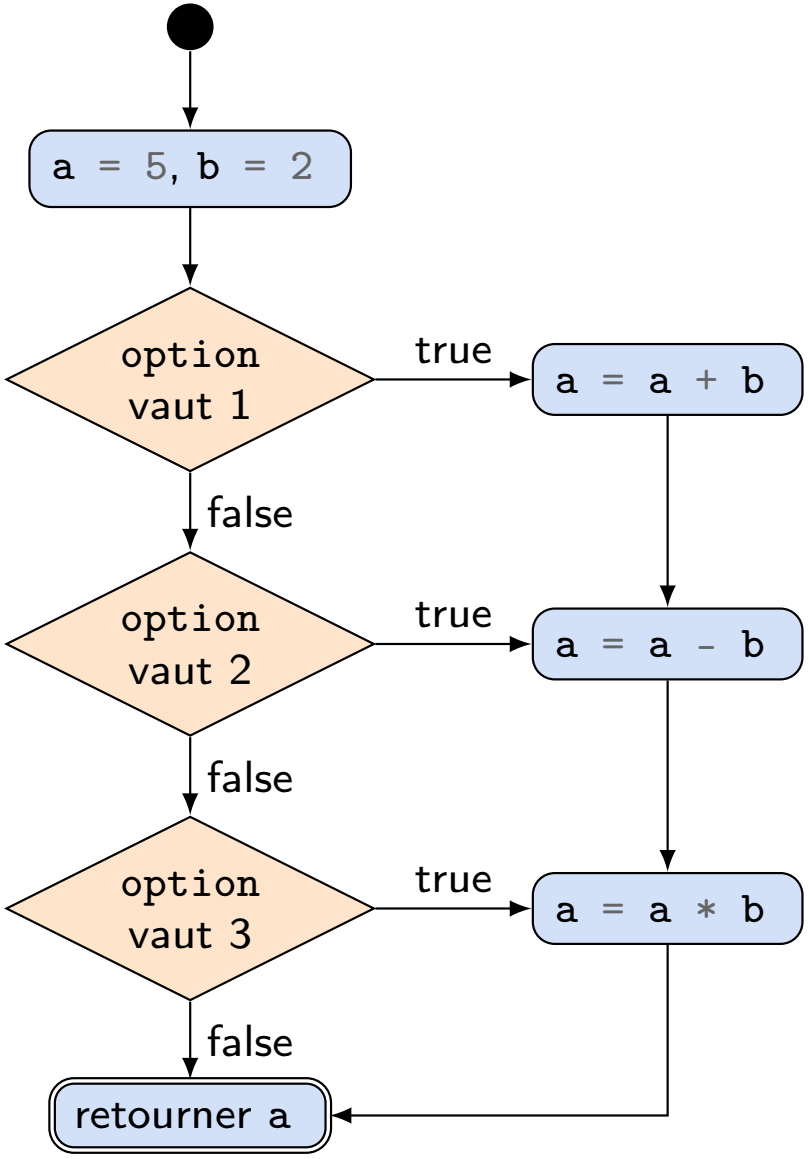
```
public int chooseOperation(int option) {
  int a = 5;
  int b = 2;
  switch (option) {
    case 1 :
      a = a + b;
      break;
    case 2 :
      a = a - b;
      break;
    case 3 :
      a = a * b;
      break;
  }
  return a;
}
```

option	1	2	3	4
résultat	7	3	10	5

Organigramme du `switch`



Le même programme sans les `break`



Exemple `switch`

Exercice

Ouvrez votre IDE pour Java et essayer le programme précédent. Essayez aussi en enlevant les instructions `break`. Vérifiez que les résultats correspondent à votre attente.

Extension aux enums, String

On peut utiliser le `switch` pour décider selon une valeur d'un type énumération, ou plus rarement du type `String`.

```
public enum Flavor { VANILLA, CHOCOLATE; }
...
switch (icecream.flavor()) {
  case Flavor.VANILLA : return "vanille";
  case Flavor.CHOCOLATE : return "chocolat";
}
switch (text) {
  case "foo" :
  case "bar" : break;
  default :
   System.out.println(text);
 }
```

Le sélecteur instructionnel sur les interfaces scellées

Exemple de `switch` avec interface scellée

```
sealed interface Shape permits Square, Circle { ... }
...
public double area(Shape shape) {
  switch (shape) {
  case Square sq :
    double side = sq.side();
    return side * side;
  case Circle circ :
    double radius = circ.radius();
    return Math.PI * radius * radius;
  }
}
```

`switch` et interface scellée

- Le `switch` permet de distinguer une référence selon chaque implémentation de l'interface scellée.
- La référence testée est donnée par l'expression sélecteur,
- Dans chaque cas (exemple : `case Square sq`) :
  - on déclare une nouvelle variable (`sq`),
  - du type d'une des implémentations (`Square`),
  - contenant la référence testée (`sq == shape`),
  - on donne des instructions pour gérer ce cas.

Test d'exhaustivité

Erreur de compilation si on n'a pas pris en compte tous les cas possibles (ou utilisé un cas `default`).

```
sealed interface Shape permits Square, Circle, Triangle
  { ... }
...
public double area(Shape shape) {
  switch (shape) {
  case Square sq :
    double side = sq.side();
    return side * side;
  case Circle circ :
    double radius = circ.radius();
    return Math.PI * radius * radius;
  } // Erreur : manque le cas Triangle
}
```

Avantage des interfaces scellées

Grâce au test d'exhaustivité, l'ajout d'un cas (par exemple `Triangle`) rend invalide tous les `switch` déjà écrits.

C'est un point positif ! Le compilateur nous avertit des méthodes à modifier lors de l'ajout d'un nouveau cas, plutôt que nous laisser faire des erreurs.

Conséquence : éviter `default` pour les `switch` sur interfaces scellés.

Test de redondance

Le compilateur interdit aussi d'écrire des cas inutiles (car *dominé* par un cas précédent).

```
sealed interface Polygon
    permits Triangle, Quadrilateral { ... }
sealed interface Quadrilateral extends Polygon
    permits Square, Rectangle { ... }
...
public int nbSides(Polygon poly) {
  switch (poly) {
  case Quadrilateral quad : return 4;
  case Triangle tri : return 3;
  case Square sq : return 4;
   // Erreur : Square est dominé par Quadrilateral
}
```

Sélecteur sur classes arbitraires

Le `switch` peut s'utiliser plus généralement :

```
switch (obj) { // obj de type Object
  case null : System.out.println("null"); break;
  case String s : System.out.println("String"); break;
  case Color c : System.out.println("Color"); break
  case Point p : System.out.println("Point"); break
  case int[] ia : System.out.println("Array"); break
  default : System.out.println("Something else");
}
```

Déconseillé ! Signe de programme mal structuré (*code smell*).
Se restreindre à la sélection de classes/interfaces scellées.

Le sélecteur expressionnel

Structure de contrôle *vs* expression

Souvenons-nous de l'expression conditionnelle :

```
return isBlue ? "blue" : "red";
```

et la structure de contrôle `if` :

```
if (isBlue) { return "blue"; } else { return "red";}
```

- `...?...:...` est un opérateur définissant des expressions.
- `if (...) { ... } else { ... }` est une structure de contrôle, contrôlant l'exécution des instructions.

Deux formes de `switch`

On a vu le `switch` comme structure de contrôle.
Il existe aussi une forme expressionnelle :

```
sealed interface Shape permits Square, Circle { ... }
...
public double area(Shape shape) {
  return switch (shape) {
  case Square sq ->
    sq.side() * sq.side();
  case Circle circ ->
    Math.PI * circ.radius() * circ.radius();
  };
}
```

Remarques

- Différence syntaxique : `->` plutôt que `:`
- En terme droit de `->`, on donne une expression, et non pas des instructions.
- Pas besoin de `break`.
- Possible d'utiliser `default` (à éviter néanmoins).

Où utiliser le sélecteur expressionnel ?

N'importe où où une expression peut être utilisée !

- en membre droit d'une affectation,

```
double area = switch (shape) { ... };
```

- en expression d'un `return`,
- en argument d'un appel de méthode,
- dans une expression plus large,
- en condition d'un `while`, `for`, `if`, en expression sélecteur d'un autre `switch`.

Dans les trois derniers cas : risque de rendre la méthode difficile à lire (donc à éviter). Utiliser une variable intermédiaire.

Le mot réservé `yield`

Que faire si dans un cas, l'expression résultat demande plusieurs instructions pour se calculer ?

Solution 1 : définir une méthode et y faire appel.
Solution 2 : utiliser `yield` dans un bloc d'instructions :

```
double area =
  switch (shape) {
  case Square sq -> sq.side() * sq.side();
  case Circle circ -> {
    double radius = circ.radius;
    yield Math.PI * radius * radius;
  };
```

$\Longrightarrow$ comme un `return` limité au sélecteur.

Partie 6

Interfaces et classes abstraites

Rappels sur les interfaces

Objets, classes, interfaces

- Les objets ont des propriétés et des comportements.
- Les classes regroupent des objets ayant :
  - des propriétés de même nature, mais pas de même valeur,
  - des comportements de même nature et de même valeur.
- les interfaces regroupent des objets ayant :
  - des comportements de même nature, mais différents.

Exemple des recyclables (I)

Imaginons les classes :

- Papier,
- Canette,
- Bouteille,
- Carton

Toutes ont des comportements différents, s'utilisent dans des contextes différents.
Mais aussi toutes sont recyclables et vont donc dans la poubelle des objets recyclables.

Exemple des recyclables (II)

Papier, Canette, Bouteille, Carton

- possèdent une méthode `recycle` (qui n'ont pas le même comportement, on ne recycle pas le verre comme le papier),
- possèdent aussi d'autres méthodes propres à chaque classe.

La Poubelle peut accepter n'importe quel objet recyclable.

Exemple des recyclables (II)

```
// class RecycleBin
private Set<???> trash;

public void empty() {
  for (??? item : trash) {
    item.recycle();
  }
}
```

Quel type pour `trash` ? pour les éléments de `trash` ?

Collections en Java

Les collections en Java sont homogènes : tous les éléments contenus dans la collection sont du même type.

- parce que c'est plus simple !
- parce que l'expérience montre que c'est ce qu'on veut quasiment toujours,
- parce que sinon on ne sait pas comment utiliser un élément issu de la collection (on ne sait pas sa nature).

Donc `trash` doit être une collection d'un type bien précis. Lequel ?

Interface

Interface

Une *interface* décrit un ensemble de fonctionnalités communes à plusieurs classes.

Pour nos objets, la fonctionnalité commune est la recyclabilité. *Recyclable* est une interface !

- les classes en question sont potentiellement très différentes (papier vs canette),
- les fonctionnalités communes ne sont pas forcément réalisées de la même façon (le recyclage du papier est différent du recyclage de l'aluminium).

Définition d'interface

Une interface

Une *interface* est la description d'une liste de services, sans préciser comment le service est accompli (sans implémentation).

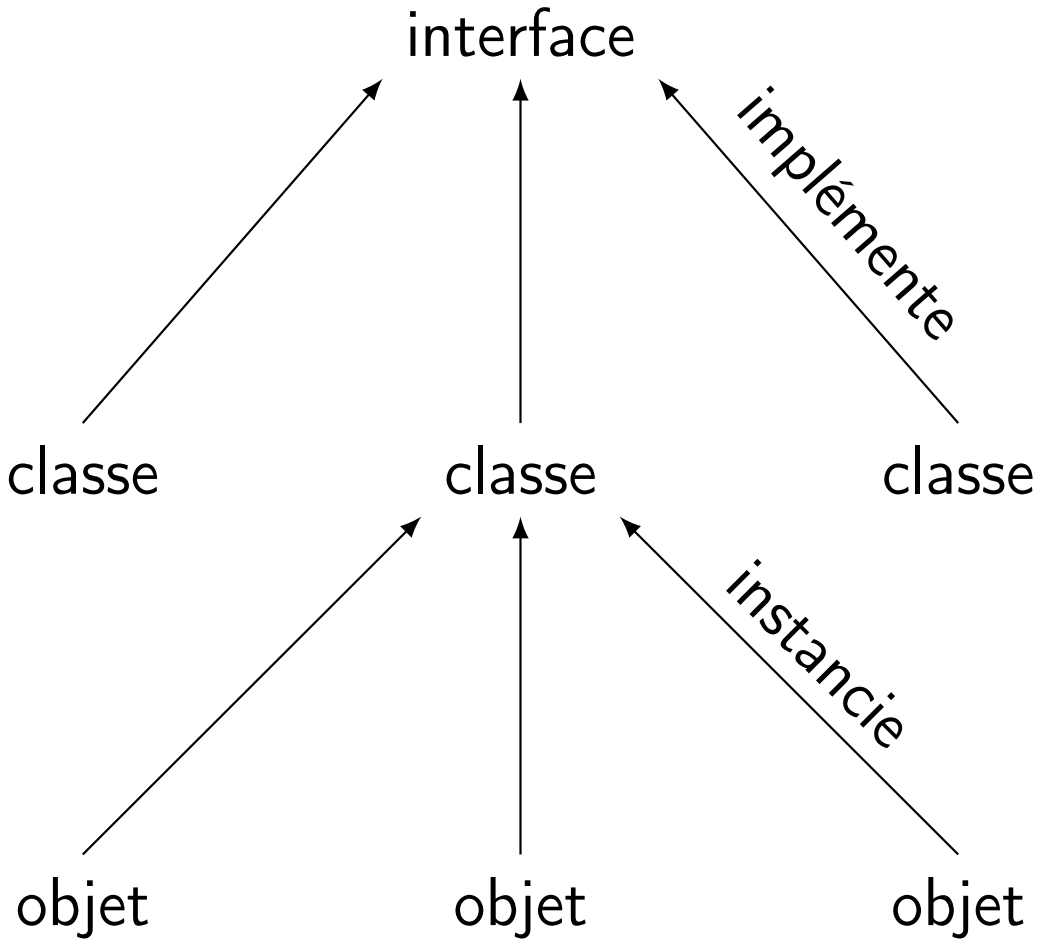
- La définition d'interface ne contient pas d'instructions, seulement des déclarations.
- Une interface peut déclarer les en-têtes d'une ou plusieurs méthodes.
- Les propriétés n'apparaissent pas dans les interfaces.

Un niveau d'abstraction supplémentaire

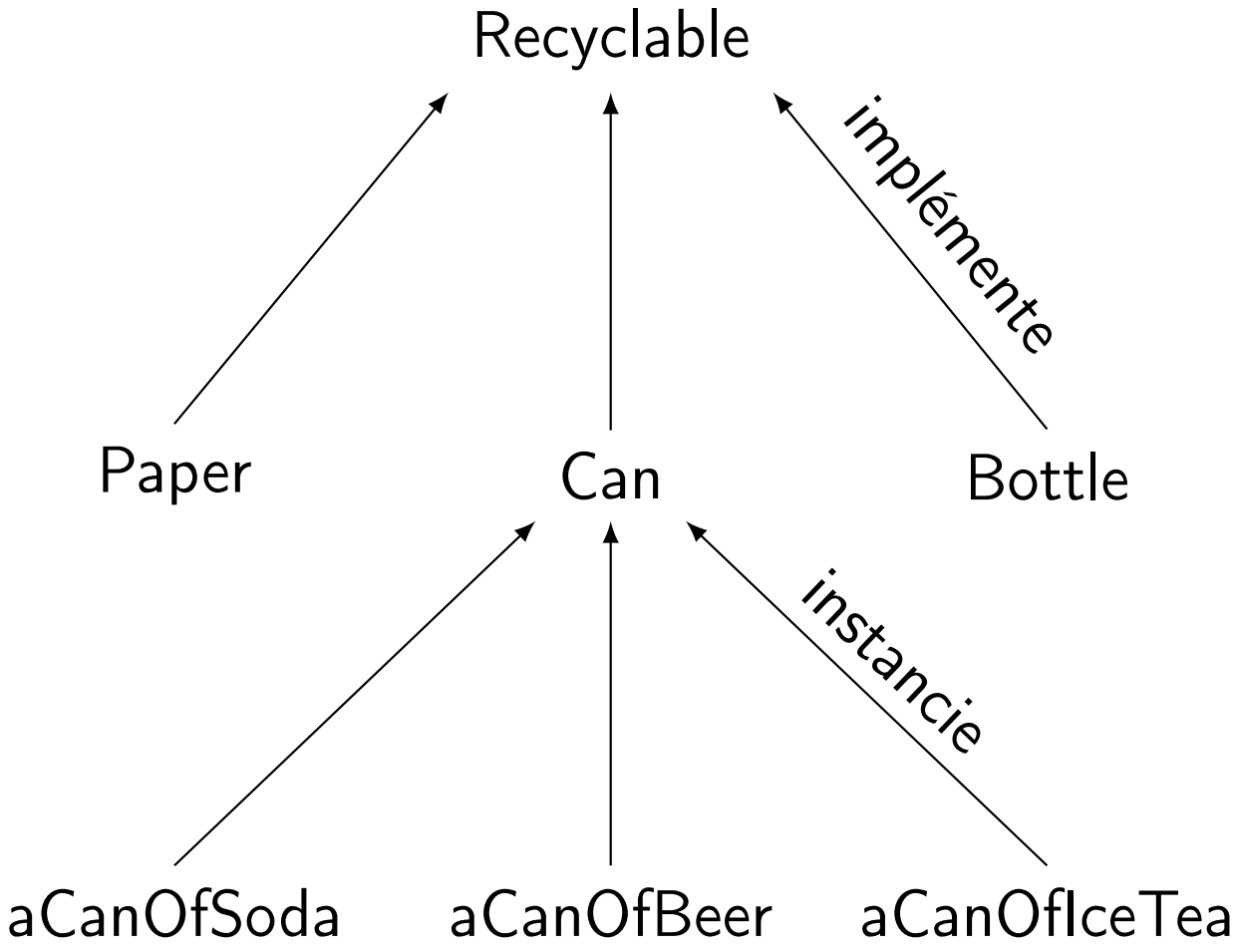
```
public interface Recyclable {
  void recycle();
}
```

Avoir l'interface `Recyclable`, c'est posséder une méthode publique `void recycle()`.

Un niveau d'abstraction supplémentaire



Un niveau d'abstraction supplémentaire



Implémentation d'une interface

Implémenter une interface

Une classe (ou un objet) *implémente* une interface si toute méthode déclarée dans l'interface est déclarée et programmée (implémentée) dans la classe.

- Pour implémenter une méthode d'une interface, la méthode de la classe doit avoir le même nom, les mêmes paramètres (de même types), le même type de retour.
- Une classe peut implémenter des méthodes en plus par rapport à l'interface.
- Une classe peut implémenter plusieurs interfaces différentes.

Implémenter une interface en Java

Pour qu'une classe soit une implémentation d'une interface en Java, il faut :

- le signaler dans la déclaration de classe, avec le mot réservé `implements`,
- implémenter chaque méthode de l'interface, avec visibilité publique.
- Pour le reste, on ajoute ce qu'on veut (propriétés, constructeurs, méthodes).

Sous IntelliJ : faire d'abord le signalement dans la déclaration de classe, puis `Alt+Entrée` pour ajouter les déclarations de méthodes.

Interfaces et types

Types

Toute interface définit un *type*.

- N'importe quel paramètre, variable, propriété, peut être déclaré de ce type.
- Le type de retour d'une méthode peut être une interface.
- N'importe quel objet `obj` instanciant une classe `ClassName` implémentant une interface `InterfaceName` a les deux types `ClassName` **et** `InterfaceName`.

Exemples d'interface comme type (I)

```
Paper item = new Paper(); // OK
Recyclable item = new Paper(); // OK
Paper item = new Recyclable(); // NO
Recyclable item = new Recyclable(); // NO
Paper item = new Can(); // NO
```

- `Paper` implémente `Recyclable` : ligne 2 OK.
- `Recyclable` est une interface, une interface n'est pas instanciable : lignes 3 et 4 incorrectes.
- `Paper` et `Can` implémentent `Recyclable`, mais sont deux classes distinctes qu'on ne peut confondre : ligne 5 incorrecte.

Exemples d'interface comme type (II)

```
ArrayList<Integer> myArrayList = new ArrayList<>();
myArrayList.addAll(List.of(1,2,3,4,5));
Collections.sort(myArrayList); // OK
Set<Recyclable> trash = new HashSet<>();
trash.add(new Paper()); // OK
trash.add(new Can()); // OK
```

- `void sort(List<Integer> list)` attend une `List<Integer>`, `ArrayList<Integer>` implémente `List<Integer>` : ligne 3 OK.
- `trash` est un ensemble de `Recyclable`, `boolean add(Recyclable)` attend un recyclable, `Paper` et `Can` implémentent `Recyclable` : ligne 5 et 6 OK.

Pause

Important

Prenez le temps de bien comprendre les deux transparents précédents !

Solution pour le recyclage

```
public interface Recyclable {
  void recycle();
}

public class Bottle implements Recyclable {
  ...
  public void recycle() { ... }
}
```

trash est de type Set<Recyclable>.

Interfaces et méthodes

Supposons que `myValue` contient une instance de la classe `MyClass` implémentant l'interface `MyInterface`.

Si `myValue` est déclarée par `MyClass myValue` :

- on peut utiliser les méthodes de `MyClass`,
- on peut utiliser les méthodes de `MyInterface`.

Si `myValue` est déclarée par `MyInterface myValue` :

- on peut utiliser les méthodes de `MyInterface`,
- on ne peut pas utiliser les autres méthodes de `MyClass`.

Exemples de méthodes accessibles

```
Paper paperC = new Paper();
Recyclable paperI = new Paper();
paperC.recycle(); // OK
paperC.tear(); // OK
paperI.recycle(); // OK
paperI.tear(); // NO
```

Contraindre le type de `paperI` interdit l'utilisation d'autres méthodes que celles de l'interface.

Types et méthodes

Ainsi, la règle est :

Typage

C'est le type déclaré de la variable qui détermine quelles méthodes peuvent être appelées.

Les contraintes de typage nous aide, en rendant impossible l'appel d'une méthode d'un objet n'ayant pas cette méthode.

À quoi servent les interfaces ?

Une interface ne contient pas d'implémentation. Il s'agit donc uniquement d'une définition de type.

- Pour définir un contrat (l'objet doit avoir telles méthodes),
- Pour manipuler simultanément des objets de différentes classes mais ayant les mêmes services (tout objet avec ces méthodes convient),
- Pour réduire les dépendances (je pourrais changer pour un autre objet avec les mêmes méthodes plus tard).

Mise en situation

Présentation

Les deux prochains transparents montrent deux classes très similaires.

Principe DRY : *Don't Repeat Yourself*, éviter les duplications dans le programme.

Problématique : comment réorganiser le programme pour éviter toute duplication ?

Exemple (I)

```
public class SumList {
  List<Integer> ints;

  public SumList(List<Integer> ints) {
    this.ints = ints;
  }

  public int eval() {
    int sum = 0;
    for (Integer i : ints) {
      sum = sum + i;
    }
    return sum;
  }
}
```

Exemple (II)

```
public class ProductList {
  List<Integer> ints;

  public ProductList(List<Integer> ints) {
    this.ints = ints;
  }

  public int eval() {
    int product = 1;
    for (Integer i : ints) {
      product = product * i;
    }
    return product;
  }
}
```

Analyse préliminaire

- même propriété, les méthodes diffèrent dans l'implémentation uniquement.
- les méthodes n'ont pas tout-à-fait les mêmes instructions : impossible de génératliser en ajouter des propriétés à l'objet.

Deux solutions :

- par délégation (interfaces),
- par extensions (classes abstraites).

Interfaces et délégation

Factorisation, étape 1

```
public class SumList {
  List<Integer> ints;

  public SumList(List<integer> ints) {
    this.ints = ints;
  }

  public int eval() {
    int value = neutral();
    for (Integer i : ints) {
      value = operator(value,i);
    }
    return value;
  }

  private int operator(int a, int b) { return a + b; }
  private int neutral() { return 0; }
}
```

Factorisation : délégation

```
public class SumList {
  List<Integer> ints;
  private IntMonoid monoid;

  public SumList(List<integer> ints, IntMonoid monoid) {
    this.ints = ints;
    this.monoid = monoid;
  }

  public int eval() {
    int value = monoid.neutral();
    for (Integer i : ints) {
      value = monoid.operator(value,i);
    }
    return value;
  }
}
```

Factorisation : délégation (usage)

```
public interface Monoid<T> {
  T neutral();
  T operator(T a, T b);
}

public class SumMonoid implements Monoid<Integer> {
  public int neutral() { return 0; }
  public int operator(int a, int b) { return a + b; }
}

SumList list = new SumList(ints,new SumMonoid());
int sum = list.eval();
```

Délégation

Délégation

Une *délégation* consiste pour un objet déléguant à faire faire un travail par un autre objet délégué, propriété du déléguant.

Intérêt :

- garder la classe simple en exportant une partie du travail dans une autre classe (chaque classe a une seul rôle),
- permettre de paramètrer la classe selon le délégué. C'est donc une technique de factorisation.
- on a ainsi une classe pour les parties communes, et une classe pour chaque comportement distinct.

Résolution de l'exemple

Dans notre exemple, la délégation porte sur la partie qui différe entre les deux classes :

- la valeur initiale dans la sommation,
- l'opérateur arithmétique utilisé.

Le délégué est le monoïde possédant une valeur et un opérateur.

N'importe quel bloc d'instruction peut être délégué à un autre objet, c'est donc une solution flexible.

Classes abstraites et extensions

Deuxième solution : classe abstraite

Idée similaire : une classe avec la partie commune, des classes pour chaque comportement distinct.

Réalisation :

- faire une classe *à trous*, certaines méthodes restent sans implémentation (classe abstraite).
- faire des classes bouchant les trous : on implémente uniquement les méthodes qui ne sont pas implémentées par la classe abstraite (extension).
- la même classe à trou peut être complétée de multiples fois (plusieurs extensions)

Exemple de classe abstraite

```
public abstract class OperatorList {
  public abstract int neutral();
  public abstract int operator(int a, int b);

  protected List<Integer> ints;

  public OperatorList(List<Integer> ints) {
    this.ints = ints;
  }

  public int eval() {
    int value = neutral();
    for (int i : ints) {
      value = operator(value,i);
    }
    return value;
  }
}
```

Exemple d'extension

```
public class SumList extends OperatorList {
  public int neutral() { return 0; }
  public int operator(int a, int b) { return a + b; }

  public SumList(List<Integer> ints) {
   super(ints);
  }
}

public class ProductList extends OperatorList {
  public int neutral() { return 1; }
  public int operator(int a, int b) { return a * b; }

  public ProductList(List<Integer> ints) {
   super(ints);
  }
}
```

Déclaration, super, extension

Une classe abstraite est définie avec `abstract` :

```
public abstract class MyAbstractClass { ... }
```

Une extension est définie avec `extends` :

```
public class MyConcreteClass extends MyAbstractClass {
   ...
}
```

`MyAbstractClass` est alors la `super`classe de `MyConcreteClass`. `MyConcreteClass` est une *extension* (ou *sous-classe*) de `MyAbstractClass`.

Méthodes abstraites ou concrètes

Une classe abstraite peut contenir des méthodes concrètes et des méthodes abstraites :

```
public abstract MyReturnType myMethod(MyArgType arg);
```

Une méthode abstraite est non-implémenté, son corps est remplacé par un “ ;”

Une extension soit est elle-même abstraite, soit implémente chaque méthode abstraite de la superclasse.

```
public MyReturnType myMethod(MyArgType arg) { ... }
```

Une extension *hérite* des méthodes concrètes de la superclasse.

Classes abstraites et propriétés

Une classe abstraite peut avoir des propriétés, commune à toutes les extensions.

Une propriété privée d'une classe abstraite **est inaccessible** depuis les extensions.

Pour avoir une propriété accessible par les extensions, mais pas par toutes les classes, on utilise la visibilité `protected`.

`protected` : visible par la classe, les extensions et les classes du même `package`. Invisible pour toute autre classe.
`protected` peut aussi s'appliquer aux méthodes, constructeurs, ...

Classes abstraites et constructeurs

Une classe abstraite **ne peut pas** être instanciée.

Une classe abstraite **peut** néanmoins posséder un constructeur :

- pour factoriser une partie de la construction du nouvel objet, quelque soit l'extension.
- les extensions doivent aussi avoir un constructeur. Le constructeur de l'extension **doit** commencer par un appel au constructeur de la classe abstraite, **s'il existe**, pour initialiser l'objet. Mot-clé `super`. Par défaut le constructeur de `super` sans argument est appelé.

```
public SumList(List<Integer> ints) {
 super(ints);
}
```

Un autre exemple

Exemple : les listes revisitées (I)

Interface d'une liste :

```
public interface LinkedList<E> implements Iterable<E> {

  boolean isEmpty();

  E head();
  LinkedList<E> tail();

  int length();
  LinkedList<E> add(E elt);
}
```

Exemple : les listes revisitées (II)

Une méthode non-implémentée déclarée dans une interface est considérée abstraite, on pourrait donc ne pas mentionner `isEmpty`, `head`,...

```
public abstract class AbstractList<E>
    implements LinkedList<E> {

  public abstract boolean isEmpty();
  public abstract E head();
  public abstract AbstractList<E> tail();
  public abstract int length();

  public AbstractList<E> add(E elt) {
    return new NonEmptyList<E>(elt,this);
  }
  public Iterator<E> iterator() {
    return new ListIterator<E>(this);
  }
}
```

Exemple : les listes revisitées (III)

`iterator` et `add` sont héritées.

Pas besoin de constructeur : `List` n'en a pas, pas de `super()` à faire.

```
public class EmptyList<E> extends List<E> {

  public boolean isEmpty() { return true; }

  public E head() {
    throw new NoSuchElementException();
  }
  public AbstractList<E> tail() {
    throw new NoSuchElementException();
  }

  public int length() { return 0; }
}
```

Exemple : les listes revisitées (IV)

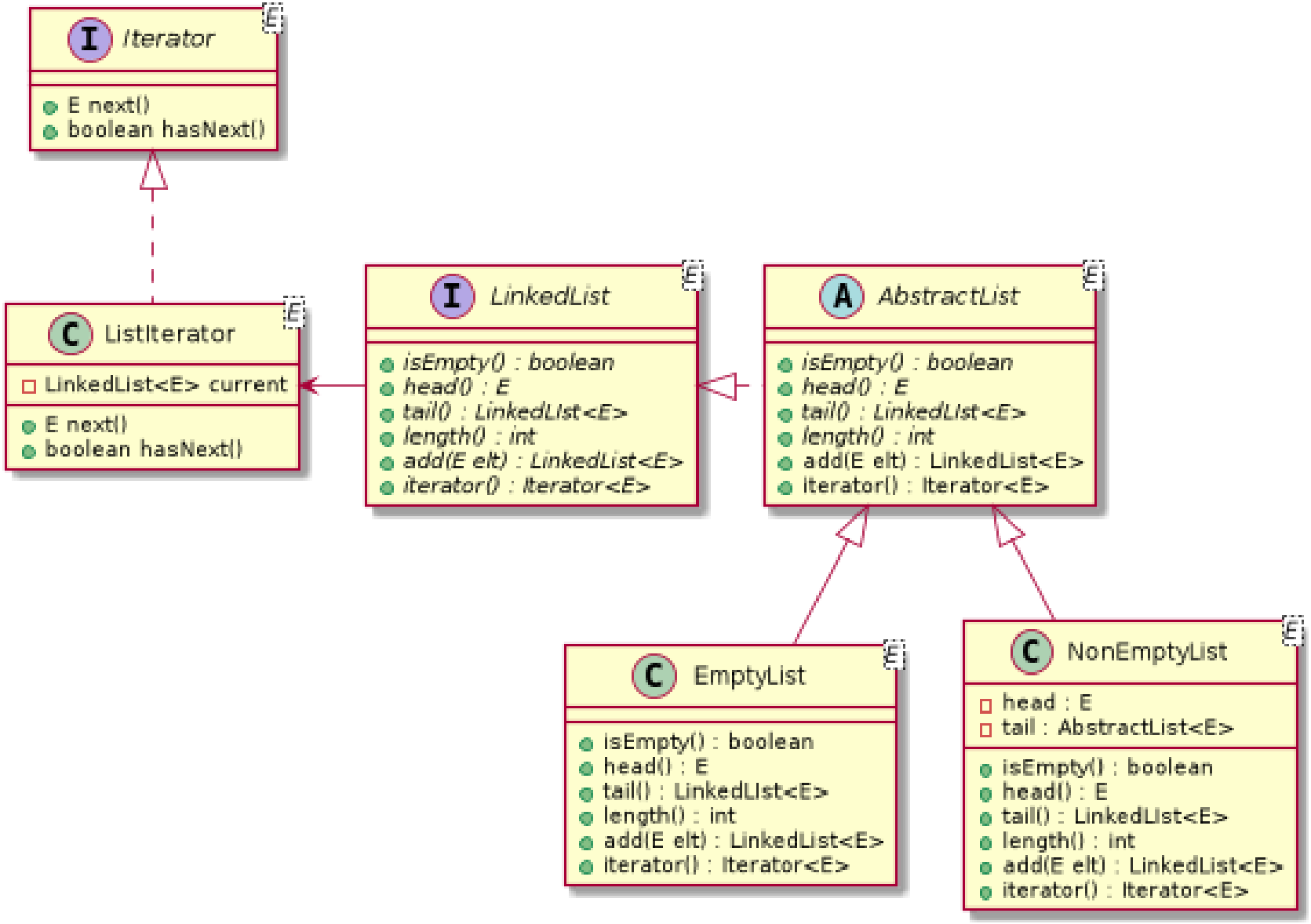
Ici non plus, pas de `super()` dans le constructeur. `iterator` et add sont hérités.

```
public class NonEmptyList<E> extends List<E> {
  private final E head;
  private final AbstractList<E> tail;

  public NonEmptyList(E head, AbstractList<E> tail) {
    this.head = head;
    this.tail = tail;
  }
  public boolean isEmpty() { return false; }
  public E head() { return head; }
  public AbstractList<E> tail() { return tail; }

  public int length() {
    return 1 + tail.length();
  }
}
```

Exemple : les listes revisitées (V)

```
public class ListIterator<E> implements Iterator<E> {
  private LinkedList<E> list;

  public ListIterator(LinkedList<E> list) {
    this.list = list;
  }
  public E next() {
    E current = list.head();
    list = list.tail();
    return current;
  }
  public boolean hasNext() {
    return !list.isEmpty();
  }
}
```

Diagramme de classes



Redéfinition de méthodes

L'extension peut redéfinir les méthodes implémentées de la classe abstraite (même arguments, même retour). Un objet de la classe étendue utilisera par défaut sa propre méthode (même dans un contexte où il possède le type de la classe abstraite).

```
abstract class A {
  public void foo() { System.out.println("A.foo"); }
}
class B extends A {
  public void foo() { System.out.println("B.foo"); }
}

B b = new B();
b.foo(); // "B.foo"
A a = b;
a.foo(); // "B.foo"
```

Redéfinition de méthode et super

L'extension peut faire appel à une méthode de sa superclasse avec le mot-clé super.

```
abstract class A {
  public void foo() { System.out.println "A.foo"; }
}
class B extends A {
  public void foo() {
    super.foo();
    System.out.println("B.foo");
  }
}

B b = new B();
b.foo(); // "A.foo", "B.foo"
```

Détermination de la méthode

Une variable `a` de type `A` peut contenir un objet instance d'une classe `C`, qui étend une classe `B`. Quelle méthode est appelée par `a.m()` ?

- l'instance est stockée dans le tas,
- l'emplacement dans le tas contient une référence sur la table des méthodes de la classe d'instanciation,
- la méthode est donc celle de la classe dont l'objet est une instance (celle du `new`).
- `super` permet de remonter à la table de la classe parent.

Deux règles différentes

Comparez :

Typage

C'est le type déclaré de la variable qui détermine quelles méthodes **peuvent être appelées**.

avec :

Polymorphisme

C'est la classe d'instanciation de l'objet qui détermine quelles méthodes **sont exécutées**.

Extensions générales

Les classes non-abstraites peuvent aussi être étendues :

- hérite des méthodes et propriétés de la superclasse,
- possède des nouvelles fonctionnalités (méthodes et propriétés),
- redéfinit des méthodes de la superclasse avec d'autres comportements.

Limitations :

- en Java, on ne peut étendre qu'une seule classe,
- on peut déclarer une classe finale, ce qui interdit de l'étendre,
- les extensions peuvent être à l'origine de problèmes subtils.

La classe Object

En Java, toute classe étend la classe Object.

```
boolean equals(Object obj);
int hashCode();
String toString();
...
```

```
Point point = new Point(3,2);
System.out.println(point.toString());
  // --> "test.Point@8c24e1"
System.out.println(point);
  // --> "test.Point@8c24e1"
```

Méthodes par défaut

Interface et méthodes par défaut

Une classe abstraite peut avoir des méthodes implémentées (non abstraite).

Une interface aussi peut avoir des méthodes implémentées : des méthodes par défaut (`default`).

- Une interface n'a pas de propriété (sauf constantes).
- Les méthodes par défaut peuvent appeler les méthodes non-implémentées de l'interface.
- Les méthodes par défaut sont signalées avec le mot-clé `default`.
- Une implémentation peut redéfinir les méthodes par défaut.

Méthode par défaut, exemple

```
public interface Monoid<E> {
  E neutral();
  E operator(E value1, E value2);

  default E reduce(List<E> values) {
    E accum = neutral();
    for (E value : values) {
      accum = operator(accum, value);
    }
    return accum;
  }
}
```

Redéfinition de méthode par défaut

```
public class ConjonctionMonoid
    implements Monoid<Boolean>
{
  public boolean neutral() { return true; }
  public boolean operator(boolean b1, boolean b2) {
    return b1 && b2;
  }

  public boolean reduce(List<Boolean> bools) {
    for (Boolean b : bools) {
      if (!b) return false;
    }
    return true;
  }
}
```

Délégation ou extension

Récapitulatif

- Classe abstraite : classe partiellement implémentée.
- Les classes abstraites sont une technique permettant de factoriser des parties communes à plusieurs classes.
- Extension : classe fabriquée à partir d'une autre classe par ajout de propriétés et de méthodes.
- Toute classe est une extension (sauf `Object`).
- L'extension redéfinit les méthodes, masque les propriétés du parent.
- La méthode exécutée est celle de la classe instanciée par `new`.
- syntaxe : `abstract`, `@Override`, `extends`, `super`, `protected`.

Classe abstraite *vs* interface

Classe abstraite :

- extension unique : chaque classe étend une seule autre classe.
- méthodes abstraites,
- possède des propriétés,
- visibilités variées (`protected`, `private`,...).

Interface :

- implémentation multiple : une classe implémente autant d'interfaces que nécessaire.
- méthodes par défaut,
- pas de propriétés (sauf constantes, implicitement `public static final`),
- tout est public.

Classe abstraite *vs* interface

Que choisir entre classe abstraite et interface (et délégation) ?

- une classe abstraite correspond à une responsabilité (je suis ...),
- une interface correspond à une capacité (je peux ...).

Une *Renault twingo* est une *voiture* et peut être *conduite* :

- *twingo* est une classe,
- *voiture* est une classe abstraite,
- *conduisible* est une interface.

Classe abstraite *vs* interface

Principe empirique : *favor composition over inheritance*

- composition : un objet possèdant des instances (les composants) d'autres classes auxquelles il délègue ses méthodes,
- inheritance : utilisation d'extension pour factoriser les méthodes communes dans la superclasse
- Préférer l'utilisation de délégation à base d'interfaces (potentiellement avec des méthodes par défaut),
- Comme le premier exemple des listes réductibles.

Quand utiliser les extensions ?

Question difficile ! Quelques éléments de réponse, que nous détaillerons plus tard.

- condition nécessaire (mais pas suffisante) : l'extension et la classe parent doivent être dans une relation de type "est un/une" (exemple : un cercle est une forme).
- *principe de substitution de Liskov* : tout comportement attendu pour les instances de la classe parent doit aussi être vrai pour les instances de l'extension.
- certains patrons de conception utilisent les extensions. Un *patron de conception* est une description d'un découpage en classes et interfaces pour résoudre certains problèmes courants en programmation.

Partie 7

Généricité

Méthodes génériques

Exemple 1 (I)

Écrire toutes les valeurs d'une collection d'entiers :

```
public String write(int[] ints) {
  boolean isFirst = true;
  StringBuilder builder = new StringBuilder();
  for (int i : ints) {
    if (isFirst) { isFirst = false; }
    else { builder.append(", "); }
    builder.append(i);
  }
  return builder.toString();
}
```

Exemple 1 (II)

Écrire toutes les valeurs d'une collection de chaînes :

```
public String write(String[] strings) {
  boolean isFirst = true;
  StringBuilder builder = new StringBuilder();
  for (String str : strings) {
    if (isFirst) { isFirst = false; }
    else { builder.append(", "); }
    builder.append(str);
  }
  return builder.toString();
}
```

Analyse de l'exemple 1

- Les deux fragments sont quasiment identiques,
- la seule différence concerne le type des éléments manipulés,
- la nature précise des éléments manipulés n'est pas utile : ce fragment fonctionnerait pour n'importe quel type,
- si on a 20 types différents, besoin d'écrire 20 fois la méthode ?

Besoin d'un mécanisme pour l'écrire une seule fois !

Solution ?

Utiliser une interface ?

- le type du paramètre devient `Iterable<String>`,
- il faut adapter la boucle `for`,
- il faut adapter tous les appels à la méthode `write` (car `int[]` n'implément pas `Iterable<String>`).

Spécificité de cette exemple :

- seuls les types changent,
- les instructions sont exactement les mêmes.

Interface : généralise plusieurs méthodes (plusieurs blocs d'instruction différents) sous un même type.

Ce n'est pas ce que nous voulons ici !

Bonne solution

Abstraire le type :

```
public <T> String write(T[] things) {
  boolean isFirst = true;
  StringBuilder builder = new StringBuilder();
  for (T thing : things) {
    if (isFirst) { isFirst = false; }
    else { builder.append(", "); }
    builder.append(thing);
  }
  return builder.toString();
}
```

T est une abstraction de type : même rôle qu'une variable en mathématique, représente une valeur (ici un type) non-précisée.

Utilisation de la méthode

```
int[] ints = { 1, 2, 5, 10, 100 };
System.out.println(write(ints));
  // output: 1, 2, 5, 10, 100
string[] strings { "Hello", "", "World", "!" };
System.out.println(write(strings));
  // output: Hello, , World, !
```

Utilisation transparente.

Syntaxe des méthodes génériques

```
public <T> String write(T[] things) {
   ...
}
```

- méthode avec des variables de types : *méthode générique*,
- `T` : *paramètre de généricité*, variable de type,
- déclaration de la variable de type, juste avant le type de retour de la méthode.
- plusieurs variables de type : `<S,T,U>`,
- le reste garde la syntaxe usuelle.

Variables de types

- <T> déclare une *variable de type*,
- les variables usuelles abstraient des valeurs, on leur substitue la valeur qui leur est donnée *à l'usage*,
- les variables de types abstraient des types, on leur substitue le type adéquat *à l'usage*.
- Différents usages : différentes valeurs, différents types.

Interfaces génériques

Exemple 2 (I)

```
public int sum(int[] ints) {
  int sum = 0;
  for (int i : ints) {
    sum = sum + i;
  }
  return sum;
}
```

Exemple 2 (II)

```
public double product(double[] doubles) {
  double product = 1;
  for (double d : doubles) {
    product = product * d;
  }
  return product;
}
```

Analyse de l'exemple 2

```
public <T> T myMethod(T[] values) {
  T result = ___;
  for (T value : values) {
    result = result ___ value;
  }
  return result;
}
```

- fragments très similaires,
- mais les valeurs ne sont pas utilisées de la même façon : pas le même comportement,
- points communs : utilisation d'une valeur initiale, utilisation d'un opérateur binaire.
- services identiques, comportements différents : interface !

Interface générique

```
public interface Monoid<T> {
  T neutral();
  T operator(T left, T right);
}

public class IntWithAddition
  implements Monoid<Integer> {

  Integer neutral() { return 0; }
  Integer operator(Integer left, Integer right) {
   return left + right;
  }
}
```

Analyse de l'interface générique

Deux abstractions simultanément :

- abstraction des comportements (blocs d'instruction) par l'interface,
- abstraction des types par la généricité

Les deux abstractions sont indépendantes, on peut par exemple :

- définir `List<E>` implémentant `Monoid<E>`, avec la liste vide comme neutre, la concaténation comme opérateur (le type reste abstrait),
- définir une méthode prenant en paramètre un `Monoid<Integer>` (le comportement reste abstrait),
- définir une interface `Group<E>`, qui étend `Monoid<E>` avec une opération d'inverse (comportements et types restent abstraits).

Utilisation de l'interface générique

```
public <T> T reduce(T[] values, Monoid<T> monoid) {
  T result = monoid.neutral();
  for (T value : values) {
    result = monoid.operator(result, value);
  }
  return result;
}


int[] ints = { 1, 2, 5, 10, 100};
int sum = reduce(ints, new IntWithAddition());
```

Utilisation de l'interface générique

Les collections sont des interfaces génériques aussi :

```
public <T> T reduce(Collection<T> values,
                    Monoid<T> monoid) {
  T result = monoid.neutral();
  for (T value : values) {
    result = monoid.operator(result, value);
  }
  return result;
}

Collection<Integer> ints =
  Arrays.asList(1, 2, 5, 10, 100);
int sum = reduce(ints, new IntWithAddition());
```

Types génériques, types paramétrés

Type générique

Un *type générique* est un type admettant des paramètres formels de types.

Exemples : `ArrayList<E>`, `Group<T>`, `Iterable<T>`

Type paramétré

Un *type paramétré* est l'*instanciation* d'un type générique avec des paramètres de types concrets.

Exemples : `ArrayList<Integer>`, `Group<String>`, `Iterable<Double>`.

L'interface Comparable

```
public interface Comparable<T> {
  int compareTo(T value);
}
```

- compareTo retourne une entier :
  - strictement positif si this > value,
  - strictement négatif si this < value,
  - nul si this = value,
- doit avoir un comportement cohérent avec equals,
- doit avoir un comportement cohérent avec equals,
- utile pour TreeMap, TreeSet, Collections.sort.

L'interface Comparator<T>

```
public interface Comparator<T> {
  int compare(T value1, T value2);
}
```

- compareTo retourne une entier :
  - strictement positif si value1 > value2,
  - strictement négatif si value1 < value2,
  - nul si value1 = value2,
- doit avoir un comportement cohérent avec equals,
- utile pour avoir des ordres autres que celui défini via Comparable.

Classes génériques

Implémentation d'une pile d'entiers (I)

```
public class IntStack implements Iterable<Integer> {
  Integer[] values = new Integer[1000];
  Integer firstEmptyCell = 0;
  boolean isEmpty() { return firstEmptyCell == 0; }
  public void push(Integer value) {
    values[firstEmptyCell++] = value;
  }
  public Integer peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return values[firstEmptyCell-1];
  }
  public Integer poll() {
    if(isEmpty()) throw new NoSuchElementException();
    return values[--firstEmptyCell];
  }
  ...
}
```

Implémentation d'une pile d'entiers (II)

```
public class IntStack implements Iterable<Integer> {
  ...
  public Iterator<Integer> iterator() {
    return new StackIterator(values, firstEmptyCell-1);
  }
  private class StackIterator
      implements Iterator<Integer> {
    private final Integer[] values;
    private Integer current = 0;
    private final Integer last;
    public StackIterator(...) { ... }
    public boolean hasNext() { return current <= last; }
    public Integer next() {
      if (!hasNext())
        throw new NoSuchElementException();
      return values[next++];
    }
  }
}
```

Implémentation d'une pile de `String` ?

- Même chose en substituant `String` à `Integer`,
- DRY : besoin d'un mécanisme pour ne pas dupliquer le programme,
- Généricité de classe : paramètrage de la classe par un type générique.

Notre classe `Stack` n'utilisait pas les méthodes des valeurs mises dans la pile : la restriction au type `Integer` n'est pas nécessaire.

Implémentation d'une pile générique (I)

```
public class Stack<T> implements Iterable<T> {
  T[] values = new T[1000];
  T firstEmptyCell = 0;
  boolean isEmpty() { return firstEmptyCell == 0; }
  public void push(T value) {
    values[firstEmptyCell++] = value;
  }
  public T peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return values[firstEmptyCell-1];
  }
  public T poll() {
    if(isEmpty()) throw new NoSuchElementException();
    return values[--firstEmptyCell];
  }
  ...
}
```

Implémentation d'une pile générique (II)

```
public class Stack<T> implements Iterable<T> {
  ...
  public Iterator<T> iterator() {
    return new StackIterator(values, firstEmptyCell-1);
  }
  private class StackIterator implements Iterator<T> {
    private final T[] values;
    private T current = 0;
    private final T last;
    public StackIterator(...) { ... }
    public boolean hasNext() { return current <= last; }
    public T next() {
      if (!hasNext())
        throw new NoSuchElementException();
      return values[next++];
    }
  }
}
```

Classe générique

Déclaration d'une classe générique :

```
public class MyGenericClass<T> { ... }
```

Instanciation d'une classe générique :

```
MyGenericClass<Integer> obj =
  new MyGenericClass<Integer>(...);
// or when there is no ambiguity:
MyGenericClass<Integer> obj =
  new MyGenericClass<>(...);
```

Tout le reste identique (méthodes, déclaration du constructeur,...).

Classe Greatest

```
public class Greatest<E> {
  private E max;
  public Greatest(E initialMax) {
    this.max = initialMax;
  }
  public void add(E elt) {
    if ( elt.compareTo(max)  > 0) { max = elt; }
  }
  public E getMax() {
    return max;
  }
}
```

Il faut que E implémente Comparable<E>.

Classe Greatest

```
public class Greatest<E extends Comparable<E>> {
  private E max;
  public Greatest(E initialMax) { ... }
  public void add(E elt) {
    if ( elt.compareTo(max)  > 0) { max = elt; }
  }
  public E getMax() { ... }
}
```

- <E ... > car E est le paramètre de type,
- <E extends Comparable<E>> car E doit implémenté la méthode de Comparable<E>.
- c'est Comparable<E> : Comparable est générique aussi.
- extends (dans ce contexte) permet donc d'imposer une *contrainte* d'interface sur le type générique.

Récapitulatif génériques

- méthodes génériques,
- interfaces génériques,
- classes génériques,
- contraintes d'interface.

```
public <T> Result myMethod(...) { ... }
public interface myInterface<T> { ... }
public class myClass<T> { ... }

... <T extends MyInterface> ...
```

Partie 8

Classes anonymes et lambdas

Classes anonymes

Un exemple

Tâche :

Filtrer une liste `cars` de `Car` selon divers critères.

```
public class Car {
  private final String brand;
  private final String productionYear;
  private final int price;
  ...

  public String getBrand() { ... }
  public int getPrice() { ... }
  ...
}
```

Quelques filtres

Solution directe :

```
public List<Car> selectByMaxPrice(int maxPrice) {
  List<Car> selection = new ArrayList<>();
  for (Car car : cars) {
    if (car.getPrice() <= maxPrice) {
      selection.add(car);
    }
  }
  return selection;
}

public List<Car> selectByBrand(String brandName) {
  List<Car> selection = new ArrayList<>();
  for (Car car : cars) {
    if (car.getBrand().equals(brandName)) {
      selection.add(car);
    }
  }
  return selection;
}
```

Analyse

- Répétition : violation du principe DRY !
- Limité : une méthode par critère.
- Impossible de composer (par exemple, pour lister les `"Vaillante"` à moins de 15000 euros).

Première solution

Utilisation d'une interface :

```
  public interface Predicate<T> { // Java 8+
    boolean test(T value);
    ...
  }

  public List<Car> select(Predicate<Car> property) {
    List<Car> selection = new ArrayList<>();
    for (Car car : cars) {
      if (property.test(car)) selection.add(car);
    }
    return selection;
  }
}
```

Réalisation des critères

```
public class HasBrand implements Predicate<Car> {
  private final string expectedBrand;
  public HasBrand(String brand) { expectedBrand = brand }
  public boolean test(Car car) {
    return car.brand.equals(expectedBrand);
  }
}

public class HasMaxPrice implements Predicate<Car> {
  private final int maxPrice;
  public HasMaxPrice(int maxPrice) { this.maxPrice = maxPrice; }
  public boolean test(Car car) {
    return car.getPrice() <= maxPrice;
  }
}

// Exemples d'utilisation
List<Car> vaillantes = carDealer.select(new HasBrand("Vaillante"));
List<Car> cheapCars = carDealer.select(new HasMaxPrice(15000));
```

Analyse de cette solution

- interface `Predicate`, représente un critère de test,
- chaque critère possible requiert une classe implémentant `Predicate`,
- méthode `select` prenant un `Predicate`,
- `select` prend donc une fonction en paramètre !

Limites :

- Verbeux,
- nécessite d'écrire une classe complète pour chaque test.
- Toujours pas composable.

Classes anonymes

On peut éviter de devoir définir une classe pour chaque critère de sélection, avec les classes anonymes.

Création d'un objet de l'interface `Comparator` :

```
Comparator<Double,Double> doubleComparator =
  new Comparator() {
    public int compare(Double x, Double y) {
      double diff = Math.abs(x - y);
      return (diff < EPSILON) ? 0:
             (x > y) ? 1:
                       -1;
    }
  };
```

Classes anonymes (interface)

Syntaxe :

```
new InterfaceName() {
  public type1 method1(Arg1 arg1) { ... }
  public type2 method2(Arg2 arg2) { ... }
  ...
}
```

Peut contenir :

- des méthodes (privées, publiques, ...),
- des propriétés (final, privées, ...),
- pas de constructeur.

Classes anonymes (extension)

On peut aussi créer une extension anonyme d'une classe.
Syntaxe :

```
new SuperClassName(constructorArgs) {
  ... // méthodes et propriétés (dont redéfinition)
}
```

Exemple :

```
Point2D origin =
  new Point2D(0,0) {
    @Override public String toString() {
      return "origin";
    }
  };
```

Classes anonymes

- l'instanciation d'une classe anonyme est une instruction !
- Définition et instanciation simultanée d'une classe,
- La classe est instanciée une seule fois,
- La classe ne porte pas de nom, donc elle n'est pas réutilisable ailleurs.
- L'objet créé implémente une interface ou étend une autre classe.
- Dans le cas d'une extension, les paramètres du constructeur de la classe parent sont donnés entre les parenthèses de la déclaration de la classe anonyme.

Limitation des classes anonymes

Une classe anonyme

- peut déclarer des méthodes et des propriétés,
- ne peut pas déclarer d'interfaces,
- ne peut rien posséder de statique (sauf des constantes),
- n'a pas de constructeur (mais peu avoir un initialiseur),
- peut utiliser des propriétés de l'objet, et des variables de la méthode qui la définit seulement si elles sont déclarées `final` !

Initialiseur

Comment initialisé une instance d'une classe anonyme ?

Solution 1 : Précalculer les valeurs des propriétés.

```
List<Integer> multiplicationTable = new ArrayList<>(n);
for (int i = 0; i < n; i++) {
  multiplicationTable.add(i * k);
}
return new IntUnaryOperator() {
  List<Integer> cached = multiplicationTable;
  public int applyAsInt(int i) {
    return cached.get(i);
  }
};
```

Initialiseur

Comment initialisé une instance d'une classe anonyme ?

Solution 2 : Utiliser un initialisateur : un bloc d'instructions non-nommé, qui est normalement exécuté juste avant les instructions du constructeur.

```
return new IntUnaryOperator() {
  List<Integer> cached = new ArrayList<>(n);
  { // this is an initializer
    for (int i = 0; i < n; i++) {
      multiplicationTable.add(i * k);
    }
  }
  public int applyAsInt(int i) {
    return cached.get(i);
  }
};
```

Application à la sélection

```
// Exemples d'utilisation
List<Car> vaillantes =
  carDealer.select(new Predicate<Car>() {
      public boolean test(Car car) {
        return "Vaillante".equals(car.getBrand());
    }
  });

List<Car> cheapCars =
  carDealer.select(new Predicate<Car>() {
      public boolean test(Car car) {
        return car.getPrice() <= 15000;
    }
  });
```

Analyse de cette solution

Avec des classes anonymes :

- pas besoin d'une classe supplémentaire (mais toujours remplaçable par une classe nommée),
- moins verbeux (disparition du constructeur, des propriétés),
- mais casse la propreté de la hiérarchie des programmes (classe, méthodes, instructions), puisqu'une méthode peut contenir un instruction définissant une classe.

Limites :

- Toujours pas composable.

Lambdas

Le problème

Comment régler le problème de la verbosité des classes anonymes ?

- Création d'un objet,
- implémentant une interface,
- souvent avec une seule méthode.

Verbeux : 4 lignes pour une instruction !

- une ligne pour la déclaration de classe,
- une ligne pour la déclaration de méthode,
- puis les instructions de la méthode (souvent une seule),
- puis fermer les deux paires d'accolades

Lambda et interface fonctionnelle

Interface fonctionnelle

Une interface est dite fonctionnelle si elle contient une seule méthode sans implémentation par défaut (et un nombre arbitraire de méthodes par défaut).

Lambda

Une lambda est une expression, utilisant une syntaxe spéciale, pour définir une classe anonyme implémentant une interface fonctionnelle.

Exemple d'interface fonctionnelle

Une interface fonctionnelle :

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

Utilisation avec des lambdas :

```
List<Car> vaillantes =
  carsDealers.select(
    car -> "Vaillante".equals(car.getBrand())
  );
List<Car> cheapCars =
  carsDealers.select(car -> car.getPrice() <= 15000);
```

Sucre syntaxique

Les lambdas sont une utilité syntaxique :

- ne permettent rien de plus que certaines formes de classes anonymes,
- peuvent toujours être remplacées par une classe anonyme équivalente.

On parle de sucre syntaxique : le compilateur traduit automatiquement la fonctionnalité vers la syntaxe verbeuse.

Les enregistrements (`record`) sont un autre exemple de sucre syntaxique.

Exemple de traduction

```
car -> car.getPrice();
```

est traduit en :

```
new ToDoubleFunction<Car>() {
  public double applyAsDouble(Car car) {
    return car.getPrice();
  }
}
```

Exemple de traduction

```
bound -> {
  int product = 1;
  for (int n = 0; n <= bound; n++) {
    product = product * bound;
  }
  return product;
}
```

est traduit en :

```
new IntUnaryOperator() {
  public int apply(int bound) {
    product = 1;
    for (int n = 0; n <= bound; n++) {
      product = product * bound;
    }
    return product;
  }
}
```

Syntaxe des lambdas

Deux syntaxes :

```
(arg1,...,argN) -> expression(arg1,...,argN)
(arg1,...,argN) -> { bloc d'instructions avec return }
```

- Si un seul paramètre, les parenthèses peuvent être omises.
- Si pas de paramètre, on le note `()`.
- Il n'est en général pas nécessaire de donner le type des paramètres.
- Si on indique le type d'un paramètre, on doit indiquer le type de tous les paramètres (c'est tout ou rien).

Inférence de type

Comment Java devine les types des paramètres ?

Utilisation d'un algorithme, l'inférence de type, qui se base sur :

- comment les paramètres sont utilisés dans la lambda,
- dans quelle contexte la lambda est utilisée (comme paramètre d'une méthode, pour être stockées dans une variable), ce contexte a généralement un type précis.
- Une lambda implémente toujours une interface fonctionnelle ! Il faut l'importer.

Limitation des lambdas

- Si l'inférence de type ne suffit pas à établir le type, ajoutez les types des paramètres.
- Comme pour les classes anonymes, la lambda ne peut utiliser une variable de méthode définie à l'extérieur seulement si elle est `final` (sinon erreur à le compilation).

Exemple :

```
List<IntUnaryOperator> multipliers = new ArrayList<>();
for (int index = 1; index < 10; index++) {
  multipliers.add(n -> n * index);
  // erreur : index n'est pas final.
}
```

Références de méthode (I)

Souvent, l'expression lambda a une des formes :

```
obj -> obj.method();
value -> obj.method(value);
value -> MyClass.staticMethod(value);
```

Exemples :

```
cars.forEach(car -> System.out.println(car));
ToIntFunction<Car> toProductionYear =
  car -> car.getProductionYear();
IntProducer randomInts = () -> gen.nextInt();
```

Références de méthode (II)

On peut alors utiliser une référence de méthode pour simplifier la notation :

```
MyClass::method // obj -> obj.method()
obj::method     // value -> obj.method(value)
MyClass::method // value -> MyClass.method(value)
MyClass::new    // value -> new MyClass(value)
```

Exemples :

```
cars.forEach(System.out::println);
IntProducer randomInts = gen::nextInt;
Comparator<Car> compareCarByProductionYear =
  Comparator.comparing(Car::getProductionYear);
```

Analyse de la solution avec lambdas

- Peu verbeux.
- Autorise la création de valeurs fonctionnelles, pour passer des fonctions en paramètres, ou retourner des fonctions.
- Cache le fait que les fonctions sont représentées en Java par des objets à une seule méthode.

Limites :

- Toujours pas composable.

La solution au prochain cours...

Partie 9

Interfaces fonctionnelles et Streams

Interfaces fonctionnelles

Définition

Interface fonctionnelle

Une interface est dite fonctionnelle si elle contient une seule méthode sans implémentation par défaut (et un nombre arbitraire de méthodes par défaut).

- Un objet implémentant une interface fonctionnelle représente une fonction.
- On annote une interface fonctionnelle avec `@FunctionalInterface`,
- `java.util.function` contient de nombreuses interfaces fonctionnelles.

L'exemple des prédicats

L'interface fonctionnelle `Predicate<T>` représente la capacité à tester une propriété des objets de types `T`.

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);

  default Predicate<T> and(Predicate<T> other) { ... }
  default Predicate<T> or(Predicate<T> other) { ... }
  default Predicate<T> negate() { ... }

  static <T> Predicate<T> not(Predicate<T> target) { ... }
  static <T> Predicate<T> isEqual(Object tergetRef) { ... }
}
```

Détail de l'interface `Predicate`

- méthode principale `test(T t)` : test d'une propriété,
- méthodes par défaut `and`, `or`, `negate` : ces méthodes permettent de composer des prédicats entre eux, pour construire des prédicats plus complexes.
- méthodes statiques `not`, `isEqual` : fabriquants de prédicats particuliers. Une interface peut ainsi contenir quelques fabriquants, on aurait pu sinon les mettre dans une classe `Predicates` (comme pour `Arrays`, `Collections`, ...).

Composabilité des prédicats

Grâce aux méthodes par défaut, on obtient enfin la composabilité.

```
Predicate<Car> isCheap =
  (car -> car.getPrice() <= 15000);
Predicate<Car> isVaillante =
  (car -> "Vaillante".isEquals(car.getBrand()));
Predicate<Car> isCheapVaillante =
  isCheap.and(isVaillante);

carDealers.select(isCheapVaillante);
```

L'interface consommateur

Consommateur :

```
@FunctionalInterface
public interface Consumer(T) {
  void accept(T value);
}
```

Exemple :

```
public void printAllCars() {
  cars.forEach(car -> System.out.println(car));
}
```

Consommateur et `ForEach`

La méthode `forEach` est une méthode par défaut d'`Iterable`.

```
public interface Iterable<T> {
  Iterator<T> iterator();
  default forEach(Consumer<T> action) { ... }
  ...
}
```

Elle permet d'écrire des boucles `for` avec une syntaxe fonctionnelle :

```
for (Car car : cars) {
  System.out.println(car);
}
// Équivalent à
cars.forEach(car -> System.out.println(car));
```

L'interface producteur

Producteur :

```
public interface Supplier<T> {
  T get();
}
```

Exemple :

```
public void printTenInts(Supplier<? extends Integer> source) {
  for (int i = 0; i < 10; i++) {
    System.out.println(source.get());
  }
}

Random gen = new Random();
Scanner scanner = new Scanner("10 4 17 2 8 12 0 3 1 19 9 7");
printTenInts(() -> gen.nextInt());
printTenInts(() -> scanner.nextInt());
```

L'interface fonction

Fonction :

```
public interface Function<T,R> {
  R apply(T value);
  ...
}
```

Exemple :

```
Matrix2D rot = new Matrix2D(0.5, -Math.sqrt(2)/2, Math.sqrt(2)/2, 0.5);
Function<Point2D,Point2D> rotation = point -> rot.transform(point);

Point2D vec = new Point2D(3,-1);
Function<Point2D,Point2D> translation = point -> point.add(vec);

Function<Point2D,Point2D> map = rotation.andThen(translation);
Point2D result = map.apply(new Point2D(10,-2));
```

Méthodes par défaut

```
public interface Function<T,R> {
  R apply(T value);
  default <V> andThen(Function<R,V> after) { ... }
  default <V> compose(Function<V,T> before) { ... }
  static <T> Function<T,T> identity() { ... }
}
```

- `f.compose(g)` : composition de deux fonctions $(f \circ g)$,
- `f.andThen(g)` : composition inversée $(g \circ f)$,
- `identity()` : fabriquant de la fonction identité.

Les interfaces opérateurs

Opérateur unaire :

```
public interface UnaryOperator<T> extends Function<T,T> {}
```

Opérateur binaire :

```
public interface BinaryOperator<T> extends BiFunction<T,T,T> {};

public interface BiFunction<T,U,R> {
  R apply(T t, U u);
  default <V> V andThen(Function<R,V> after) {...}
}
```

L'interface comparateur

Comparateur :

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  ...
}
```

Exemple :

```
ToIntFunction<Car> toProductionYear =
  car -> car.getProductionYear();
Comparator<Car> compareCarByProductionYear =
  Comparator.comparing(toProductionYear);
int result =
  compareCarByProductionYear.compare(car1,car2);
```

Composition des comparateurs

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  default Comparator<T> reversed() { ... }
  default int thenComparing(Comparator<T> other) { ... }
  static <T,U extends Comparable<U>> Comparator<T>
    comparing(Function<T,U> keyExtractor) { ... }
  ...
}
```

- `reversed()` : un comparateur dans l'ordre inverse.
- `cmp1.thenComparing(cmp2)` : un comparateur qui compare selon `cmp1`, et utilise `cmp2` pour départager les égalités.
- `comparing(toKey)` : compare des instances de `T`, en utilisant `toKey` pour obtenir des valeurs de type `U`, ce sont les `U` qui sont comparés.

amU Faculté des sciences Aix Marseille Université

Interfaces pour les types de bases

Les interfaces se déclinent pour les types `int`, `double`, ...

- `DoubleUnaryOperator`
- `LongConsumer`
- `IntFunction`
- `ToIntFunction`
- `DoublePredicate`
- `...`

Subtilité des génériques

Voici les authentiques déclarations de `Comparator` :

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  default Comparator<T> reversed() { ... }
  default int thenComparing(Comparator<? super T> other) { ... }
  static <T,U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T,? extends U> keyExtractor) { ... }
  ...
}
```

Que signifie les `<? super T>` ou `<? extends U>` ?

Subtilité des génériques

- `Comparator<? super T> other` signifie que `other` est un comparateur d'une super-classe de `T` (ou de `T` elle-même). Un tel comparateur peut effectivement comparer des instances de `T`.
- `Function<? super T, ? extends U> keyExtractor` signifie que `keyExtractor` est une fonction d'une super-classe de `T` (ou de `T`), vers une extension de `U` (ou `U`) elle-même.

Ces deux notations permettent un peu plus de flexibilité, au prix de notations difficiles à lire. Conseil : ignorer ces notations, interpréter `<? super T>` ou `<? extends T>` comme simplement `T`.

Stream

Exemple

```
public class Car {
   ...
   public String getBrand();
   public int getProductionYear();
   public double getTopSpeed();
   ...
}
```

Tâche :
Étant donnée une liste de voitures, calculer la moyenne d'âge des voitures de marque "Vaillante" ayant une vitesse maximum d'au moins $200\mathrm{km.h}^{-1}$.

Solution 1

Solution impérative :

```
public double averageFastVaillanteAge(List<Car> cars) {
  int count = 0;
  int sumAge = 0;
  for (Car car : cars) {
    if (!"Vaillante".equals(car.getBrand())) continue;
    if (car.getTopSpeed() < 200) continue;
    sumAge = sumAge + (currentYear - car.getProductionYear());
    count++;
  }
  if (count == 0) { throw new NoSuchElementException(); }
  return (double) sumAge / count;
}
```

Solution 2

Solution fonctionnelle :

```
public double averageFastVaillanteAge(List<Car> cars) {
  return cars.stream()
  .filter(car -> "Vaillante".equals(car.getBrand()))
  .filter(car -> car.getTopSpeed() >= 200)
  .mapToDouble(car -> currentYear - car.getProductionYear())
  .average()
  .orElseThrow(NoSuchElementException::new);

}
```

Analyse de la deuxième solution.

Avantages :

- pas de structure de contrôle,
- description nommées des opérations effectuées (`filter`, `map`, `average`) : le programme décrit l'intention, et non pas la méthode de calcul,
- parallélisable automatiquement (`parallelStream()`)

Désavantages :

- techniquement plus compliqué (streams, lambdas) ? (question d'habitude)

Les streams

- Un objet de l'interface `Stream<T>` est un flux de données de type `T`.
- Ces données sont en séquence (la première, la deuxième, ...).
- Elles peuvent être générées au besoin (paresseux) : les données ne sont générées qu'au moment d'être utilisées. Le `Stream` ne stocke pas de donnée.
- Le flux peut être infini.
- Les calculs sur ces données sont parallélisables.

Cycle de vie d'un `Stream`

Un `Stream` est toujours utilisé en trois phases :

- Création du `Stream` (à partir d'une collection, d'un fichier,...),
- Multiples transformations (*intermediate operations*) de la `Stream` (suppression d'éléments, transformation de chaque élément, combinaison) en une autre `Stream`,
- Réduction (*terminal operation*) de la `Stream` en une seule valeur (non `Stream`) (calcul de la somme, de la moyenne,...). Une fois l'opération de réduction effectuée, le `Stream` n'est plus utilisable : on ne réduit qu'une seule fois.

Exécution du `Stream`

Un `Stream` peut être déclaré en plusieurs étapes (création dans une méthode, ajouts de transformations dans d'autres méthodes, réduction dans encore une autre méthode). Mais ...

Aucun calcul n'est effectué tant que l'opération de réduction n'est pas appliquée.

Transformations

Une transformation est une méthode de `Stream` retournant un `Stream`.

- `dropWhile`, `takeWhile`, `limit`, `skip` permettent de garder ou de retirer un préfixe du `Stream`,
- `filter` permet de ne garder que les éléments passant un test,
- `map`, `peek`, `flatMap` permettent d'appliquer une opération sur chaque élément,
- `distinct` permet de retirer les doublons,
- `sorted` permet de trier le `Stream`.

Garder/Supprimer un préfixe

- `limit`, `skip` : supprime/garde un préfixe de longueur donné.
- `dropWhile`, `takeWhile` : supprime/garde un préfixe d'éléments vérifiant la condition spécifiée.

```
Stream<T> limit(int n);
Stream<T> skip(int n);
Stream<T> dropWhile(Predicate<? super T> predicate);
Stream<T> takeWhile(Predicate<? super T> predicate);

Stream.of(11,12,13,14,15,16,17,18).limit(3); // 11, 12, 13
Stream.of(11,12,13,14,15,16,17,18).skip(3); // 14, 15, 16, 17, 18
Stream.of(1,2,3,4,5,1,2,8).dropWhile(n -> n < 5); // 5, 1, 2, 8
Stream.of(1,2,3,4,5,1,2,8).takeWhile(n -> n < 5); // 1, 2, 3, 4
```

Filtrer

- `filter` : garde uniquement tous les éléments qui satisfont un prédicat donné.

```
Stream<T> filter(Predicate<? super T> predicate);

Stream.of(1,2,3,4,5,6,7,8).filter(n -> n % 2 == 0); // 2,4,6,8
Stream.of("foo", "bar", "foobar", "baz")
  .filter(str -> str.length() <= 3); // "foo", "bar", "baz"
Stream.of(0.0, 3.14, -1.23, 0.5, -9.99)
  .filter(x -> x > 0.0); // 3.14, 0.5
```

distinct/sorted

- `distinct` supprime les doublons du `Stream` (utilise la méthode `equals` des objets),
- `sorted` trie le `Stream`.

Attention : ces deux opérations ne respectent pas le principe que le `Stream` ne stocke pas de donnée.

```
Stream<T> distinct();
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);

Stream.of(1,6,3,2,4,4,5,2,4).distinct(); // 1,6,3,2,4,5
Stream.of(3,4,2,5,1,6,8).sorted(); // 1,2,3,4,5,6,8
Stream.of(3,4,2,5,1,6,8).sorted((i,j) -> j - i); // 8,6,5,4,3,2,1
```

Manipuler chaque élément

- peek exécute une action pour chaque élément, ne modifie pas le Stream,
- map remplace chaque élément par un nouvel élément (qui peut être d'un autre type),
- flatMap remplace chaque élément par plusieurs nouveaux éléments (qui peuvent être d'un autre type).

```
Stream<T> peek(Consumer<? super T> action);
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
<R> Stream<R> flatMap(Function<T,Stream<R>> mapper); // simplifié !

// affiche "foo\nbar\nfoobar", retourne "foo", "bar", "foobar"
Stream.of("foo", "bar", "foobar").peek(System.out::println);
// retourne "FOO", "BAR", "FOOBAR"
Stream.of("foo", "bar", "foobar").map(String::toUpperCase);
// retourne 'f, 'o', 'o', 'b', 'a', ...
Stream.of("foo", "bar", "foobar").flatMap(String::chars);
```

Réductions

Les réductions consomment le `Stream` pour fournir une seule valeur.

- `allMatch`, `anyMatch`, `noneMatch` pour tester les éléments du `Stream`,
- `count` pour compter le nombre d'éléments,
- `findAny`, `findFirst`, `min`, `max` pour obtenir un seul élément,
- `forEach` pour exécuter une action sur chaque élément,
- `collect`, `reduce` pour des actions plus générales, combinant tous les éléments du `Stream`.

Tester les éléments

- `allMatch` teste qu'un prédicat est vrai pour tous les éléments.
- `anyMatch` teste qu'un prédicat est vrai pour au moins un élément.
- `noneMatch` teste qu'un prédicat n'est vrai pour aucun élément.

```
boolean allMatch(Predicate<? super T> predicate);
boolean anyMatch(Predicate<? super T> predicate);
boolean noneMatch(Predicate<? super T> predicate);

Stream.of(1,2,3,4,5,6,7,8).allMatch(n -> n % 2 == 0); // false
Stream.of(0,2,4,6,8,10).allMatch(n -> n % 2 == 0); // true
Stream.of(1,2,3,4,5,6,7,8).anyMatch(n -> n % 2 == 0); // true
```

Récupérer un élément

- `findAny` retourne un élément arbitraire, s'il y en a.
- `findFirst` retourne le premier élément, s'il y en a.
- `min` et `max` retourne le minimum et le maximum, s'ils existent.
- les 4 méthodes retournent un `Optional<T>`, un objet contenant zéro ou une instance de `T`.

```
Optional<T> findAny();
Optional<T> findFirst();
Optional<T> min(Comparator<? super T> comparator);
Optional<T> max(Comparator<? super T> comparator);

Stream.of(1,2,3,4).findFirst(); // 1
Stream.of(3,2,4,1).max(Integer::compare); // 4
Stream.of("foo", "bar", "baz").min(String::compare); // "bar"
```

Utiliser chaque élément

- `forEach` permet d'appliquer une action à chaque élément, indépendamment.
- `forEachOrdered` assure en plus que les actions sont effectuées dans l'ordre des éléments du `Stream`.

```
void forEach(Consumer<? super T> action);
void forEachOrdered(Consumer<? super T> action);

// affiche "Hello\nWorld\n" ou "World\nHello\n"
Stream.of("Hello", "World").forEach(System.out::println);

List<Integer> list = new ArrayList<>();
// ajoute les entiers à la liste, dans n'importe quel ordre
Stream.of(1,2,3,4,5).forEach(list::add);
```

Collecter et réduire

- `collect` permet d'appliquer un `Collector` aux éléments du `Stream`,
- `reduce` permet de combiner les éléments du `Stream` avec un opérateur binaire,
- la classe `Collectors` contient de nombreux `Collectors` directement utilisables.

```
T reduce(T identity, BinaryOperator<T> accumulator);
<R,A> R collect(Collector<? super T, A, R> collector);

Stream.of(1,2,3,4,5).reduce(0,Integer::sum); // 15
Stream.of(5,3,4,1,2).reduce(0,Integer::max); // 5
// construit la liste 1,2,5,3,4
Stream.of(1,2,5,3,4).collect(Collectors.toList());
Stream.of("foo", "bar", "baz").summingInt(String::length); // 9
```

Générer un `Stream`

- À partir d'une collection : méthodes `stream()` ou `parallelStream()` (autorise l'évaluation en parallèle des transformations).
- À partir d'un tableau : méthode `Arrays.stream`.
- Par des fabriquants dans certaines classes : `IntStream.range`, `Random.ints`, ...
- En utilisant un objet producteur de valeurs, comme `Iterator<T>`.
- Certaines méthodes retournent des `Stream`.
- En implémentant l'interface `Spliterator` (difficile !)

Exemple de génération

```java
Stream.of(1,2,3,4,5); // explicite
IntStream.range(1,5); // 1, 2, 3, 4
IntStream.iterate(0, n -> n < 10, n -> n + 2); // 0, 2, 4, 6, 8
List<Integer> list = List.of(1,2,3,4);
list.stream(); // 1, 2, 3, 4
Integer[] ints = { 1, 2, 3, 4 };
Arrays.stream(ints); // 1, 2, 3, 4
Random.ints(0,10).limit(5); // 4, 7, 2, 4, 0
```

Chaînage

Les opérations de transformation consomment le `Stream` et retourne un nouveau `Stream`. On peut donc chaîner la génération, les transformations, et la réduction en une seule expression.

```
public int sumOfOddSquares(int n) {
  return IntStream.rangeClosed(1,n) // génération
    .filter(k -> k % 2 != 0) // transformation 1
    .map(k -> k * k) // transformation 2
    .sum(); // réduction
}
```

Chaînage de méthode : `obj.m1().m2().m3()...`

IntStream, DoubleStream, LongStream

Pour des raisons d'efficacité, il existe des versions non-génériques de `Stream` pour les types primitifs `int`, `long` et `double`.

- Usage similaire des méthodes `filter`, `map`, ...
- Fabriquants adaptés dans chaque classe, comme `IntStream.range`.
- Réductions supplémentaires (`average`, `summaryStatistics`).

Conversion entre Stream

Pour passer de Stream<T> vers IntStream,... :

```
// dans Stream<T>
IntStream mapToInt(ToIntFunction<? super T> mapper);
IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper);
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);
...
```

Pour passer de IntStream vers Stream<T>, ... :

```
// dans IntStream
Stream<Integer> boxed();
<U> Stream<U> mapToObj(IntFunction<? extends U> mapper);
DoubleStream mapToDouble(IntToDoubleFunction mapper);
LongStream mapToLong(IntToLongFunction mapper);
// pas de version adaptée de flatMap
```

Récapitulatif

Traits fonctionnels de Java :

- objets persistents avec propriétés finales,
- interfaces fonctionnelles pour manipuler des valeurs fonctionnelles,
- lambda pour définir des valeurs fonctionnelles facilement,

```
(rho,theta) -> new Point2D(rho * cos(theta), rho * sin(theta))
```

- `Stream` et ses méthodes, prenant des fonctions en paramètres.

Récapitulatif

Manipulation des `Stream` :

- Création (par exemple depuis une collection),
- Transformations (suppression d'éléments `filter`, transformation de chaque élément `map`, tri `sorted`),
- Réduction à une seule valeur (action sur chaque élément `forEach`, collecte par un objet `Collector` avec `collect`).

Partie 10

Exceptions

Définition et exemple simple

Définition

Un programme peut être confronté à une condition exceptionnelle durant son exécution.

Exception

Une exception est une situation qui empêche l'exécution normale du programme.

- un fichier nécessaire à l'exécution du programme n'existe pas,
- une division par zéro,
- un débordement dans un tableau,
- un besoin de se connecter à un serveur et celui-ci est injoignable,
- un dépilement d'une pile vide,

Empêcher l'exécution normale ?

Que signifie empêcher l'exécution normale ? Exemple : lecture d'un fichier de données.

Situation normale :

- ouverture du fichier,
- lecture du fichier,
- fermeture du fichier.

Situation exceptionnelle :

- ouverture du fichier $\rightarrow$ échec !
- ne peut pas lire le fichier !
- ne peut pas fermer le fichier !

Le traitement normal du fichier n'est pas possible en situation d'exception.

Exception *vs.* erreur

Une exception peut être une erreur. . .ou pas. Ce peut aussi être un comportement anormal et rare, mais inévitable par le programmeur.

Erreurs :

- division par zéro (`ArithmeticException`),
- débordement d'un tableau (`ArrayOutOfBoundException`),
- dépassement de pile (`StackOverflowException`),. . .

Comportement rare :

- fichier inouvrable (`FileNotFoundException`),
- connexion coupée (`ClosedChannelException`),
- opération système interrompue (`InterruptedIOException`),. . .

Gestion des exceptions en Java

Comment une méthode doit gérer une exception ?

- elle ne retourne pas,
- elle crée un objet exception (étendant `Exception`),
- elle termine en émettant cette exception (mot-clé `throw`).
- l'exception est propagée à la méthode appelante, qui elle-même termine immédiatement en émettant l'exception, ...
- toute méthode peut rattraper l'exception, pour reprendre le calcul normalement après un éventuel correctif.
- Une exception non-rattrapée termine le programme.

Terminer avec une exception

Comment émettre une exception ?

- mot-clé `throw` avec une instance d'une exception.

```
throw new NoSuchElementException();
```

- `throw` est une instruction terminant l'exécution de la méthode.
- une méthode terminée avec `throw` ne retourne pas de valeur !
- une méthode qui peut terminer avec une exception doit le mentionner dans sa déclaration, avec le mot-clé `throws`.

Exemple simple

On considère une roulette de casino :

- 36 numéros rouges ou noirs, plus le 0 vert (la banque encaisse tous les paris).
- On tire un numéro aléatoire à chaque partie.
- Les joueurs peuvent parier sur un numéro précis, une couleur, une parité, un groupe de numéros, ...
- On considère le tirage du 0 exceptionnel. Ce n'est pas un bon cas d'usage car ce n'est ni anormal, ni erroné, mais cela va nous permettre d'illustrer les notions sur un cas simple.

Exemple d'émission

```
import java.util.Random;

public class Roulette {

  private static final Random gen = new Random();
  public static class ZeroException extends Exception {}

  public int roll() throws ZeroException {
    int result = gen.nextInt(37);
    if (result == 0) {
      throw new ZeroException();
    }
    return result;
  }
}
```

Exercice

Exercice

Dans le transparent précédent, repérer :

- la définition de la classe d'exception,
- l'émission de l'exception (`throw`),
- l'instanciation d'une exception (avec `new`),
- la déclaration d'exceptions (`throws`).

Méthode subissant une exception

Une méthode `m()` peut exécuter une instruction (par exemple appeler une autre méthode `mayFail()`) qui termine avec une exception. `m()` peut ignorer ou traiter l'exception.

si `m()` ignore l'exception :

- `m()` termine immédiatement,
- `m()` ne retourne pas de valeur,
- `m()` propage l'exception, qui peut être traitée ou ignorée par la méthode ayant fait appel à `m()`,
- `m()` doit déclarer avec `throws` qu'elle peut propager l'exception.

Exemple (suite)

```
  public class Main {
    private static void playThreeTimes(Roulette roulette)
      throws Roulette.ZeroException
    {
      for (int i = 0; i < 3; i++) {
        System.out.println(
          (roulette.roll() % 2 == 0) ? "Pair" : "Impair"
        );
      }
    }
  ...
}
```

`playThreeTimes` ignore l'exception. Si la roulette tombe sur le 0 vert, `playThreeTimes` se termine en propageant l'exception créée par `roulette.roll()`.

Méthode subissant une exception

Une méthode `m()` peut exécuter une instruction (par exemple appeler une autre méthode `mayFail()`) qui termine avec une exception. `m()` peut ignorer ou traiter l'exception.

si `m()` traite l'exception :

- `m()` rattrape l'exception en utilisant la syntaxe `try catch`,
- `m()` définit des instructions spécifiques au traitement de l'exception (bloc d'instructions `catch`),
- `m()` a accès à l'instance d'exception (qui peut avoir des propriétés et des méthodes portant des informations sur l'exception),
- `m()` ne déclare pas propager l'exception.

Rattraper une exception

Pour rattraper une exception :

- les instructions pouvant produire l'exception sont placés dans un bloc précédé du mot-clé `try`,
- après ce bloc, on ajoute un bloc introduit par le mot-clé `catch` et une déclaration d'exception entre parenthèses.
- le bloc `catch` contient les instructions pour traiter le cas exceptionnel.

```
try { // bloc des instructions pouvant produire l'exception
  doSomethingThatMayFail();
}
catch (MyException exc) { // déclaration de l'exception exc
  // bloc des instructions pour traiter l'exception exc
  System.out.println(exc);
}
```

Exemple (suite)

```
public class Main {
  ...
  public static void main(String[] args) {
    Roulette roulette = new Roulette();
    try {
      playThreeTimes(roulette);
    }
    catch (Roulette.ZeroException exc) {
      System.out.println("Zéro, la banque prend tout !");
    }
  }
}
```

Rattrapage de l'exception : lorsque l'exception est levée par `roll`, `roll` puis `playThreeTimes` terminent en propageant l'exception, `main` rattrape l'exception.

Exemples de sortie

Une partie sans exception :

```
Pair
Impair
Impair
```

Une partie interrompue par une exception au deuxième tirage (le troisième tirage n'a pas lieu) :

```
Impair
Zéro, la banque prend tout !
```

La classe `Exception`

- `throw` ne peut émettre que des instances ayant le type `Throwable`.
- `Exception` et `Error` sont deux extensions de `Throwable`.
- `Error` représente les erreurs sérieuses ne devant pas être rattrapées, `Exception` représente les erreurs et comportements exceptionnels pouvant être rattrapés.
- `Exception` est une classe pouvant être étendue.
- N'importe quelle extension d'`Exception` peut être émise, propagée, rattrapée.
- Les extensions peuvent possèder des propriétés et des méthodes, ce sont des classes normales.

La classe `RuntimeException`

- `RuntimeException` est une extension de `Exception`,
- les exceptions de type `RuntimeException` ne nécessitent pas de déclaration `throws`,
- c'est le cas de `RuntimeException` et de ses extensions, comme :
  - `ArithmeticException`,
  - `NoSuchElementException`,
  - `IllegalArgumentException`,
  - `NullPointerException`,...
- souvent, ces exceptions sont conséquences d'erreurs du programmeur !

Usage avancé

finally

- Le mot-clé `finally` permet d'introduire un bloc d'instructions après un `try`.
- Les instructions de `finally` sont toujours exécutées, même en cas de retour (`return`) ou d'émission d'exception (`throw`).
- Les instructions sont exécutées quand le contrôle quitte le bloc `try` ou `catch`.
- Les instructions peuvent ne pas être exécutées si le programme (ou le processus) termine (avec `System.exit` par exemple).
- On peut utiliser `try finally` sans utiliser d'exceptions.
- Cas d'usage principal : fermeture des ressources (fichiers, sockets, ...).

Exemple d'usage de `finally`

Lecture d'entiers dans un fichier :

```
Scanner scanner = new Scanner(Path.of("data.txt"));
List<Integer> ints = new ArrayList<>();
try {
  while (scanner.hasNext()) {
    ints.add(scanner.nextInt());
  }
} catch (InputMismatchException exc) {
  System.out.println("Wrong input at " + ints.size());
} finally {
  scanner.close();
};
```

Exception levée ou pas, le scanner est bien fermé.

Ressource avec `Autoclosable`

La fermeture des ressources est tellement typique qu'il existe une syntaxe adaptée, si la ressource est une classe implémentant `Autoclosable` :

- création de la ressource juste après le `try`,
- la ressource, si elle est ouverte, est automatiquement fermée en fin de bloc.

```
List<Integer> ints = new ArrayList<>();
try (Scanner scanner = new Scanner(Path.of("data.txt"))) {
  while (scanner.hasNext()) {
    ints.add(scanner.nextInt());
  }
} catch InputMismatchException exc) {
  System.out.println("Wrong input at " +  ints.size());
}
```

Autre exemple avec `Autoclosable`

Compter le nombre de lignes d'un fichier :

```
try (BufferedReader reader =
       Files.newBufferedReader(new File("foo.txt"))
    )
{ System.out.println(reader.lines().count()); }
catch (Exception e) {
  System.out.println("Something bad happened.");
}
```

Le fichier est garantie d'être fermé à la fin de la méthode.

Erreur à l'exécution

Une exception non-rattrapée provoque l'arrêt du programme avec un message d'erreur. Exemple :

```
Exception in thread "main" com.univamu.parseFile.ParsingExceptionWithLinesRead
at com.univamu.parseFile.FileParser.readAll(FileParser.java:42)
at com.univamu.parseFile.Main.readFile(Main.java:12)
at com.univamu.parseFile.Main.main(Main.java:29)
Caused by: com.univamu.parseFile.LongLineParsingException
at com.univamu.parseFile.FileParser.readLine(FileParser.java:21)
at com.univamu.parseFile.FileParser.processNextLine(FileParser.java:52)
at com.univamu.parseFile.FileParser.readAll(FileParser.java:37)
... 2 more
```

Analyse du message d'erreur

- Ligne 1 : `ParsingExceptionWithLinesRead` exception à haut-niveau, c'est l'exception non rattrapée,
- Lignes 2 à 4 : la liste des méthodes ayant émise (`readAll`) et propagées (`readFile`, `main`) l'exception
- Ligne 5 : exception bas-niveau `LongLineParsingException`, c'est l'exception qui est à l'origine de l'erreur,
- Lignes 6 à 8 : méthodes l'ayant émise et propagée.
- L'exception bas-niveau est détaillée parce que elle a été déclarée comme cause de `ParsingExceptionWithLinesRead` dans son constructeur :

```
this.initCause(exc);
```

- Ces informations peuvent aussi être obtenues avec la méthode `printStackTrace` de la classe `Exception`.

Cause d'une exception

Les exceptions peuvent possèder une cause,

- de type `Throwable` ;
- accesseur `getCause`, mutateur `initCause` ;
- indique quelle erreur à provoquer en chaîne la production de cette erreur ;
- la première erreur n'a pas de cause (`null`).

Initialiser la cause

Lorsque qu'une exception est levée en réponse à une autre exception (dans un `catch`), il faut lui renseigner sa cause :

```
try {
   ...
} catch (SomeLowLevelException exc1) {
   SomeHighLevelException exc2 =
       new SomeHighLevelException();
   exc2.initCause(exc1);
   throw exc2;
}
```

La trace de la pile d'appels.

- Chaque méthode interrompue à cause d'une exception provoque la suppression d'une couche de la pile en mémoire.
- Les exceptions enregistrent les méthodes interrompues.
- `exc.printStackTrace()` permet d'afficher les méthodes interrompues.
- `exc.getStackTrace()` fournit le tableau des méthodes interrompues (objets de la classe `StackTrace`).

Difficultés supplémentaires.

Que se passe-t-il

- si les instructions d'un `catch` provoque une exception ?
- si les instructions d'un `finally` provoque une exception ?

La sémantique des exceptions est complexe. Conseil :

- limiter l'emploi des exceptions aux cas exceptionnels ;
- garder les instructions des blocs `catch` et `finally` aussi simples que possible.

Récapitulatif

Récapitulatif (I)

- On utilise les exceptions en cas de comportement empêchant l'exécution normale du programme.
- Les exceptions sont soumises à un mécanisme en 3 étapes : émission (`throw`), propagation, rattrapage (`catch`).
- Une exception non-rattrapée termine le programme.
- Les exceptions sont des objets, instances de `Exception` ou d'une de ces extensions.
- Une méthode qui émet ou propage des exceptions doit le déclarer (`throws`).
- Une méthode terminée par une exception (émission ou propagation) ne retourne pas de valeur. L'exécution du programme reprend dans un bloc `catch`.

Récapitulatif (II)

- `try catch` permet de rattraper les exception.
- Un `try` peut possèder plusieurs `catch` pour rattraper différents types d'exceptions.
- un `catch` rattrape les exceptions d'un type d'exception et de tous ses sous-types. Un `throws` déclare la propagation d'un type d'exception et de tous des sous-types.
- `finally` permet de garantir l'exécution d'instructions lorsque le contrôle sort d'un bloc, d'une méthode.
- Lors de la réémission d'une exception, il faut définir la cause de la nouvelle exception avec la méthode `initCause`.

Partie 11

Bonus : Lambda-calcul

Le λ-calcul

Définition

- Le λ-calcul (λ est la lettre grecque *lambda*) est un langage permettant de décrire des calculs.
- Il est défini par des expressions et par une règle de transformation sur les expressions.
- Il a été découvert par Alonso Church dans les années 1930.
- C'est une des premières définitions de la notion de calcul (qu'est-ce que calculer, que peut-on calculer).

Expressions du λ-calcul

Les expressions du λ-calcul (appelées λ-termes) sont de trois formes possibles :

(variable) une variable (x, y, ...) est une expression,

(application) si e_1 et e_2 sont des expressions, alors $(e_1\ e_2)$ est une expression,

(abstraction) si x est une variable et e est une expression, alors $(x \to e)$ est une expression.

Les règles se combinent pour donner des expressions arbitrairement complexes, comme

$$((x \to (z \to (f\ ((y \to z)\ x))))\ a)\ ((y \to y)\ b).$$

Signification des expressions

Le λ-calcul peut se voir comme une formalisation des notions de variables, de fonctions et d'applications des mathématiques.

(variable) une variable x dénote une valeur arbitraire,

(application) l'application $(f\ x)$, usuellement notée $f(x)$ en mathématiques, est l'application d'une fonction f à un argument x.

(abstraction) l'abstraction $(x \to e)$ dénote la fonction d'un argument x vers un terme e pouvant dépendre de x.

(La notation classique de $(x \to e)$ en λ-calcul est plutôt $\lambda x.e$)

La substitution

L'opération de substitution généralise le remplacement d'une variable par la valeur qui lui est attribuée.

e_1	e_2	$e_1[x \leftarrow e_2]$
$\cos\ x$	π	$\cos\ \pi$
$((f\ (g\ x))\ x)$	$(h\ y)$	$((f\ (g\ (h\ y)))\ (h\ y))$
$(y \rightarrow (x\ (x \rightarrow x)))$	f	$(y \rightarrow (f\ (x \rightarrow x)))$

- La substitution dans un λ-terme e_1, d'une variable x par un terme e_2, consiste en un λ-terme $e_1[x \leftarrow e_2]$ obtenu en remplacant chaque occurence libre de x dans e_1 par e_2.
- Une variable x est libre si elle n'apparait pas dans une abstraction ayant pour variable x (de la forme $(x \rightarrow e)$).

La réduction

La réduction est l'opération qui permet de calculer un λ-terme.

- Une réduction (plus précisément une β-reduction) dans un λ-terme t consiste à remplacer un redex $((x \to e_1)\ e_2)$ par le λ-terme $e_1[x \leftarrow e_2]$.
- Un redex (ou terme réductible) est un terme de la forme $((x \to e_1)\ e_2)$, avec x une variable arbitraire, et e_1 et e_2 des expressions arbitraires.

Exemples de *redex* : $((x \to f\ x)\ y)$, $(y \to (x \to y))\ (z\ t)$, $((y \to y)\ (x \to x))$

Exemples de réductions

Exemples :

$$((x \to f\ x)\ y) \longrightarrow_\beta (f\ y)$$

$$((y \to (x \to y))\ (z\ t)) \longrightarrow_\beta (x \to (z\ t))$$

$$((y \to y)\ (x \to x)) \longrightarrow_\beta (x \to x)$$

La réduction peut avoir lieu dans un sous-terme :

$$(z\ ((x \to (y \to (x\ y)))\ t)) \longrightarrow_\beta (z\ (y \to (t\ y)))$$

α-conversion

Il est interdit de substituer si une variable libre devenait non-libre :

$$((x \to (y \to (x\ y)))\ y) \longrightarrow_\beta (y \to (y\ y)) \qquad \text{Faux !}$$

(y était libre et devient non-libre)
Pour cela, il est autorisé de renommer la variable d'une abstraction (α-conversion), avant de réduire :

$$\begin{aligned}((x \to (y \to (x\ y)))\ y) &\equiv_\alpha ((x \to (z \to (x\ z)))\ y) \\ &\longrightarrow_\beta (z \to (y\ z))\end{aligned}$$

Valeurs et calculs

Une valeur est un λ-terme qui n'est pas réductible (il n'y a pas de réduction possible, même avec α-conversion).
Un calcul est une suite de réduction depuis un λ-terme jusqu'à obtenir une valeur.
Exemple :

$$\begin{aligned}(((x \to (y \to (y\ (x\ y))))\ t)\ z) &\longrightarrow_\beta ((y \to (y\ (t\ y)))\ z) \\ &\longrightarrow_\beta (z\ (t\ z))\end{aligned}$$

Une suite de réduction peut ne jamais terminer :

$$((x \to (x\ x))\ (x \to (x\ x))) \longrightarrow_\beta ((x \to (x\ x))\ (x \to (x\ x)))$$

Confluence du λ-calcul

- lors de la réduction d'un terme, il peut y avoir plusieurs *redexes* disponibles,
- la suite de réductions n'est donc pas unique,
- si un terme est réductible à deux valeurs v_1 et v_2 alors $v_1 = v_2$ (quelque soit la suite de réductions choisie, si elle termine le résultat sera le même).
- Il existe des termes admettant des suites de réductions qui termine et des suites de réductions qui ne terminent pas.

On appelle confluence la propriété que la valeur obtenue est la même pour toute suite qui termine.

Propriétés du λ-calcul.

Le λ-calcul

- est confluent,
- est Turing-complet : tout ce qui est calculable avec le λ-calcul est calculable avec une machine de Turing, et inversement,
- autrement dit, tout programme exécutable avec un ordinateur peut être écrit comme un λ-terme.

Le λ-calcul est donc un langage de programmation ! (certes extrêmement basique)

Programmer avec des λ-termes

Comment peut-on programmer avec, sans avoir défini de représentation de données (nombres, textes,...) ?

- On a définit une notion de valeur : les termes non-réductibles.
- On va choisir certaines de ces valeurs pour représenter nos données : par exemple une valeur par entier.
- On pourra alors trouver des fonctions particulières qui calculent les opérations usuelles, comme l'addition ou la division.

Trop de parenthèses

- Nous allons avoir besoin de λ-termes plus complexes. Les parenthèses sont compliquées à lire. Nous adoptons donc les conventions suivantes :
- Les λ-termes de la forme $((((f\ e_1)\ e_2)\ \ldots)\ e_n)$ seront écrites $(f\ e_1\ e_2\ \ldots\ e_n)$ (application d'une fonction f aux arguments $e_1, \ldots, e_n$).
- Les λ-termes de la forme $(x_1 \to (x_2 \to \ldots (x_n \to e)\ldots))$ seront écrites $(x_1\ x_2\ \ldots\ x_n \to e)$ (définition d'une fonction à n arguments).
- Une réduction peut donc être de la forme

$$((x\ y\ z \to e)\ a\ b\ c) \longrightarrow_\beta ((y\ z \to e[x := a])\ b\ c)$$

.

Exemple : encodage des booléens

On pose :

$$\begin{aligned} \text{true} &= (x\ y \to x) \\ \text{false} &= (x\ y \to y) \\ \text{if} &= (b\ y\ n \to (b\ y\ n)) \end{aligned}$$

Exemple :

$$\begin{aligned} (\text{if true } p\ q) &= ((b\ y\ n \to (b\ y\ n))\ \text{true}\ p\ q) \\ &\longrightarrow_\beta ((y\ n \to (\text{true}\ y\ n))\ p\ q) \\ &\longrightarrow_\beta ((n \to (\text{true}\ p\ n))\ q) \\ &\longrightarrow_\beta (\text{true}\ p\ q) \\ &= ((x\ y \to x)\ p\ q) \\ &\longrightarrow_\beta ((y \to p)\ q) \longrightarrow_\beta p \end{aligned}$$

amU Faculté des sciences Aix Marseille Université

Exercice

Exercice

Vérifiez le calcul précédent. Puis faites le calcul de
(if false p q) et vérifiez que cela donne q.

Opérations booléennes

On pose :

$$\text{not} = (b\ x\ y \to (b\ y\ x))$$
$$\text{and} = (b_1\ b_2\ x\ y \to (b_1\ (b_2\ x\ y)\ y))$$

Exemple :

$$\begin{aligned}
&(\text{and true true})\\
&= ((b_1\ b_2\ x\ y \to (b_1\ (b_2\ x\ y)\ y))\ \text{true true})\\
&\longrightarrow_\beta ((b_2\ x\ y \to (\text{true}\ (b_2\ x\ y)\ y))\ \text{true})\\
&\longrightarrow_\beta (x\ y \to (\text{true}\ (\text{true}\ x\ y)\ y))\\
&\longrightarrow_\beta (x\ y \to (\text{true}\ x\ y))\\
&= (x\ y \to ((x\ y \to x)\ x\ y))\\
&\longrightarrow_\beta (x\ y \to ((y \to x)\ y))\\
&\longrightarrow_\beta (x\ y \to x) = \text{true}
\end{aligned}$$

Exercice

Exercice

Vérifiez le calcul précédent. Puis donner une définition pour or et vérifier que $(\mathrm{or}\ \mathrm{false}\ \mathrm{false})$ se réduit en false.

Définitions alternatives

Nous aurions pu donner d'autres définitions :

$$\begin{aligned} \text{not} &= (b \to (\text{if } b \text{ false true})) \\ \text{and} &= (b_1\ b_2 \to (\text{if } b_1\ b_2 \text{ false})) \\ \text{or} &= (b_1\ b_2 \to (\text{if } b_1 \text{ true } b_2)) \end{aligned}$$

- Définitions utilisant des λ-termes déjà définis.
- Ces définitions alternatives sont plus *haut-niveau* que les précédentes : elles sont plus claires, bien que plus longue (en remplaçant if, true, false par leurs définitions).
- Ceci illustre la facilité du λ-calcul en terme de composition : composer des fragments de calculs pour construire des calculs plus complexes.

À quoi sert le λ-calcul ?

- Le λ-calcul fournit donc une base pour programmer.
- La programmation fonctionnelle est une discipline de programmation inspirée par le λ-calcul.
- Le λ-calcul est aussi fondamental en théorie des langages de programmation (la discipline de recherche visant à concevoir des langages de programmation mieux adaptés aux besoins).
- Il est aussi beaucoup utilisé en logique, et dans d'autres domaines.

La programmation fonctionnelle

En programmation fonctionnelle :

- le programme est organisé en fonctions, au sens mathématique du terme,
- il n'y a pas d'instruction ni structure de contrôle, un programme est une suite d'expressions à évaluer,
- l'évaluation se fait par substitution, comme en λ-calcul,
- il n'y a (*a priori*) pas de mémoire (pas d'affectation), pas d'entrée/sortie,
- les fonctions sont considérées comme des valeurs comme les autres : on peut les passer en arguments, les retourner comme résultat d'une autre fonction, les composer, ...

Quelques langages : Haskell, OCaml, Scheme
Purescript, LISP, Clojure

Exemple

La programmation fonctionnelle est déclarative : on déclare des valeurs, par des définitions, sans préciser comment les calculer.

```haskell
-- exemple : tri d'une liste en Haskell
sort :: [Integer] -> [Integer]
sort [] = []
sort [element] = [element]
sort (first:others) = sort smaller `append` sort bigger
  where
    smaller = first : filter (<= first) others
    bigger = filter (> first) others

filter :: (Integer -> Bool) -> [Integer] -> [Integer]
filter isValid [] = []
filter isValid (first:others)
  | isValid first = first : filter isValid others
  | otherwise = filter isValid others
```

Pourquoi cela nous concerne ?

- Les versions récentes de Java supportent partiellement la programmation fonctionnelle. Quasiment tous les langages populaires proposent maintenant un tel support.
- Plusieurs langages de programmation récents essaient de faire le pont entre programmation objet et programmation fonctionnelle (Swift, Scala, Kotlin).
- Les programmes (bien) écrits en fonctionnel possèdent des propriétés désirables :
  - plus composables,
  - plus faciles à tester,
  - plus faciles à maintenir, à refactoriser.

Java fonctionnel ?

Comment programmer *fonctionnellement* en Java ?

- Utiliser des objets persistents : toutes les propriétés doivent être finales !
- Utiliser des collections persistentes (malheureusement la librairie standard n'en propose pas).
- Représenter les fonctions comme des objets à une seule méthode.
- Utiliser des librairies adaptées (malheureusement rares).

Le support de Java pour ce style de programmation est donc assez limité.

Partie 12

Bonus : Conception et principes SOLID

Conception

Principe

Un programme propre :

- respecte les attentes des utilisateurs,
- est fiable,
- peut évoluer facilement/rapidement,
- est compréhensible.

En résumé :
Un programme informatique est de qualité si l'effort nécessaire à l'ajout d'une nouvelle fonctionnalité par un développeur extérieur au projet est faible.

Pourquoi programmer proprement ?

- Pour programmer les fonctionnalités les unes après les autres.
- Pour ajouter des fonctionnalités à moindre coût.
- Pour que les programmes soient utilisables plus longtemps.
- Pour valoriser le travail des développeurs (car réutilisables).
- Pour effectuer des tests à toutes les étapes du développement.
- Pour que les développeurs soient heureux de travailler.

Comment bien programmer ?

- Respecter les règles de nommage.
- Découper le programme en méthodes simples et compréhensibles et bien nommées.
- Respecter les règles d'indentation, d'espacement.
- Commenter intelligemment : les commentaires ne sont pas une excuse pour écrire un programme illisible.
- À la fin de chaque travail, prendre un temps de relecture et de réorganisation et renommage.

Références :

- *Clean Code*, de Robert C. Martin,
- Cours de Bertrand Estellon `http://pageperso.lif.univ-mrs.fr/bertrand.estellon/javaL3/cours2.pdf`
- Guide de la licence `http://pageperso.lif.univ-mrs.fr/~arnaud.labourel/dil_code_style/`

Diagrammes de classes

UML

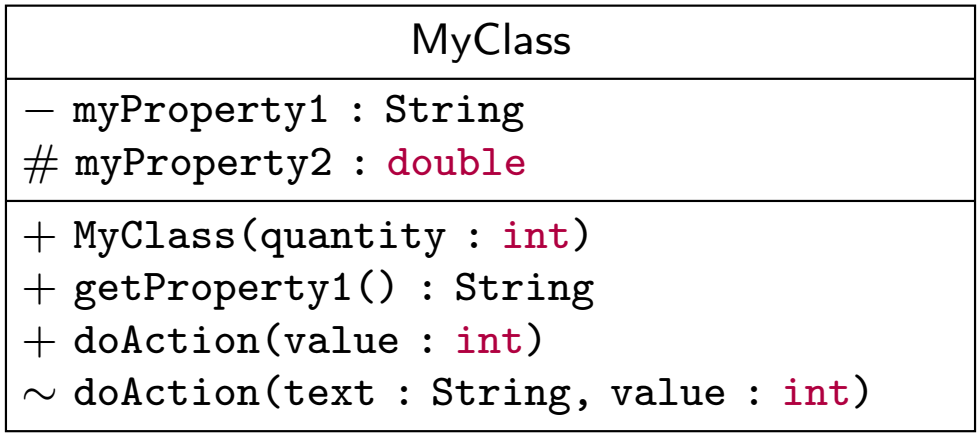
UML : *Unified Modeling Language*

- langage de modélisation graphique,
- principalement pour la conception orientée objet,
- spécifie comment représenter un projet avec 13 types de diagrammes.

Diagrammes de classes :

- représentent les classes et interfaces,
- et leurs interactions,
- sont statiques : représentent la structure du programme, et non pas son comportement à l'exécution.

Schéma d'une classe, exemple



- syntaxe d'annotation de type "`entité : type`",
- visibilités `public` (+), `private` (−), `protected` (#) et par défaut (∼).

Exercice

Exercice

Dans le transparent précédent, repérer la syntaxe pour les constructeurs, pour les méthodes de retour `void`, pour les autres méthodes.

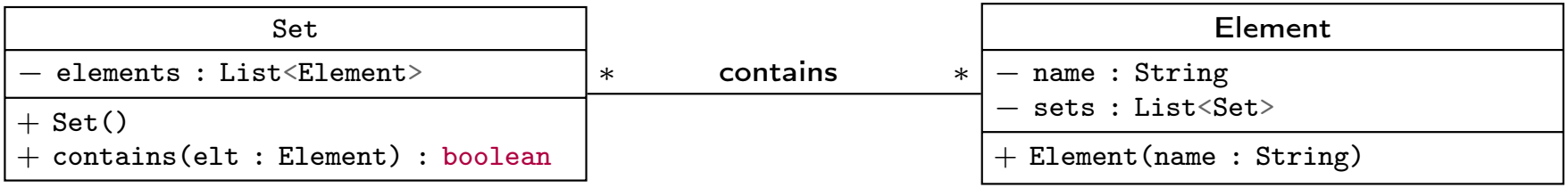
Association

Association

Deux classes sont en relation d'association si une instance d'une des classes peut interagir avec une instance de l'autre classe (typiquement par un appel de méthode)

- Une association est représenté par un lien dans le diagramme de classe.
- Des indices de cardinalités (0, 1, $*$, 1..n, *etc.*) permettent de préciser le nombre d'instances en interaction.
- Un terme peut préciser la sémantique de l'association.

Association, exemple de diagramme



Un ensemble contient de multiples éléments. Un élément peut être contenu dans de multiples ensembles.

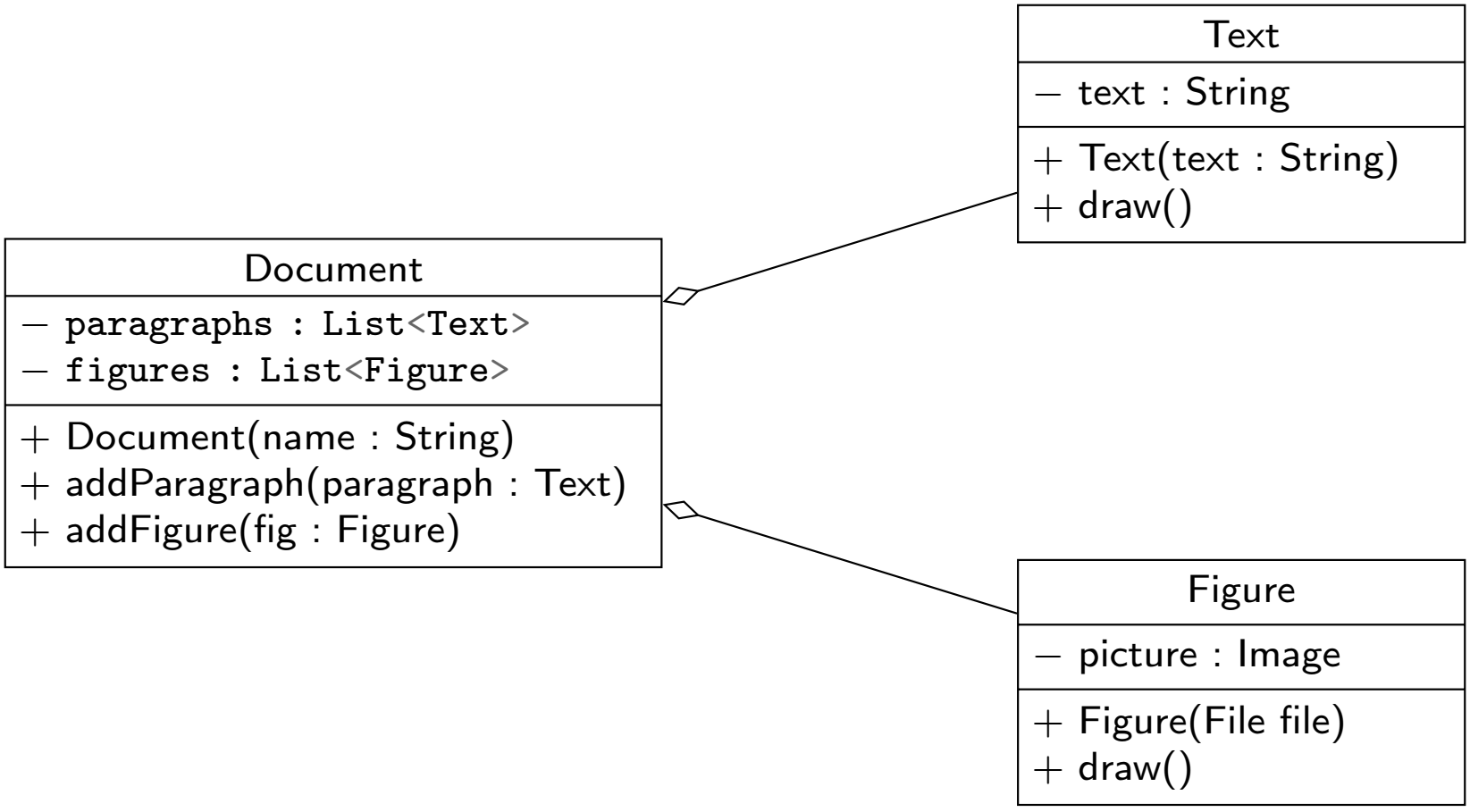
Agrégations

Agrégation

Une classe `A` est une agrégation de plusieurs autres classes si les instances de `A` contiennent des instances de ces autres classes.

- L'agrégation est représentée dans les diagrammes de classes par un lien terminé par un diamant vide.
- L'agrégation n'est pas propriétaire des objets agrégés : ils continuent d'exister si l'agrégation disparait.

Exemple d'agrégation



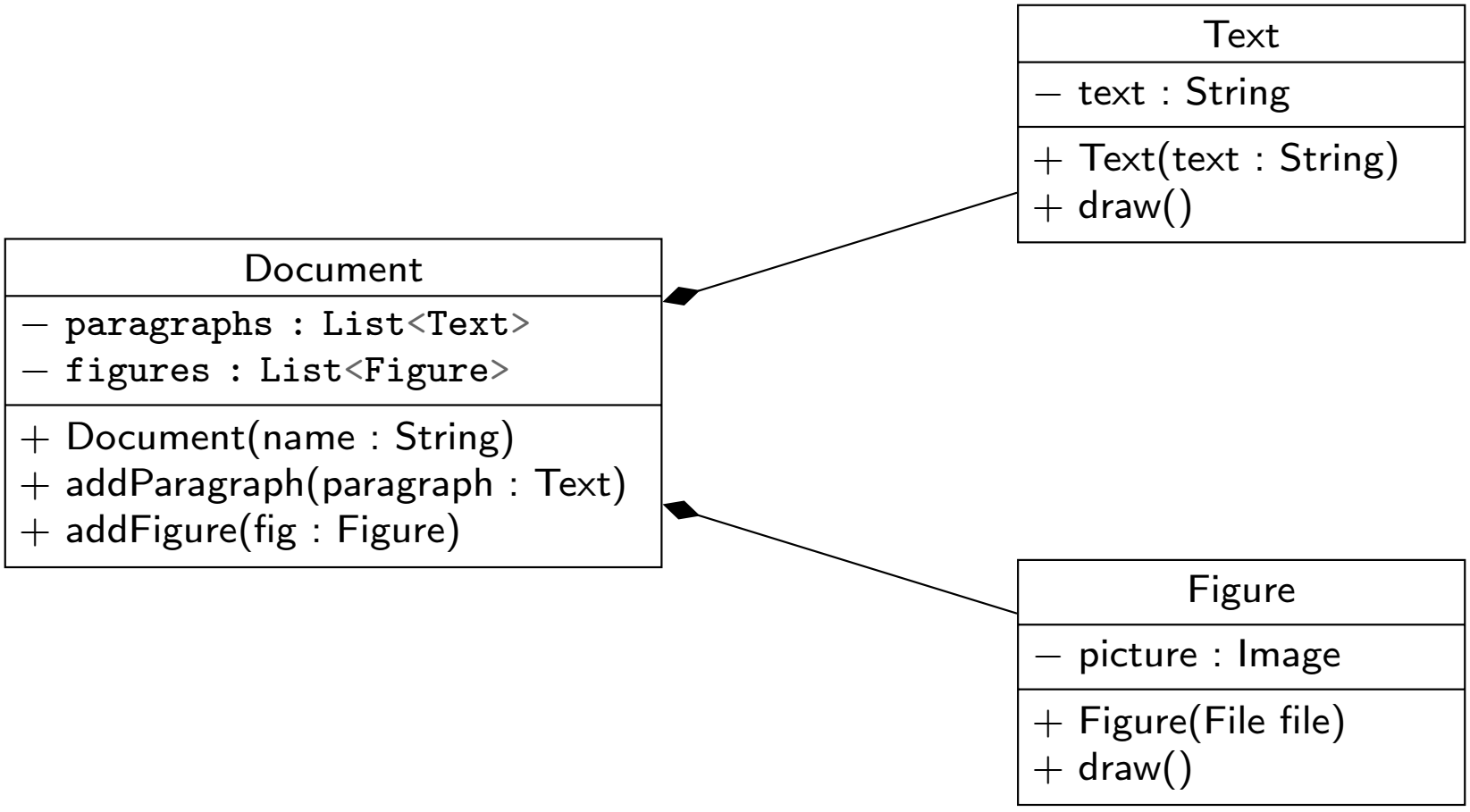
Composition

Composition

Une classe `A` est une composition de plusieurs autres classes si les instances de `A` sont composées d'instances de ces autres classes.

- La composition est représentée dans les diagrammes de classe par un lien terminé par un diamant plein.
- La composition est propriétaire des objets composés : ils cessent d'exister si l'agrégation disparait.

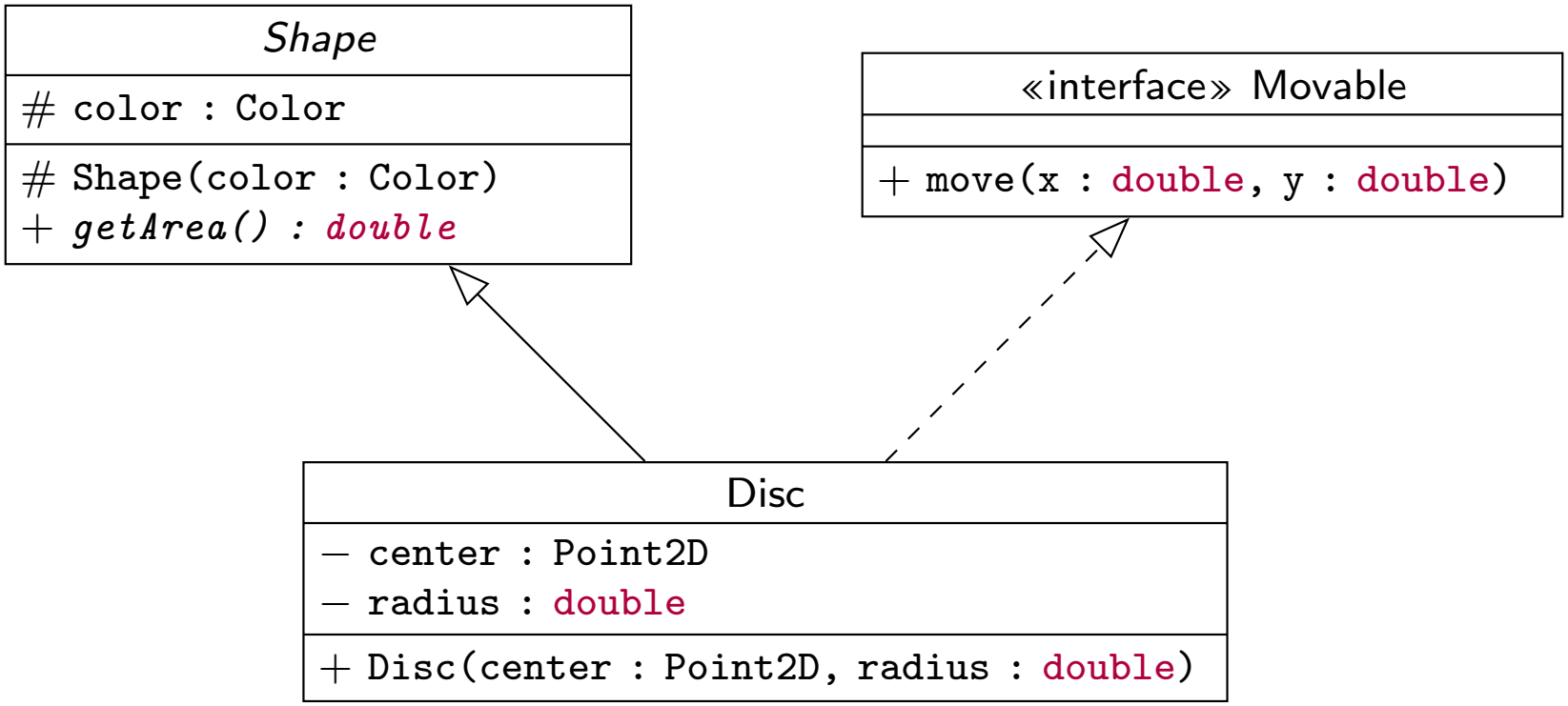
Exemple de composition



Extension et implémentation

- Les extensions sont représentées par des flèches continues.
- Les implémentations d'interfaces sont représentées par des flèches en pointillé.
- Pour les implémentations multiples, on utilise donc plusieurs flèches en pointillé.
- On n'écrit pas les méthodes et propriétés d'une classe lorsqu'elles proviennent de la super classe ou d'une interface implémentée.
- Les classes et méthodes abstraites sont écrites en *italique*.
- Le nom d'une interface doit être précédé par "«interface»"

Exemple d'extension et implémentation



Principes SOLID

Conception d'un programme

Un programme est "bien conçu" s'il permet de :

- Absorber les changements avec un minimum d'effort.
- Implémenter les nouvelles fonctionnalités sans toucher aux anciennes.
- Modifier les fonctionnalités existantes en modifiant localement le programme.

Objectifs :

- Limiter les modules impactés à chaque changement.
- Gagner du temps.

Le développement d'une application est une suite d'évolutions.

Les 5 principes SOLID

- SRP Single Responsability Principle : une classe n'a qu'une seule responsabilité,
- OCP Open/Close Principle : le programme est ouvert aux extensions, fermé aux modifications,
- LSP Liskov Substitution Principle : les extensions sont substituables à leur parent,
- ISP Interface Segregation Principle : les objets ne dépendent pas de méthodes qu'ils n'utilisent pas,
- DIP Dependency-Inversion Principle : les dépendances se font envers des abstractions, pas envers des implémentations.

Single-Reponsability Principle (SRP)

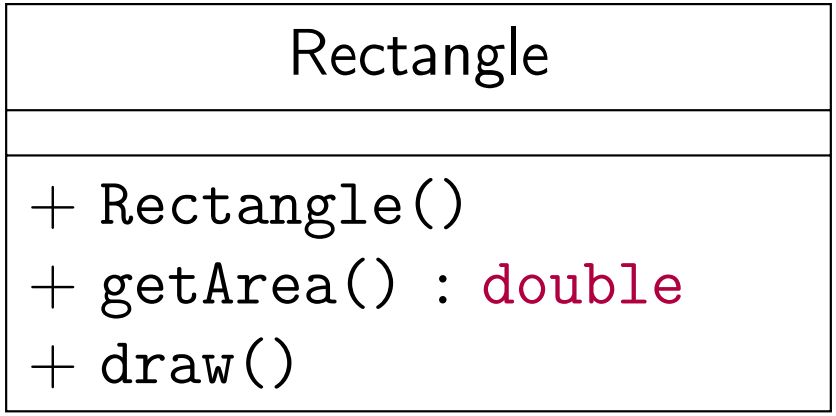
Une classe ou une interface ne doit avoir qu'une seule responsabilité :

- une responsabilité correspond à une raison de changer,
- si une classe a plusieurs responsabilités, elles sont couplées,
- modifier une responsabilité couplée risque d'altérer l'autre responsabilité, ce qui est indésirable (introduction de bug, besoin de plus de tests, perte de temps).

Solution :

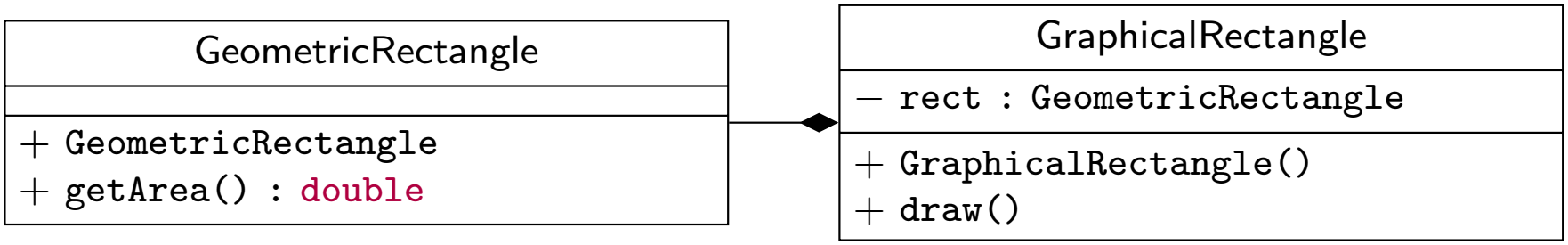
- Faire plusieurs classes : programme plus simple, plus facile à tester, plus modulaire, plus réutilisable.

Exemple de violation SRP



Couplage entre `draw()` (comment le rectangle est dessiné) et la représentation géométrique.

Réparation du principe SRP



On peut changer la représentation géométrique ou la méthode de dessin indépendemment.

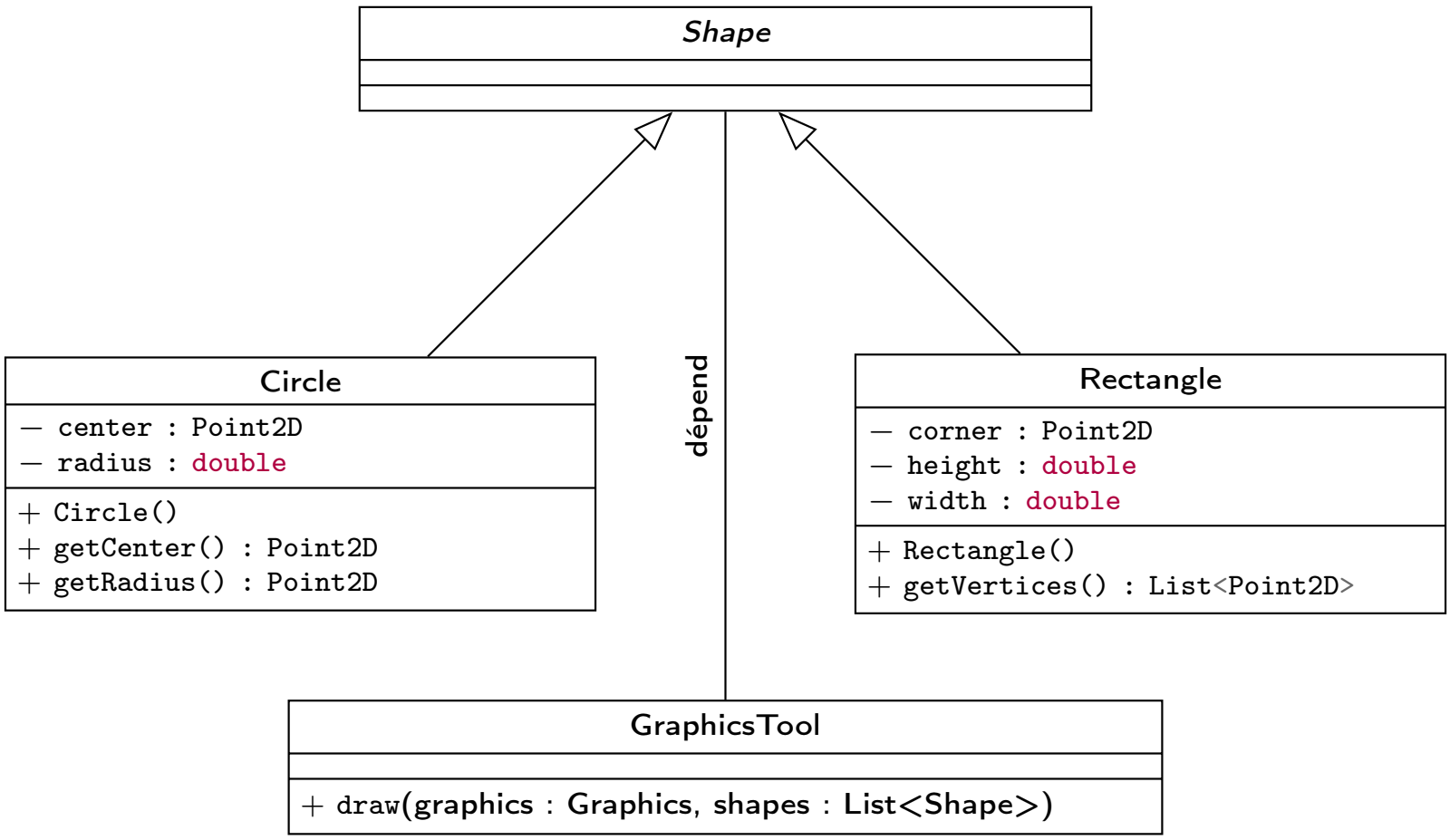
Open/Close Principle (OCP)

Programme ouvert pour l'extension, fermé pour la modification.

L'ajout de fonctionnalité doit se faire :

- en ajoutant de nouvelles classes (ouvert pour l'extension),
- sans modifier les méthodes et classes existantes (fermé pour la modification).

Exemple de violation OCP



Analyse de la violation OCP

```
public class GraphicsTool {
  ...
  public void draw(Graphics graphics, List<Shape> shapes) {
    for (Shape shape : shapes) {
      if (shape instanceof Circle) {
        ...
      }
      if (shape instanceof Rectangle) {
        ...
      }
    }
```

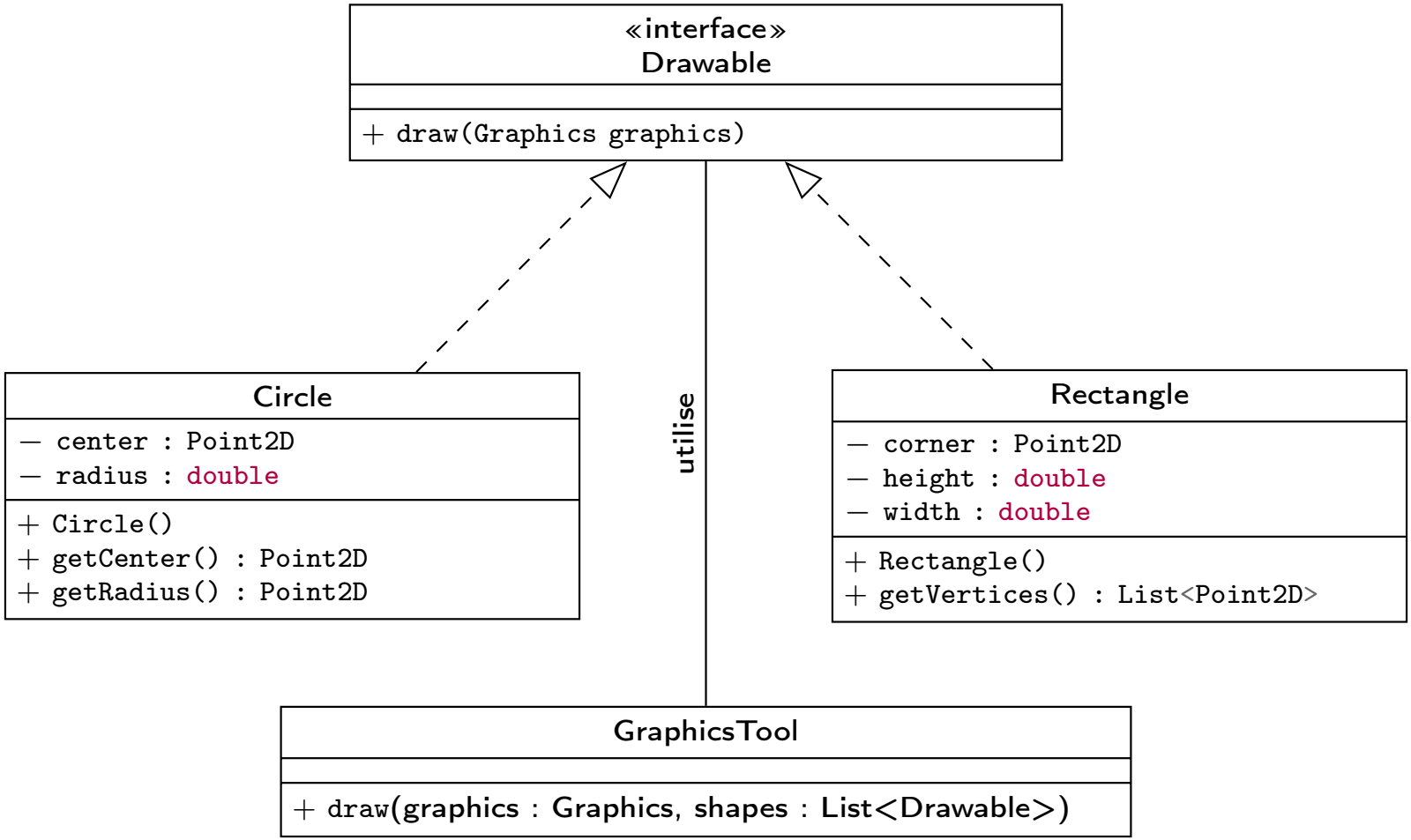
Problème :

- Ajout d'une nouvelle forme $\Longrightarrow$ modification de `draw`.

Solutions :

- Chaque objet se dessine lui-même,
- ou patron de conception *visiteur*.

Diagramme corrigé



Liskov Substitution Principle

Principe de substitution de Barbara Liskov : les extensions sont substituables à leur parent.

- si la classe `B` étend la classe `A`,
- et que le programme P manipule des instances de `A`,
- alors le même programme P avec des instances de `B` doit avoir le même comportement.

Avantages :

- plus simple, plus fiable, plus prédictible.

Exemple de violation LSP

```
public class Box {
  protected double width = 0;
  protected double height = 0;

  public void setWidth(double w) { width = w; }
  public void setHeight(double h) { height = h; }
  public double getArea() { return width * height; }
}

public class SquareBox extends Box {
  @Override
  public void setWidth(double w) { width = w; height = w; }
  @Override
  public void setHeight(double h) { width = h; height = h; }
}
```

Comportement violant LSP

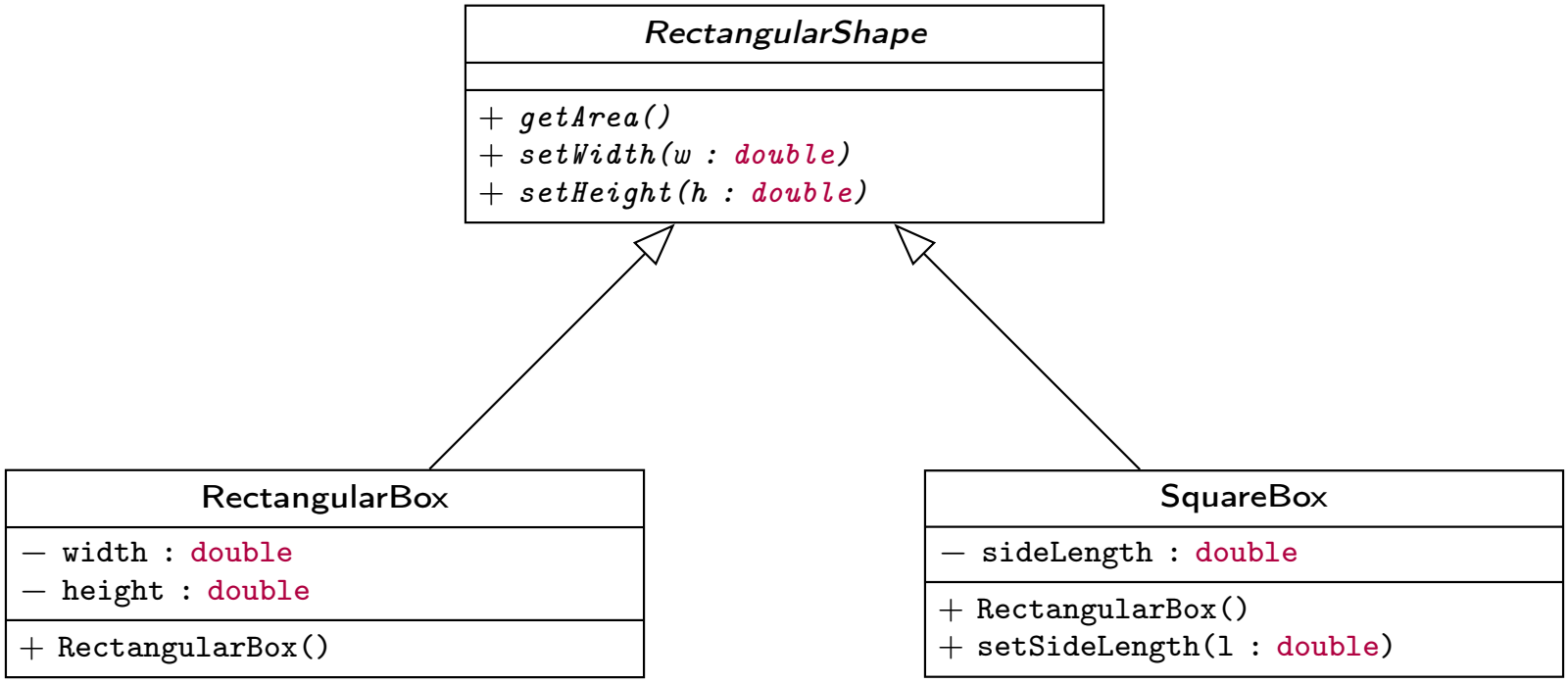
```
Box box1 = new Box();
box1.setWidth(3);
box1.setHeight(5);
int area1 = box1.getArea(); // 15

Box box2 = new SquareBox();
box2.setWidth(3);
box2.setHeight(5);
int area2 = box2.getArea(); // 25
```

Le comportement de box2 est inattendu en regard de son type !

Analyse de l'erreur.

- Une boîte carrée est une boîte,
- on aurait donc envie de faire une extension,
- mais la question n'est pas "une boîte carrée est-elle une boîte ?" mais "une boîte carrée a-t-elle le même comportement qu'une boîte ?",
- la réponse est **non**, donc pas d'extension.

Solution respectant LSP



Ségrégation des interfaces (ISP)

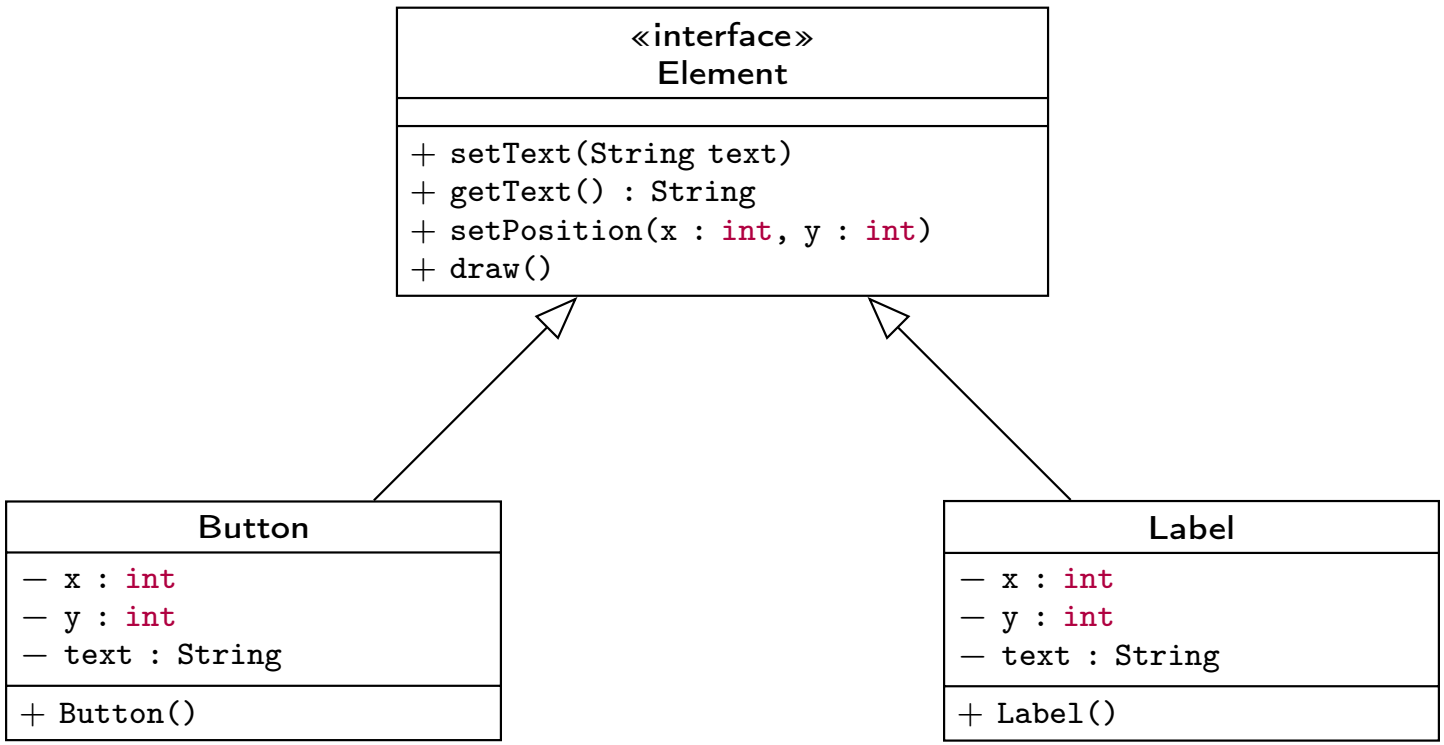
Principe de ségrégation des interfaces : une classe ne doit pas dépendre de méthodes qu'elle n'utilise pas.

- une classe doit implémenter toutes les méthodes de ses interfaces,
- y compris si ses méthodes ne lui servent pas,
- il est donc préférable de garder des interfaces minimalistes (avec peu de méthodes non-implémentées),
- et donc de découper les grosses interfaces en plusieurs petites interfaces (idéalement fonctionnelles).

Avantages :

- plus simple, plus facile à modifier, plus facile d'ajouter des fonctionnalités.

Exemple de violation ISP



Quel est le problème ?

On souhaite ajouter des éléments images.

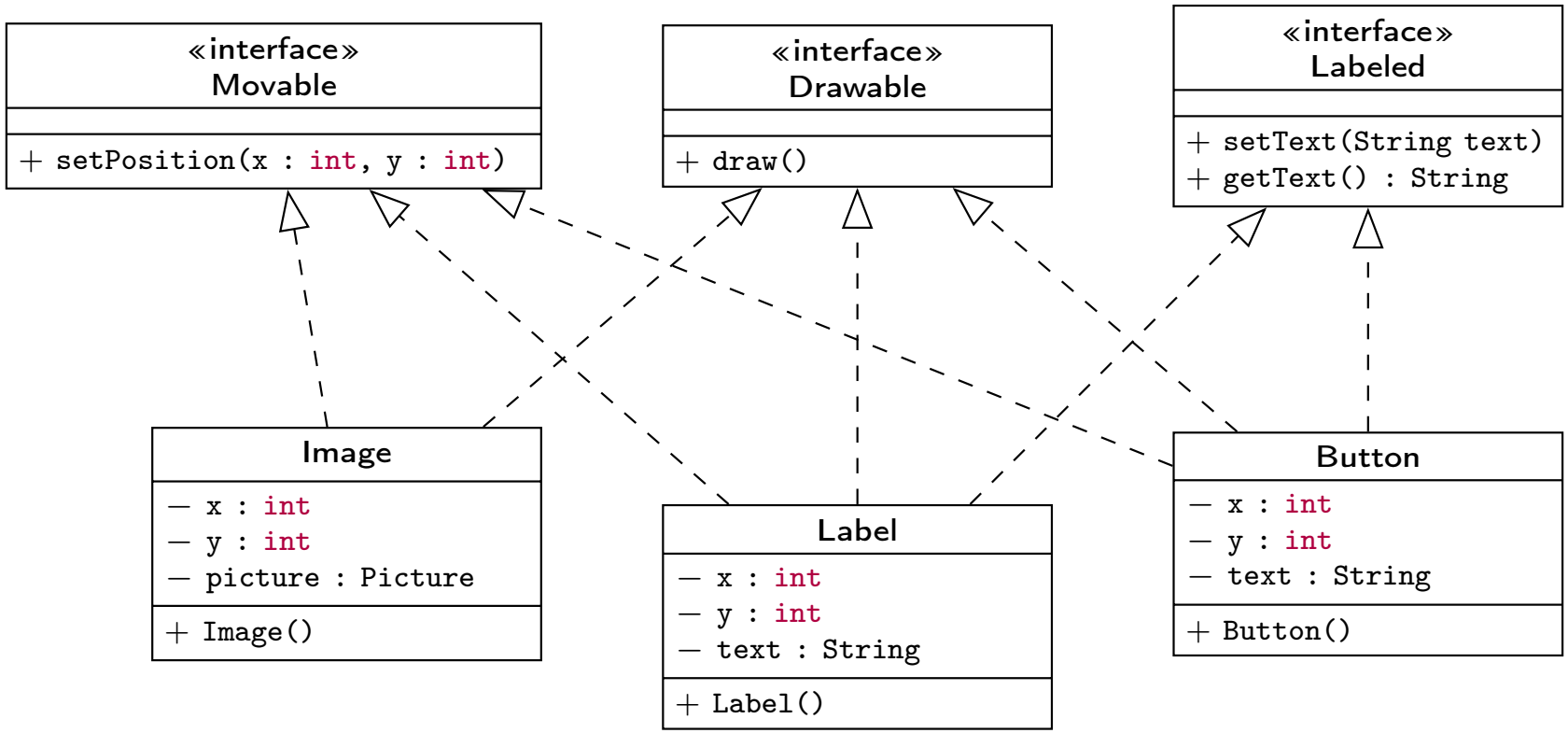
Problème : une image n'a pas de texte !

- Que faire dans `setText` et `getText` ?

Solution : Découper l'interface :

- `Drawable` pour la méthode `draw`,
- `Labeled` pour `setText` et `getText`,
- `Movable` pour `setPosition`.

Diagramme solution



Dependency-Inversion Principle (DIP)

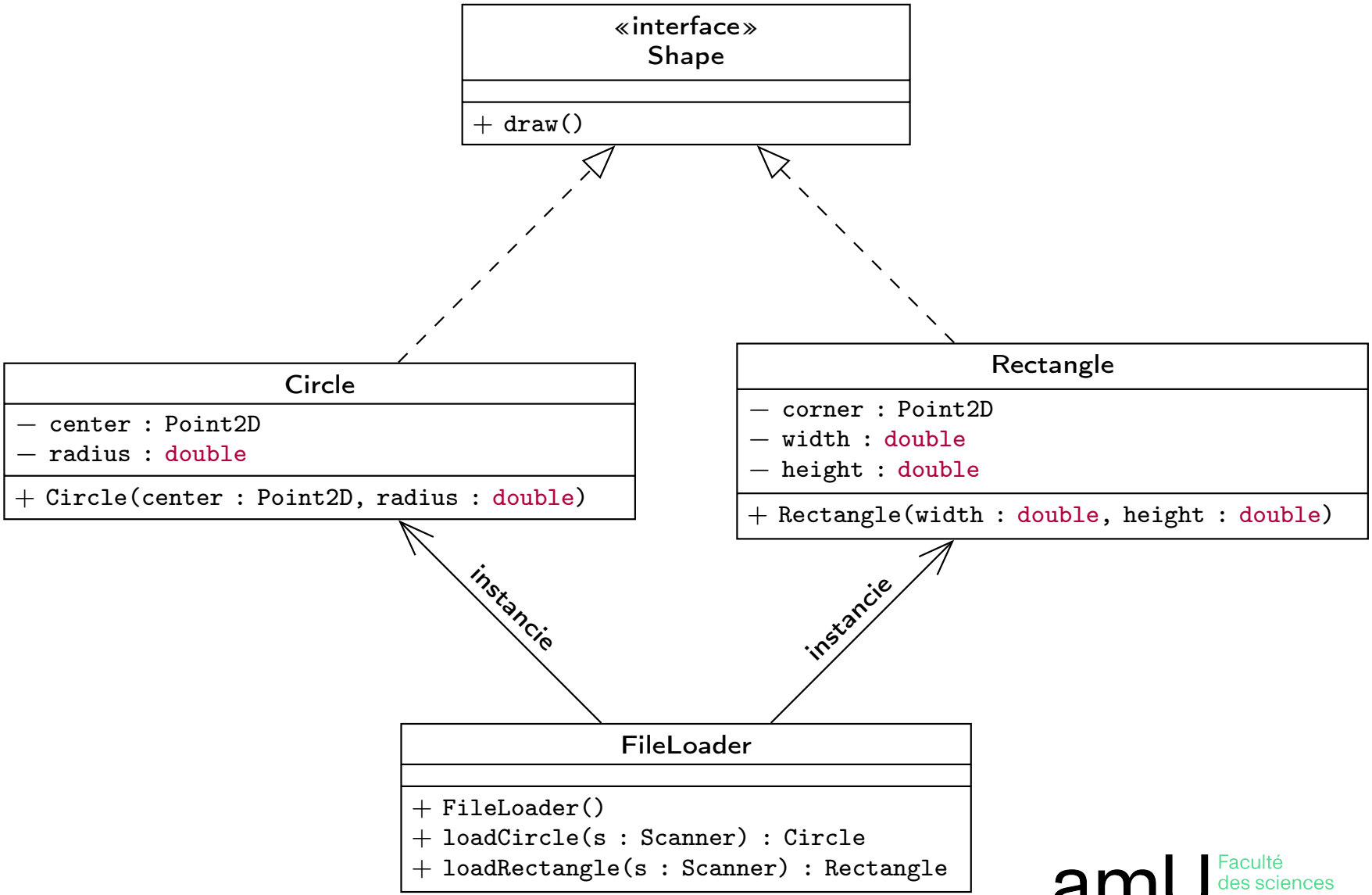
Principe de l'inversion des dépendances : le programme ne doit dépendre que d'abstraction, il ne doit pas dépendre d'implémentation.

- Les dépendances se font en terme d'interface.
- Une classe préfère utiliser une autre interface qu'une autre classe.
- On choisit des interfaces pour les types.

Avantages :

- Plus facile de réutiliser les entités, plus facile de les remplacer, plus facile de les tester.

Exemple de violation DIP



Problème

- Que se passe-t-il si on crée une extension `NewCircle` pour remplacer `Circle` ?
- il faut aussi modifier `FileLoader` pour que `loadCircle` retourne un `NewCircle`.
- c'est une violation de OCP, parce que DIP est violé : `FileLoader` dépend directement des autres classes.

Solution

- `FileLoader` devrait utiliser (et donc dépendre de) une instance de `AbstractShapeFactory` :

```
public interface AbstractShapeFactory {
  Rectangle createRectangle(double width, double height);
  Circle createCircle(Point2D center, double radius);
}
```

- `FileLoader` utiliserait alors une nouvelle implémentation de `AbstractShapeFactory` qui crée des `NewCircle`. Ainsi `FileLoader`, `Circle`, `OldShapeFactory` ne sont pas modifiés, `NewCircle` et `NewShapeFactory` sont créés.

Autre violation de DIP

- La classe `FileLoader` devrait être remplacée par une classe `ShapeLoader` capable de lire depuis diverses sources,
- `ShapeLoader` doit donc utiliser une instance d'une interface permettant la lecture (`Scanner`, `Reader`, ...),
- ceci rend la classe plus facilement réutilisable, le programme plus facilement modifiable.

Les principes déjà vus.

- DRY *Don't Repeat Yourself*, ne pas écrire deux fois la même chose, factoriser les points communs.
- un programme qui n'est pas DRY est WET, *Waste Everybody's Time*.
- KISS *Keep It Simple, Stupid* (ou *Keep It Simple and Stupid*), chercher la solution la plus simple possible, celle demandant le moins d'effort intellectuel à comprendre.

Partie 13

Bonus : Patrons de conception

Définition

Patron de conception

Un patron de conception (*design pattern*) :

- est une solution standard à un problème courant de conception,
- décrit un ensemble de classes, interfaces, et méthodes à implémenter,
- permet de valider les principes SOLID,
- est indépendant du langage de programmation.

Le terme peut aussi s'appliquer pour des langages non orientés objet, le patron s'exprime alors selon les concepts des langages visés.

Origine des patrons de conceptions

- Issus du savoir-faire d'experts,
- inspirés et validés par l'expérience de nombreux développeurs,
- formalisés en premier dans le livre *Design Pattern : Elements of Reusable Object-Oriented Software*, 1994.
- Notion d'anti-patron (*anti-pattern*) : erreurs de conception fréquentes.

Catégories de patrons

- patrons de création,
- patrons de structure,
- patrons de comportement.

Patrons de création

- Description de la fabrication des objets (création et initialisation).
- Découplage entre la définition/représentation des objets (classe ou interface) et leur création.
- Construction d'objets complexes, finement paramétrable (paramètres optionnels, optimisation de la construction, ...).

Exemples : Factory, Builder, AbstractFactory, ...

Patrons de structure

- Description des relations entre entités (objets, classes, interfaces).
- On parle ici de relations structurelles : implémentation, extension, composition, délégation, ...
- Réduire les dépendances entre ces entités.
- Rendre le programme plus simple à enrichir, à faire évoluer.

Exemples : Adapter, Composite, Decorator, ...

Patrons de comportement

- Description des interactions, des comportements de communication entre les objets du programme.
- Diminuer les dépendances entre les entités concernées.
- Simplifier ces interactions, en assurant le respect de SRP.
- Rendre le programme plus simple à enrichir, à faire évoluer.

Exemples : Strategy, State, Visitor, ...

Le patron Builder : problématique

Une classe complexe, compliquée à initialiser :

```
public class User {
  public String firstName; // obligatoire
  public String lastName; // obligatoire
  public int age; // optionnel
  public String phoneNumber; // optionnel
  public String email; // optionnel

  public User(String firstName, String lastName, int age, ...) {
    this.firstName = firstName;
    ...
    this.email = email;
  }
```

Analyse du problème

- Pour chaque choix de donner, ou pas, une valeur à une propriété optionnelle, il nous faut un constructeur : 2^n choix possibles.
- Ou bien, un seul constructeur, avec toutes les propriétés (pouvant être mise à `null`). Ingérable si le nombre de propriétés possibles est très grand !
- Si on ajoute plus tard de nouvelles propriétés, il faudra tout modifier : le constructeur, et toutes les utilisations du constructeur (violation OCP).

Solution

Séparation entre représentation et construction de l'objet.

- création d'une interface dédiée à la construction de l'objet,
- contrôle de la création (valeurs optionnelles ou obligatoires, validité de la valeur construite, multiples étapes de création).

La classe UserBuilder (I)

```java
public class UserBuilder {
  private final String firstName;
  private final String lastName;
  private int age = 0; // valeur par défaut
  private String phoneNumber = ""; // valeur par défaut
  private String email = ""; // valeur par défaut

  public UserBuilder(String firstName, String lastName) {
    this.firstName = firstName; // valeur obligatoire
    this.lastName = lastName; // valeur obligatoire
  }

  public User build() {
    return new User(firstName, lastName, age, phoneNumber, email);
  }

  ...
}
```

build permettra de créer l'objet User

La classe `UserBuilder` (II)

Pour les propriétés optionnelles, on dispose de *setters* :

```
  public UserBuilder setAge(int age) { // optionnel
    this.age = age;
    return this;
  }

  public UserBuilder setPhoneNumber(String phoneNumber) { // optionnel
    this.phoneNumber = phoneNumber;
    return this;
  }

  public UserBuilder setEmail(String email) {
    this.email = email;
    return this;
}
```

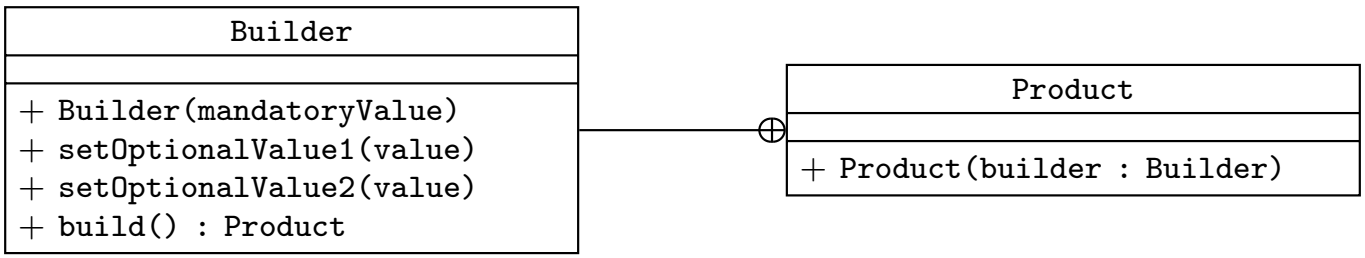
Utilisation d'`UserBuilder`

- Instanciation d'un `builder`,
- chaînage des méthodes optionnelles (c'est pour cela que les méthodes retournent `this`),
- extraction du `User` avec `build()`

```
User alice = new UserBuilder("Alice","Carroll").setAge(8).build();
User charles =
  new UserBuilder("Charles", "Dodgson")
    .setAge(65)
    .setEmail("charles.dodgson@wonderland.net")
    .build();
User john = new UserBuilder("John", "Doe").build();
```

Commentaires sur `Builder`

- Possible d'en faire une classe interne de `User` : `new User.Builder(...)` (UML : flèche ——⊕).
- Le constructeur d'`User` prend en paramètre l'instance de `Builder`.
- `StringBuilder` est un exemple d'utilisation de ce patron (permet d'optimiser les resources utilisées pour la création).
- En diagramme de classes :



Le patron Composite : problématique

On souhaite manipuler des formes. `Rectangle`, `Disc` sont déjà définies. Certaines formes sont définies par union ou par intersection de plusieurs formes.

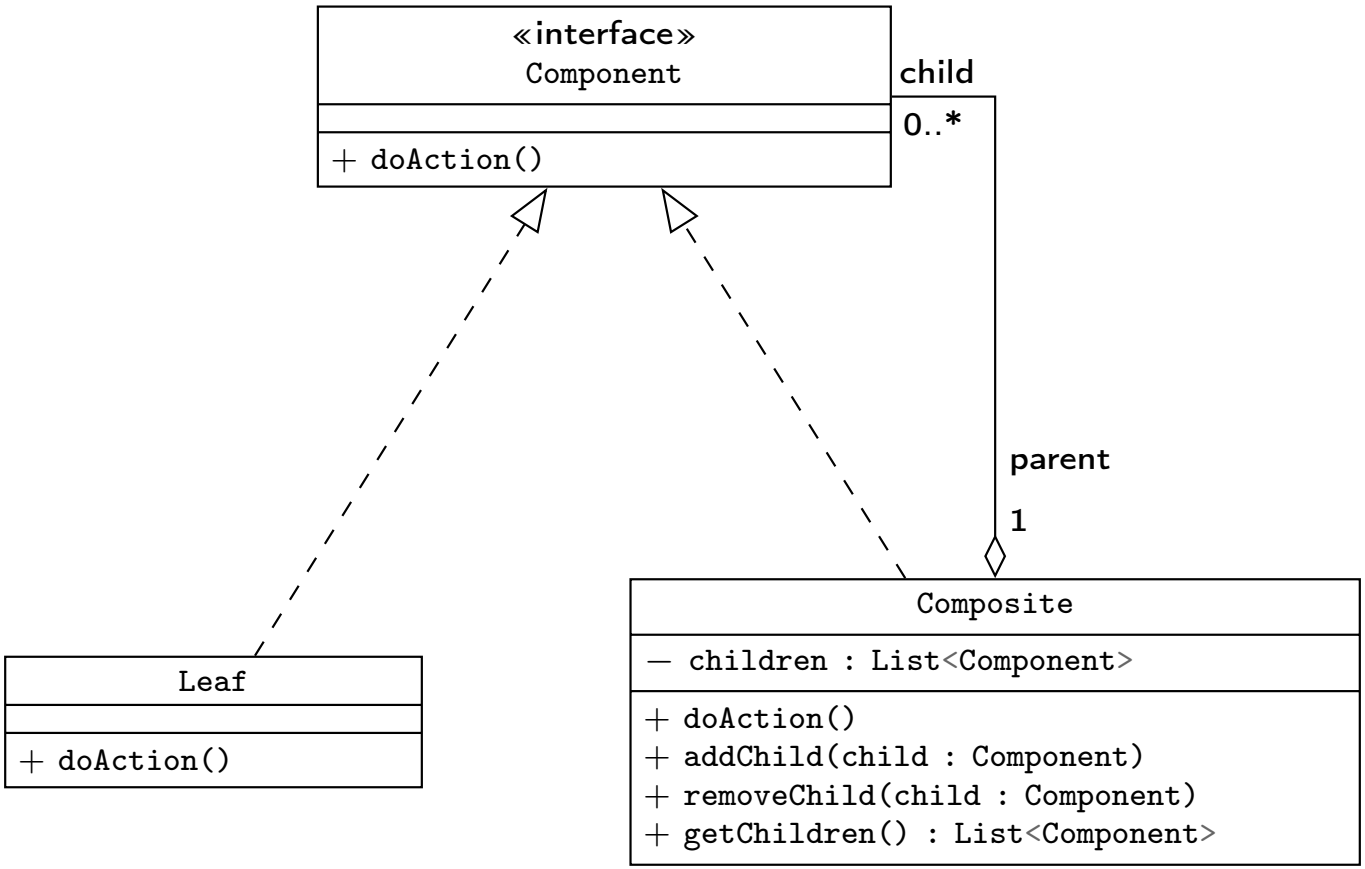
```
public interface Shape {
  boolean contains(Point2D point);
}

public class UnionShape implements Shape {
  private List<Shape> shapes;
  public UnionShape(List<Shape> shapes) { this.shapes = shapes; }

  public boolean contains(Point2D point) {
    shapes.stream().anyMatch(shape -> shape.contains(point));
  }
}
```

Analyse de la problématique

- Définition d'une structure d'arbre,
- Tous les nœuds sont des objets, avec une interface commune,
- Les opérations sont effectuées récursivement.

Autres exemples : formules arithmétiques, document HTML.

Solution : patron Composite



Le patron Decorator : problématique

```java
public class IntArrayStack {
  private final int[] stack = new int[100];
  private int size = 0;

  public int pop() {
    size = size - 1;
    return stack[size];
  }

  public void push(int value) {
    stack[size] = value;
    size = size + 1;
  }
}
```

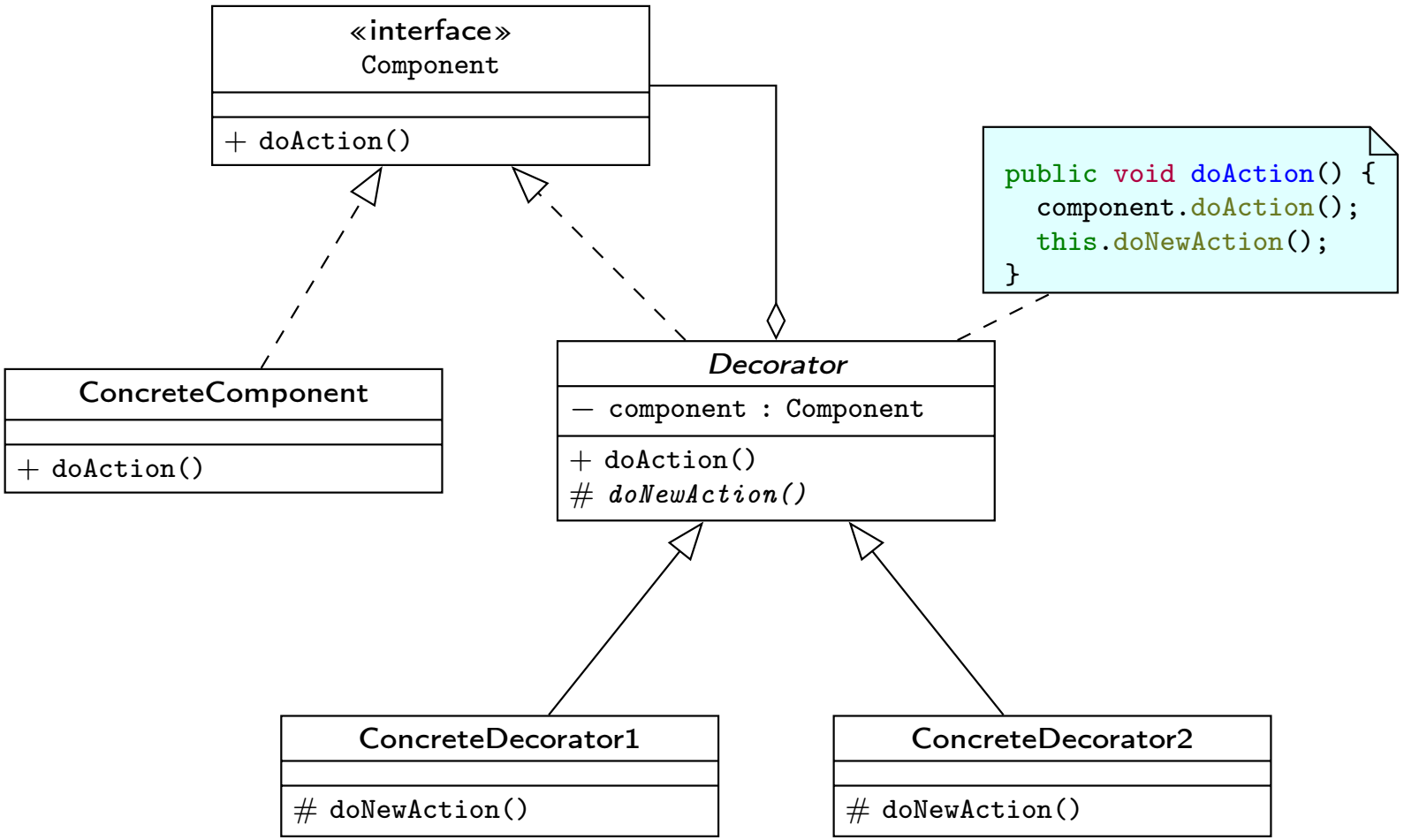
On veut ajouter des logs (dans `System.out`) à chaque opération de pile.

Analyse de la problèmatique

- Classe et méthodes pré-existantes,
- On souhaite ajouter un comportement aux méthodes (faire des statistiques sur les appels, écrire des logs, ...),
- On ne veut pas modifier la classe actuelle,
- On doit pouvoir ajouter plusieurs nouveaux comportements, selon les besoins.

Solution : Définir une nouvelle classe avec une composition de l'ancienne classe.

Diagramme de classes de Decorator



Interface du composant

On crée une interface (`Component`) pour généraliser la pile :

```
public interface IntStack {
  void push(int value);
  int pop();
}
```

L'implémentation devient notre base (`Concrete`) :

```
public class IntArrayStack implements IntStack {
  ... // comme avant
}
```

Classe abstraite Decorator

On abstrait les différents décorateurs :

```
public abstract class IntStackDecorator implements IntStack {
  private Stack stack;
  public StackDecorator(Stack stack) {
    this.stack = stack;
  }

  protected abstract void pushAction(int pushed);
  protected abstract void popAction(int popped);

  public void push(int value) {
    stack.push(value);
    pushAction(value);
  }
  public int pop() {
    int popped = stack.pop(value);
    popAction(popped);
    return popped;
  }
}
```

Réalisation de décorateurs

```
public class VerboseIntStack extends IntStackDecorator {
  public VerboseIntStack(IntStack stack) { super(stack); }
  public void pushAction(int pushed) {
    System.out.println("push(" + pushed + ")");
  }
  public void popAction(int popped) {
    System.out.println("pop(" + popped + ")");
  }
}

public class CountingIntStack extends IntStackDecorator {
  private int size = 0;
  public VerboseIntStack(IntStack stack) { super(stack); }
  public void pushAction(int pushed) { size++; }
  }
  public void popAction(int popped) { size--; }
  public int getSize() { return size; }
}
```

Utilisation

On peut décorer une pile avec un ou plusieurs décorateurs, ou de différentes façons.

```
IntStack stack = new IntStack();
VerboseStack verbose = new VerboseStack(stack);
CountingStack counting = new CountingStack(verbose);
stack.push(1);  // rien
verbose.push(2); // "push(2)"
counting.push(3); // "push(3)"
counting.push(4); // "push(4)"
verbose.pop(); // "pop(4)"
counting.getSize(); // 2
```

Le patron Visitor : Problématique

```
public interface Shape {
  double getArea();
  void draw(GraphicsContext context)
}

public class Circle { ... }

public class Rectangle { ... }
```

Que se passe-t-il si on veut :

- ajouter des méthodes à Shape ?
- ajouter des implémentations à Shape ?

Analyse de la problématique

	Rectangle	Circle	Pentagon	...
draw				...
getArea				...
contains				...
...				...

Une colonne par classe, une ligne par méthode.

- violation SRP (chaque classe a trop de responsabilité),
- violation OCP (trop de travail pour ajouter une classe).

Solution : patron Visitor

On fait de chaque méthode une classe. Une méthode est un visiteur. Les formes sont *visitées* par les méthodes, elles doivent accepter d'être visitées.

```
public interface Shape {
  <R> R accept(Visitor<R> ShapeVisitor);
}

public interface ShapeVisitor<R> {
  R visit(Rectangle rect);
  R visit(Circle circle);
}
```

Classe Circle

```
public class Circle implements Shape {
  private Point2D center;
  private double radius;

  public Circle(Point2D center, double radius) {
    this.center = center;
    this.radius = radius;
  }

  public Point2D getCenter() { return center; }
  public double getRadius() { return radius; }

  public <R> R accept(ShapeVisitor<R> visitor) {
    visitor.visit(this);
  }
}
```

Classe Rectangle

```
public class Rectangle implements Shape {
  private Point2D corner;
  private double width;
  private double height;

  public Circle(Point2D corner, double width, double height) {
    this.corner = corner;
    this.width = width;
    this.height = height;
  }

  public Point2D getCorner() { return corner; }
  public double getWidth() { return width; }
  public double getHeight() { return height; }

  public <R> R accept(ShapeVisitor<R> visitor) {
    visitor.visit(this);
  }
}
```

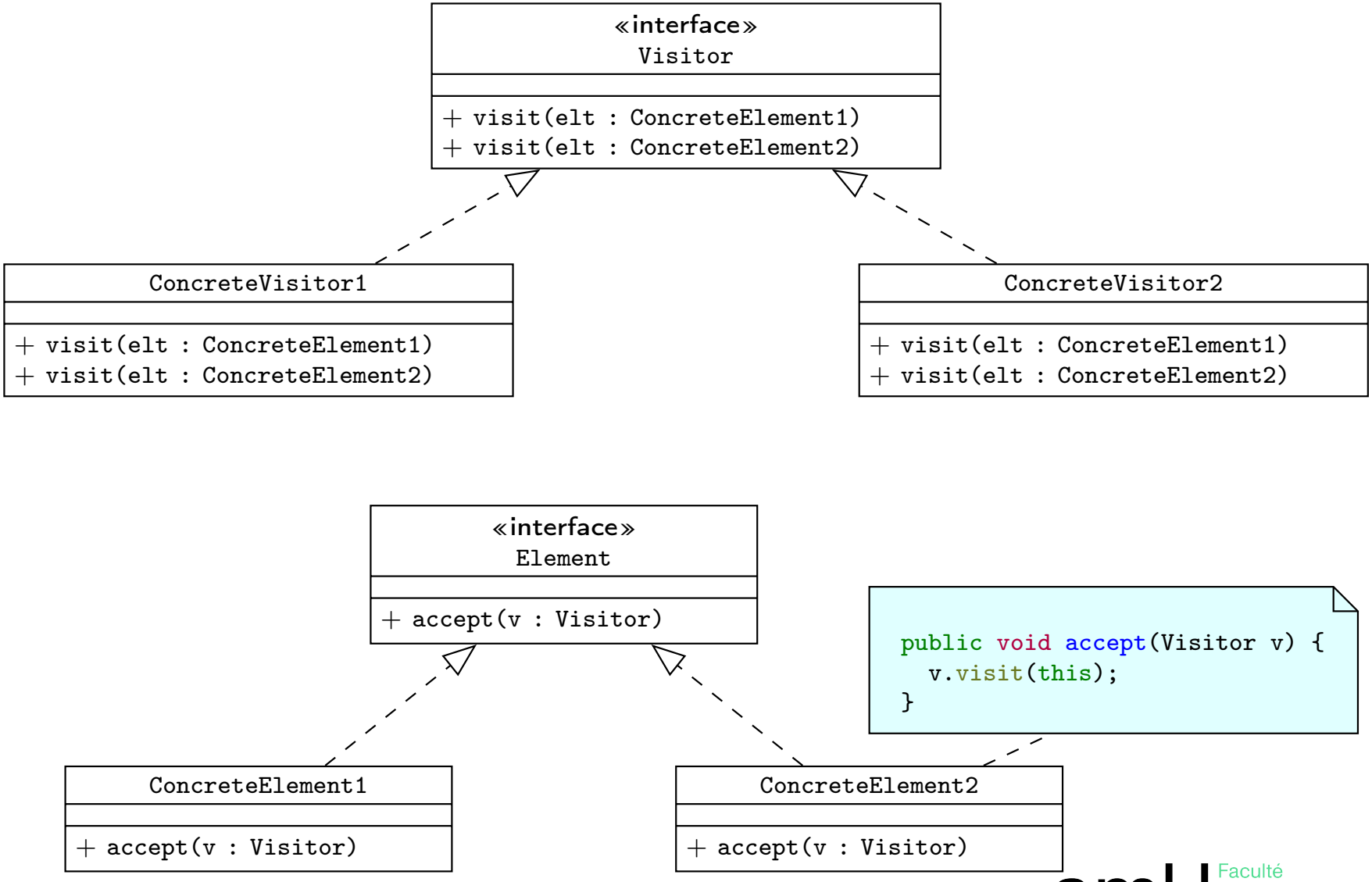
Implémentation de `getArea`

```
public class AreaVisitor implements ShapeVisitor<Double> {
  public double visit(Rectangle rect) {
    return rect.getWidth() * rect.getHeight();
  }

  public double visit(Circle circle) {
    return Math.PI * Math.pow(circle.radius,2);
}
```

Implémentation de `draw`

```
public class DrawerVisitor implements ShapeVisitor<Void> {
  private GraphicsContext graphics;

  public DrawerVisitor(GraphicsContext graphics) {
    this.graphics = graphics;
  }

  public void visit(Rectangle rect) {
    graphics.strokeRect(
      rect.getCorner().getX(), rect.getCorner().getY(),
      rect.getWidth(), rect.getHeight()
    );
  }

  public void visit(Circle circle) {
    graphics.strokeOval(
      circle.getCenter().getX(), circle.getCenter().getY(),
      circle.getRadius(), circle.getRadius()
    }

}
```

Diagramme de classes Visitor



Analyse de cette solution

- Facile de rajouter une méthode,
- Difficile de rajouter une forme (il faut changer tous les visiteurs).
- On l'utilise donc quand les classes visitées sont fixées, mais de nouvelles méthodes peuvent apparaître.
- Cas d'usage : définition de listes inductives, d'arbres inductifs (typiquement des représentations de documents, de programmes). Implémentation d'algorithmes récursifs sur ces listes et arbres.

Récapitulatif

- Les patrons de conceptions fournissent des solutions standards respectant SOLID,
- ils sont décrits par des diagrammes de classes,
- trois catégories : création, structure, comportement.

Cours *Programmation et Conception (L3 informatique)* : étude des patrons de conception.

Patrons d'architecture

- Niveau supérieur : le patron d'architecture propose un modèle d'organisation d'une application complète. Exemples :
- MVC modèle-vue-contrôleur (et ses variantes), pour les applications avec une interface utilisateur,
- ECS *Entity Component System*, pour les jeux vidéos.

Partie 14

Conclusion

Notions apprises

Cette année :

- structures de contrôle,
- tableaux,
- génériques (approfondissement),
- itérateurs,
- classes abstraites et extensions,
- sous-typage,
- exceptions,
- lambdas et streams,
- principes SOLID et patrons de conception.

Bilan

- 95% du langage Java couvert,
- 99.9% des concepts utiles en Java couvert,
- vous pouvez prendre un livre sur Java de niveau intermédiaire, et comprendre n'importe quel chapitre,
- usage basique de la librairie standard (collections, un peu d'IO, un peu de JavaFX).

Principaux manques : gestion de la concurrence, introspection.
Un cours complet de conception.

Que faire ensuite ?

- Se renforcer, notamment en conception. Quelques livres :
  - *Effective Java*, Joshua Bloch,
  - *Code Complete*, Steve McConnell,
  - *Design Patterns*, Gamma *et al.*.
- Apprendre à utiliser d'autres librairies.
- Apprendre d'autres langages, si possible très différents, par exemple :
  - bas-niveau : C
  - objet mais différemment : Elixir, Squeak
  - fonctionnel typé : Haskell
  - fonctionnel non-typé : Scheme
- Pratiquer ! C'est nécessaire si vous voulez progressez !