

LEPL1402 : Informatique 2

Synthèse

Professeur : Pierre Schaus

Année 2022-2023

Module 1 :

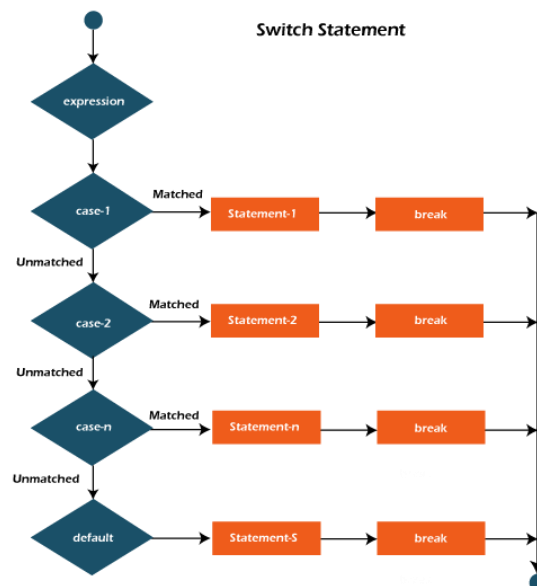
Java est un langage statiquement typé, çàd qu'on définit une variable avec son type (String, int,...), et qu'il est impossible de le changer

⇒ À l'opposé, Python, langage dynamiquement typé, permet de changer le type d'une variable sans problèmes

Différence entre une méthode statique et non-statique :

Si l'on souhaite utiliser la classe non-statique, on doit créer un objet de la classe en question. Elle doit être appliquée sur une instance de la classe. Alors qu'une méthode statique peut être utilisée indépendamment des instances et de la classe.

Méthode **void** : méthode qui ne retourne rien, pas de *return* dans le corps de la méthode



2 type d'égalités en Java :

1. Égalité de Référence : vérifie que 2 objets pointent vers la même référence
`==`
2. Égalité Logique : vérifie que 2 objets aient les mêmes attributs
`.equals()`

Les paramètres des méthodes des objets sont passés par RÉFÉRENCE, puisqu'on récupère une copie de la référence de l'objet. On ne peut pas modifier la référence en dehors de la méthode

`int a = b ? 2 : 5;` ➔ la variable a vaut 2 si b est true sinon elle vaut 5

Lorsqu'on introduit une variable à l'intérieur d'une méthode, on ne peut pas la redéfinir dans un niveau d'imbrication inférieur. On peut bien sûr la modifier mais pas la redéfinir.

A c = new C(); (C est une sous-classe de la classe B, qui est elle-même une sous-classe de A)

- ⇒ On déclare l'objet comme de type A, il prend le nom de « c », et on l'instancie avec la classe B()
- ⇒ On peut avoir des hiérarchies de classes qui s'étendent les unes les autres
- ⇒ On peut toujours déclarer un objet de type d'une classe ancêtre de l'instance de l'objet auquel on affecte la variable (qui est déclarée du type ancêtre)

Lorsqu'on déclare un objet de type d'une classe ancêtre à celle qui est instanciée, on ne peut pas utiliser les méthodes de la classe instanciée

Casting

Casting : transformation du type d'objets

((C) a).hello();

On force la transformation du type de a vers un type C, pour pouvoir utiliser la méthode *hello()*, qui est une méthode de la sous-classe C

Integer i = (Integer) p1.second;

- ⇒ On transforme le p1.second en type objet en entier pour que i soit bel et bien un entier

toString()

Java va implicitement par défaut étendre chaque classe que l'on crée avec la classe *Object*, qui est une classe prédéfinie dans Java. Toute classe étend implicitement la classe *Object*, qui est la classe-parent de toutes les classes de Java.

Le **System.out.println()** appelle implicitement la méthode **toString()**

Interface

Définir un comportement sans expliquer comment ce comportement va se réaliser (exemple : se déplacer, puisque tous les animaux au monde ne courent pas de la même manière)

- ⇒ On va donc définir la méthode **moveAround** sans l'implémenter == **INTERFACE**

Un objet peut implémenter plusieurs interfaces à la fois

Les interfaces sont donc fort utiles pour définir des abstractions mais elles ne contiennent pas l'implémentation

Classe Abstraite

Une **classe abstraite** est une classe possédant des méthodes implémentées et d'autres non-implémentées. C'est un petit mix entre Classe et Interface.

Le type-mère de tout les types de données en Java : le Type Objet

Comme vu auparavant, on ne peut pas transformer le type d'une variable en une autre. Si on souhaite le faire, on doit faire un casting

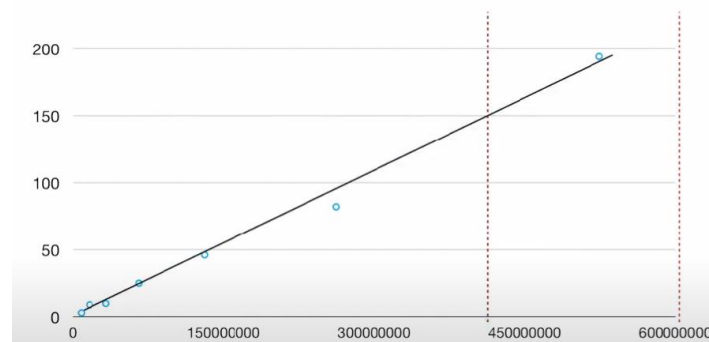
Module 2 :

⇒ Tous les programmes ne s'exécutent pas à la même vitesse

```
long t0 = System.currentTimeMillis();
...
long t1 = System.currentTimeMillis();
```

Permet de mesurer le temps que le bout de code entre les 2 instructions met à produire

La trajectoire de la fonction du temps mit en fonction de la taille de l'argument est presque linéaire → on a quelques fluctuations, qui dépendent de plusieurs choses (comme si on a fait tourner d'autres logiciels sur l'ordinateur)



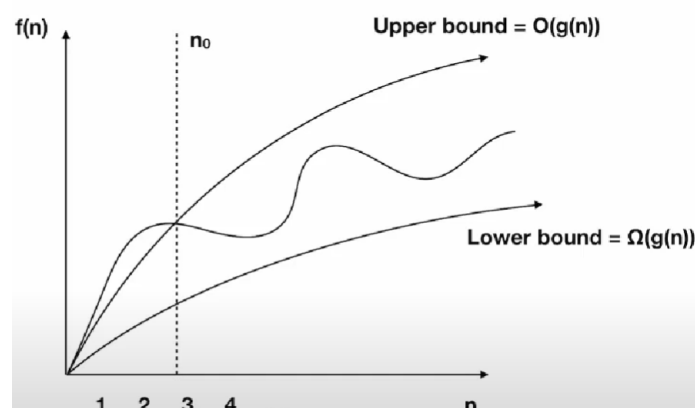
Est-il possible d'évaluer le temps que prendra un algorithme de manière théorique ?

- On va compter aussi précisément que possible le nombre d'étapes élémentaires de l'algorithme
- Il faut reconnaître que certains algorithmes ont un PIRE cas et un MEILLEUR cas

Borne supérieure et inférieure de l'algorithme

1. Borne supérieure : on dit que notre fonction $f(n)$ appartient à $O(g(n))$ si on peut borner par le dessus notre fonction $f(n)$ à partir de n_0
2. Borne inférieure : on dit que notre fonction $f(n)$ appartient à $\Omega(g(n))$ si on peut borner par le dessous notre fonction $f(n)$ à partir de n_0

⇒ S'il est possible de borner par le dessous et le dessus en même temps, alors la fonction $f(n)$ appartient à $\Theta(g(n))$



- $3n^2 - 100n + 6 \in \mathcal{O}(n^2)$ as rule is respected for $c = 3, n_0 = 0$:
 $3n^2 \geq 3n^2 - 100n + 6 \quad \forall n \geq 0$
- $3n^2 - 100n + 6 \notin \mathcal{O}(n)$ as its impossible to find c, n_0 such that
 $c \cdot n \leq 3n^2 - 100n + 6 \quad \forall n \geq n_0$
- $3n^2 - 100n + 6 \in \Omega(n)$ as $3n^2 - 100n + 6 \geq n \quad \forall n \geq 34$
- $3n^2 - 100n + 6 \in \Omega(n^2)$ as $3n^2 - 100n + 6 \geq n^2 \quad \forall n \geq 50$
- $3n^2 - 100n + 6 \notin \Omega(n^3)$
- $3n^2 - 100n + 6 \in \Theta(n^2)$ because function is both $\in \mathcal{O}(n^2)$ and $\in \Omega(n^2)$
- $3n^2 - 100n + 6 \notin \Theta(n^3)$ because function is $\notin \Omega(n^3)$

L'avantage de la notation O est qu'elle a la capacité de se simplifier très facilement :

- On doit uniquement garder le terme de la fonction qui croît le plus rapidement
- On peut se débarrasser des coefficients de multiplications sur les termes

For example

$$f(n) = c \cdot n^a + d \cdot n^b \quad \text{with} \quad a \geq b \geq 0 \quad \text{and} \quad c, d \geq 0$$

We have that

$$f(n) \in \Theta(n^a)$$

Even if c is very small and d very big!

$$\mathcal{O}(c \cdot f(n)) = \mathcal{O}(f(n)) \quad (\text{For a positive } c)$$

$$\mathcal{O}(f(n) + g(n)) \subseteq \mathcal{O}(\max(f(n), g(n))) \quad \text{Not true for multiplication, of course (what is the rule?)}$$

We can thus simplify this way:

$$\mathcal{O}(1) \subseteq \mathcal{O}(\log n) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(c^n) \subseteq \mathcal{O}(n!)$$

Usually, you must simplify if possible:

$$f(n) = n^4 - 10n^3 + 20n^2 + 8 \iff f(n) \in \mathcal{O}(n^4 - 10n^3 + 20n^2 + 8)$$

Is true, but $f(n) \in \mathcal{O}(n^4)$ is more concise

Recherche dichotomique

⇒ Permet de retrouver une valeur spécifique et son indice dans un tableau trié

Recherche séquentielle : parcourir chaque élément du tableau et voir si l'on trouve la valeur que l'on cherche → Meilleur cas $\Omega(1)$ et Pire cas $O(n)$ = Complexité générale $O(n)$

Recherche Dichotomique : on va au milieu du tableau et on regarde si notre valeur est plus petite ou plus grande, s'il elle est plus petite on va s'intéresser à la partie gauche de notre sous-tableau, et à la partie droite si la valeur est plus grande que la valeur du milieu. On fait cela récursivement → Meilleur cas $\Omega(1)$ et Pire cas $O(\log_2 n)$ = Complexité générale $O(\log_2 n)$

Pour trouver la complexité d'un algorithme récursif = formule de récurrence

Pour trier un tableau, il n'existe pas d'algorithme plus efficace (en termes de complexité) que le MergeSort

MergeSort : algorithme qui fonctionne récursivement en faisant 2 actions principales : diviser et fusionner.

BucketSort : algorithme encore plus efficace que le MergeSort avec une complexité de $O(n)$

⇒ Est en fait plus rapide que MergeSort pour de plus petites valeurs en entrée

Complexité Spatiale

Tout ce qui concerne la mémoire nécessaire pour qu'un algorithme s'exécute

Complexité spatiale \subseteq complexité temporelle → par exemple si on veut créer un tableau de taille n , on va le créer et on va utiliser une complexité temporelle d'au moins de n

3 types abstrait de données fréquemment utilisés :

1. **Bag** : collection non-ordonnée d'objets que l'on va stocker. On peut ajouter 2x un même élément (différence d'un bag avec un ensemble (*set*)). Pas moyen de supprimer un élément par contre.

Bag<Item>

2. **Queue** : collection ordonnée qui respecte le principe FIFO (*First In First Out*)

Queue<Item>

3. **Stack** : collection ordonnée qui respecte le principe LIFO (*Last In First Out*)

Stack<Item>

API = ensembles des méthodes d'une interface

Si l'on veut qu'une variable ne puisse pas être modifiée après son implémentation, on peut rajouter le mot devant *final*

2 stratégies pour se protéger d'une modification d'une liste/collection de données pendant qu'on est en train de l'itérer :

1. **Fail-Fast** : il lance une *CurrentModificationException* s'il y a une modification structurelle de la collection.
2. **Fail-Safe** : On n'itère pas sur la collection originale mais bien sur une copie de cette collection pour être sûr. On ne lance donc ici aucune erreur.

Les tableaux en Java, pas comme en Python, ont une taille fixée. Si l'on voit que notre tableau est rempli et n'a plus de cases vides, on va alors doubler la taille de notre tableau afin d'avoir de l'espace supplémentaire

⇒ Redimensionnement en $\Theta(n)$

Lorsqu'on a une complexité où il faut distinguer deux cas (un cas optimiste et un autre pessimiste), on parle de **complexité amortie/moyenne**

Invariant : formule ou propriété logique qui est vraie avant, pendant et après chaque exécution de la boucle → souvent pour prouver l'exactitude des algorithmes itératifs

Preuve par Induction : hypothèse inductive que l'on pose comme correcte. On utilise un cas de base ($i = 0$ ou 1) et puis on cherche à prouver pour $i + 1$

Module 3 :

Une **liste récursive** = liste d'un élément qui est imbriquée dans une liste d'un élément, qui est elle-même imbriquée dans un élément, etc...

Un arbre est soit :

- Une feuille, qui ne contient qu'un seul élément (feuille = qui n'a pas de sous-arbre)
- Une feuille avec un élément + une liste de sous-arbre récursif (forêt)

De quel parcours avons-nous besoin pour parcourir Le Binary Expression Tree ?

Le parcours Post-Ordre car on a besoin du résultat de l'arbre de gauche et du résultat de l'arbre de droite, et puis seulement on peut appliquer l'expression.

La méthode `.toString()` permet d'afficher la représentation String de n'importe quel objet

On exprime la complexité temporelle en termes de n (n = nombres de nœuds). On ne peut pas faire plus efficace que cette traversée puisqu'on visite le nœud 1 seule fois.

Sérialisation = stockage d'arbre binaires dans un fichier permettant de le réutiliser par après. On peut alors les partager plus facilement lorsque c'est sous forme d'un fichier. On représente l'arbre sous forme d'une chaîne, d'une String.

Les **fichiers XML** permettent de faire ce genre de choses.

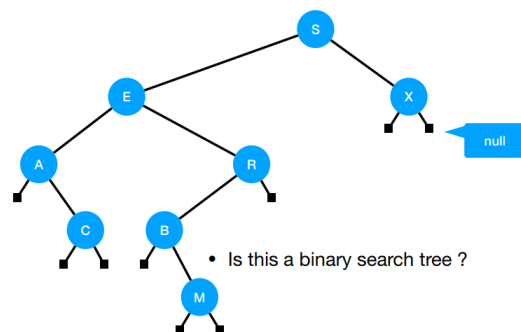
Désérialisation = Processus qui consiste à lire depuis le fichier l'arbre binaire pour le reconstruire par après.

Quel parcours choisir parmi les 3 disponibles ?

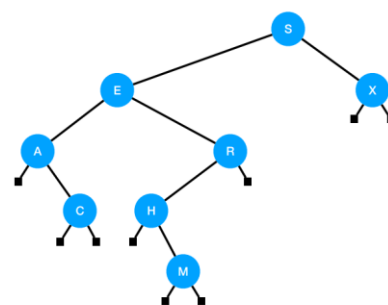
- ⇒ *Le parcours infixe ne va pas être pris car il y a une ambiguïté : On pourrait avoir 2 arbres différents qui nous donnerait le même parcours infixe en termes de résultat. Il est impossible de reconstruire de manière unique l'arbre de départ.*
- ⇒ *Pour le parcours postfixe et préfixe, c'est possible. Notre choix pour effectuer la sérialisation doit se diriger vers ces 2 parcours-là.*

Arbre Binaire de recherche = Arbre Binaire avec une « clé » dans chaque nœuds de sorte que

- Cette clé est plus grande que toutes les autres clés situées dans le sous-arbre de gauche
- Cette clé est plus petite que toutes les autres clés situées dans le sous-arbre de droite



Cet arbre est-il un arbre binaire de recherche ? Non car E est situé à gauche de B alors qu'il est plus grand que B. Propriété violée donc pas un arbre binaire de recherche



Celui-ci respecte bien les 2 règles donc est bel et bien un arbre binaire de recherche

Pour trouver la clé la plus petite de l'arbre, il suffit d'aller à la clé la plus à gauche possible de l'arbre. **Complexité en $O(h)$ où h est la hauteur maximale de notre arbre**

Pour énumérer de façon croissante toutes les clés, on utilise le parcours infixe (gauche, nœud, droite). **Complexité en $\Theta(n)$ puisque dans tout les cas, on doit visiter tous les nœuds**

Il est également possible de placer un ABR dans un tableau :

- On place le fils gauche du nœud à la $2*n$, où n est la position du nœud dans le tableau
- On place le fils droit du nœud à la $2*n + 1$, où n est la position du nœud dans le tableau
- ⇒ On commence la racine (premier nœud) à l'index 1
- ⇒ On peut retrouver le nœud parent d'un nœud en faisant index de l'enfant divisé par 2

4 grands principes de la Programmation Orientée Objet :

1. Encapsulation

Le fait que l'on puisse grouper dans un programme ensemble des variables et fonctionnalités dans une Classe, et ces variables pourront être accédées via des méthodes et on pourra modifier les objets grâce à cela.

Le constructeur a un rôle important en programmation orientée objet.

Une instance = un objet avec un comportement et un état

Il y a toujours un constructeur par défaut qui est créé par Java qui ne prends aucun paramètre

2. Abstraction

Le fait de cacher de certains détails à l'utilisateur. En POO, on peut le faire via des méthodes *public* et *private*

Le niveau d'abstraction le plus élevé en Java est l'interface

3. L'Héritage

Le fait de pouvoir réutiliser des méthodes et fonctionnalités d'une classe dans une autre classe grâce à l'Héritage : la classe-fille hérite des méthodes de la classe-mère.

Cela permet d'éviter le copier-coller en informatique, qui est du gaspillage de mémoire et de lignes de code.

4. Le Polymorphisme

Le fait de fournir une interface unique à des entités pouvant avoir différents types.

Module 4

On peut tester un programme uniquement s'il on sait exactement ce que l'on veut

Différentes possibilités pour spécifier ce qu'un programme doit faire :

1. *Spécification formelle*
2. *Document de spécification, ce que le client peut donner*
3. *Besoins des utilisateurs*

Le test doit montrer si le programme satisfait les **besoins fonctionnels** ou **les besoins non-fonctionnels** (complexité, vitesse du programme, langage de programmation utilisé, ...)

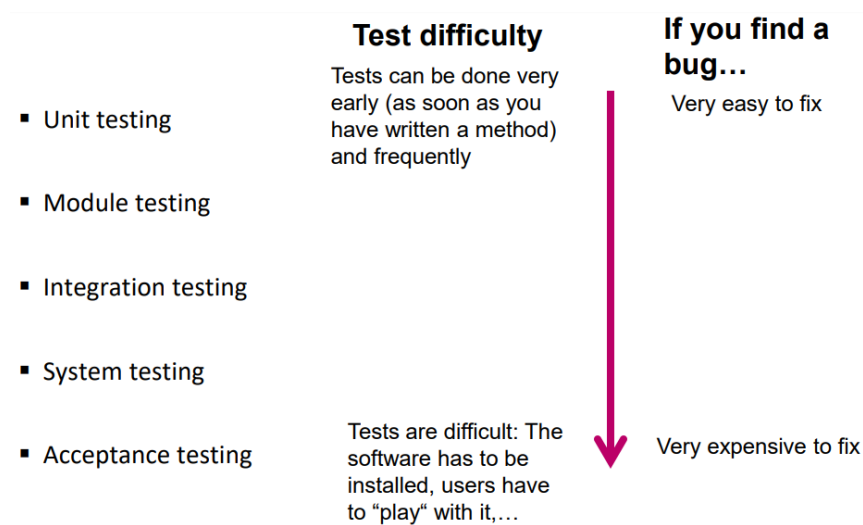
On doit tester un programme avec des valeurs d'input qui proviennent de son domaine d'input

⇒ On regarde le domaine d'input, les sous-domaines intéressants, les cas spéciaux

On peut tester un programme à différents niveaux :

- **Unit Testing** : test d'une simple méthode

- ⇒ Réalisé par l'auteur du code
- **Module Testing** : test d'un module (en java, module = classe)
- ⇒ Réalisé par l'auteur du code
- **Integration Testing** : test d'un ensemble de modules ensembles
- ⇒ Réalisé par l'équipe de développeurs
- **System Testing** : Test de l'ensemble du programme
- ⇒ Réalisé par l'équipe de tests
- **Acceptance Testing** : Essai chez le client
- ⇒ Réalisé par le client



Les tests sont quelque chose de répétitif : on doit tester notre programme à chaque petit changement, on doit le faire pour chacun de nos programmes,...

➔ Il faut essayer d'automatiser nos tests le plus possible

On peut alors utiliser le module JUnit :

```
import static org.junit.Assert.*;

public class MainTest {

    @org.junit.Test
    public void testFirstNumberLessThanSecondNumber() {
        assertEquals("Minimum of 3 and 5 should be 3", 3, Main.min(3,5));
    }

    @org.junit.Test
    public void testFirstNumberGreaterThanSecondNumber() {
        assertEquals("Minimum of 5 and 3 should be 3", 3, Main.min(5,3));
    }
}
```

Comment savoir s'il on a effectué assez de tests ?

On peut différencier 2 types de tests :

Blackbox Test

Test où l'on ne regarde pas au code source du programme, on regarde juste ce que le programme retourne

WhiteBox Test

Test qui va tester le code source du programme

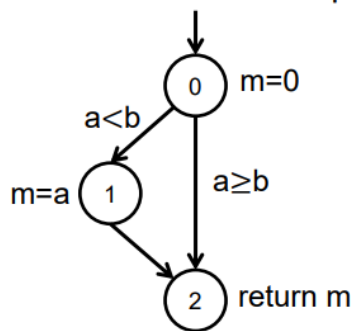
Le **Control Flow Graph** permet de nous dire si nos tests sont complets et que l'on teste bien tous les cas possibles

Est-ce que si l'on a 100% de node coverage, est-on sûr que notre programme est correct ?

Pas forcément, à la place d'avoir 100% de node coverage, c'est mieux d'avoir 100% de edge coverage.

- If we have 100% node coverage, can we be sure that our program is correct?

```
int min(int a, int b) {
    int m=0;
    if(a<b) {
        m=a;
    }
    return m;
}
```



- We can cover all statements with the test case a=3, b=5
 → Program goes through nodes 0,1,2 ⇒ 100% node coverage
 But: We have not tested the direct path 0→2 !

Mais même avec 100% de edge coverage, on est même pas sûr d'avoir un programme fonctionnel à 100% !

- ⇒ Pour être sûrs que notre programme est correct, nous devrions tester tous les chemins possibles dans le code, mais cela n'est pas forcément faisable dans un programme contenant des boucles. Dans ce cas, on se contente du node et edge coverage

Conseils sur les tests :

- ⇒ On doit écrire du code qui est facilement testable, et qui peut facilement produire des valeurs de test
- ⇒ Si on test des méthodes non-statique, on doit travailler avec des objets
- ⇒ On doit tester comme dit précédemment les différents cas possibles

Un **Design Pattern** est un arrangement de modules, reconnu comme une bonne pratique en réponse à un problème de conception d'un logiciel.

Dans les pages qui vont suivre, on va voir 4 design pattern différents

1. Singleton Design Pattern :

Lorsque l'on a une méthode qui est dite privée, on ne peut pas appeler le constructeur de cette méthode. On peut juste utiliser la méthode.

Les auteurs de la classe privée voulaient être sûr en la mettant en *private* qu'il n'y avait qu'une et un objet de cette classe → classe Singleton

C'est un peu comme une variable globale, il n'y a qu'un seul objet de la classe Singleton dans le programme entier

Ces singletons sont populaires en Java : quelques exemples = Runtime, Desktop

Les Singletons ont du sens si on a certaines ressources qui n'existent qu'une seule fois (Runtime). On utilise les Singletons uniquement si c'est absolument nécessaire !

La *Lazy Initialisation* permet de créer un objet Singleton uniquement on en a réellement besoin

```
public class Runtime {  
    // don't create the object at the beginning  
    private static final Runtime currentRuntime = null;  
  
    private Runtime() {}  
  
    public static Runtime getRuntime() {  
        if(currentRuntime==null) {                // already created?  
            currentRuntime=new Runtime();        // create now  
        }  
        return currentRuntime;  
    }  
  
    public long totalMemory() { ... }  
}
```

2. Factory Design Pattern

Les *Factories* (usines) sont utiles lorsque l'on veut créer des objets mais que l'on ne sait pas quelle sorte d'objets on veut créer.

En créant une « usine », notre classe Zoo est devenu indépendante des sous-classes qui créaient des objets pour le Zoo. Les utilisateurs peuvent maintenant utiliser leur classe Zoo avec de nouveau animaux qu'ils définissent eux-mêmes, selon leurs envie.

3. The Observer/Observable Design Pattern

ActionListener est une interface à une méthode

Dans cette interface, on a 2 sortes d'actions :

- **Les actions d'observation** : qui attend qqch pour produire résultat (ex : ActionListener)
- **Les actions observables** : qui produit un résultat visible à l'œil nu (ex : bouton)

Le Observer/Observable Design Pattern permet de séparer les objets contenant des données et le code réagissant aux changements de ces données. La classe

Seul petit bémol : le programme devient plus dur à comprendre.

4. Visitor Design Pattern

Dans le **Visitor** Design Pattern, les structures de données complexes autorisent les visiteurs de « visiter » leurs éléments. Cela permet d'utiliser certaines fonctionnalités ou de faire certaines choses que nous ne voulons pas mettre dans la classe.

```
public interface Visitor {
    public void visit(Wheel wheel);
    public void visit(Engine engine);
    public void visit(Car car);
}

public class PrintVisitor implements Visitor {
    @Override
    public void visit(Wheel wheel) { System.out.println("A wheel"); }

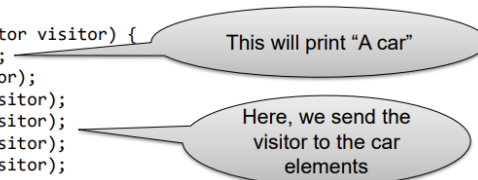
    @Override
    public void visit(Engine engine) {
        System.out.println("Engine with "+engine.getHP()+" hp");
    }

    @Override
    public void visit(Car car) { System.out.println("A car"); }
}
```

On a également une interface **Visitable** qui permet de dire si on accepte que le visiteur puisse faire tel ou tel chose. On utilise le mot-clé `.visit`. Les éléments de l'interface Visitable ne savent pas ce qu'ils acceptent ou non.

```
class Car implements Visitable {
    private Engine engine=new Engine(90);
    private Wheel[] wheels=new Wheel[] {
        new Wheel(), new Wheel(),
        new Wheel(), new Wheel()
    };

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
        engine.accept(visitor);
        wheels[0].accept(visitor);
        wheels[1].accept(visitor);
        wheels[2].accept(visitor);
        wheels[3].accept(visitor);
    }
}
```



- We can now print the car description for a car object:

```
car.accept(new PrintVisitor());
```

- *Le visiteur n'a pas besoin de savoir comment une Car est structurée en interne*
- *Le visité n'a pas besoin de savoir comment print une description*

Les visiteurs sont un moyen de séparer la structure d'un objet du code fonctionnant sur cet objet

- Objet visité = l'objet contenant les données d'intérêt
- Visiteur = le code qui travaille sur les données

⇒ Chouette Lien : [The Catalog of Design Patterns \(refactoring.guru\)](https://refactoring.guru/catalog/design-patterns/visitor-design-pattern.html)

Module 5

On remarque que la boîte de dialogue apparaît chaque fois au milieu de la page, c'est embêtant, on aimerait changer ça.

Une **Classe Interne** est une classe qui est simplement stockée dans une autre classe. La classe interne peut avoir accès aux méthodes et attributs de la classe englobantes. On peut avoir des classes interne à l'intérieur d'une autre classe ou même d'une méthode.

Plusieurs avantages :

- ⇒ Mieux organisé, une classe interne n'est qu'utilisée dans la classe dans laquelle elle est située
- ⇒ On peut utiliser une classe interne qui a le même nom qu'une autre classe existante

Une **Classe Anonyme** est une classe interne située dans une méthode qui n'a pas de noms

Une **Expression Lambda** permet de simplifier le code d'une interface fonctionnelle (= interface avec une seule méthode)

(Expression sans type) -> corps de la méthode

La partie gauche de l'expression a un ou plusieurs paramètres La partie droite de l'expression contient l'expression à effectuer

Une Expression Lambda peut aussi contenir un résultat → Exemple : $(x,y) \rightarrow x + y$

On peut aussi avoir un bloc de déclarations sur le côté droit, dans ce cas-là on doit utiliser le *return* pour le résultat → Exemple : $(int\ i) \rightarrow \{ int\ j = i*2 ; return\ j+1 \}$

- Les expressions lambda peuvent être utilisées lorsqu'il existe une interface avec une seule méthode abstraite (= interface fonctionnelle)

Le JDK a déjà défini des interfaces fonctionnelles utiles :

1. Interface Fonction $\langle T,R \rangle$ pour les fonctions du type $T \rightarrow R$
2. Interface Bifonction $\langle T,U,R \rangle$ pour les fonctions du type $T \times U \rightarrow R$
3. Interface Prédicat $\langle T \rangle$ pour les fonctions du type $T \rightarrow \text{booléen}$

Il est également possible de combiner deux 2 fonctions :

- Let $f: T \rightarrow R$ and $g: V \rightarrow T$
- We can define a new function $h: V \rightarrow R$ with $h(x) = f(g(x))$
- The Java interface `Function<T,R>` has already a `compose` method that does exactly this:

```
Function<V, R> compose(Function<V,T> g) {
    return (V v) -> apply(g.apply(v));
}
```

▪ **Example:**

```
Function<Double, Double> f = (d) -> d/2.5;
Function<Integer, Double> g = (i) -> Math.sqrt(i);
Function<Integer, Double> h = f.compose(g);
double r = h.apply(25);
```

L'Interface Comparator<T> pour les fonctions $T \times T \rightarrow \text{int}$ retourne un entier

- < 0, si premier argument < second argument
- 0, si premier argument = second argument
- > 0, si premier argument > second argument

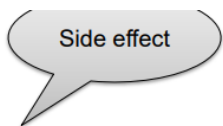
Grâce à l'Interface Comparatoir c'est très simple de changer l'ordre de triage en définissant simplement différemment la fonction comparator

Rappelons-nous que les expressions lambda en Java sont implémentées comme des classes internes anonymes. Elles sont autorisées à accéder aux membres de la classe externe

```
public class SideEffect {
    int sum=0;

    public void run() {
        Function<Integer,Integer> add=(i) -> { sum++; return i+sum; };

        System.out.println(add.apply(3));
        System.out.println(add.apply(3));
    }
}
```



Cependant, ce type de code devrait être interdit. En mathématiques, on attend à ce qu'une fonction retourne toujours la même chose pour le même argument. Une bonne fonction ne doit pas avoir **d'effet secondaire**.

Une fonction ne devrait pas changer les variables et objets existant. Le code d'une fonction ou méthode est plus facile à comprendre si le résultat ne dépend uniquement de ses arguments.

Une structure de données **Immutable** est une structure de données qui ne peut pas être changée après sa création. Les objets String sont immutables en Java par exemple.

```

class Cons {
    public final int value;      // value of the element
    public final Cons next;     // next element, null if end of list

    public Cons(int value, Cons next) {
        this.value = value;
        this.next = next;
    }
}

```

- We can add an element to the head of a list without changing the list:

```

Cons list1 = new Cons(3,null); // this is the list [3]
Cons list2 = new Cons(5,list1); // this is the list [5,3]
Cons list3 = new Cons(1,list2); // this is the list [1,5,3]

```

La méthode *map* permet d'appliquer qqch à tout les éléments d'une liste.

La méthode *filter* va poser une condition et pour chaque élément de la liste, regarder si la condition est true ou non. Si elle est true, alors on l'ajoute dans une nouvelle, sinon non. On obtient ensuite une liste contenant chaque élément pour lequel la condition est vraie. On « filtre » la liste

Appliquer des fonctions à des listes d'objets est si populaire en Java, que Java 8 possède des classes spéciales pour cela : Stream, BasicStream, IntStream,...

Un **Stream** est une séquence d'objets

- ⇒ Comme pour les FList, il y a pleins d'opérations utiles définies pour les streams, comme *map*, *filter*
- ⇒ Les Streams sont un concept de programmation fonctionnelle. En général, les opérations de streams n'ont / ne devraient pas avoir d'effet secondaire.

```

class Account {
    private int value;
    public Account(int value) { this.value=value; }
    public int getValue() { return value; }
}

// Create streams from values
Stream<Integer> stream1=Stream.of(1,2,3,4,5);
Stream<Account> stream2=Stream.of(new Account(100), new Account(200));

// Create a stream from an array.
// For base types (like int and double), Java has optimized stream
// implementations. In general, it is more efficient to use a DoubleStream
// instead of Stream<Double>
double[] a=new double[]{ 1.0, 2.0, 3.0 };
DoubleStream stream3=Arrays.stream(a);

// Create a stream from a List
LinkedList<Integer> list=new LinkedList<>();
list.add(3); list.add(4);
Stream<Integer> stream4=list.stream();

```



```

Stream<Integer> stream1=Stream.of(1,2,3,4,5);

// apply function on each element
Stream<Integer> mappedStream=stream1.map( (i)->i+1 );

// filter elements
Stream<Integer> filteredStream=mappedStream.filter( (i)->i<4 );

// sort stream
Stream<Integer> sortedStream=filteredStream.sorted();

// transform object stream into IntStream
Stream<Account> stream2=Stream.of(new Account(100), new Account(200));
IntStream intStream=stream2.mapToInt( (a)->a.getValue() );

```

Lorsque l'on travaille avec des streams, souvent **la dernière opération est une opération qui retourne certains résultat qui ne sont pas des streams**. Exemples :

- Counting all elements greater than 5:

```
int n=stream.filter( (i)->i>5 ).count();
```

- Transforming the stream into an array:

```
Stream<Account> stream=Stream.of(new Account(100), new Account(200));
int[] values=stream.mapToInt( (a)->a.getValue() ).toArray();
```

- Doing something with each element:

```
Stream<String> stream = Stream.of("Hello", "World");
stream.forEach( (s)-> System.out.println(s) );
```

This is like a map-operation, but it does not have a result.

Souvent, on veut faire une opération avec tout les éléments d'un Stream et calculer une simple valeur de résultat. Par exemple, on veut calculer la somme de tout les éléments dans un stream avec des entiers. Pour cela, on peut utiliser la méthode `.reduce(s,f)`

```

Stream<Integer> stream = Stream.of(1,2,3,4,5);
int result = stream.reduce(0, (a,b)->a+b);

```

La méthode reduce demande 2 arguments :

1. La valeur de départ *s* pour l'opération
2. La fonction *f* qui est appliquée à tout les éléments du stream

On peut simplifier comme cela :

`(s) -> System.out.println(s) ⇔ System.out ::println`

`f = (s,i) -> System.out.format(s,i) ⇔ f = System.out ::format`

`(s) -> s.length() ⇔ String ::length`

La *notation ::* peut aussi être utilisée pour les méthodes statiques :

Instead of

```
Stream<Integer> stream = Stream.of(1,2,3,4,5);
int n = stream.reduce(0, (a,b)->a+b);
```

we can write:

```
Stream<Integer> stream = Stream.of(1,2,3,4,5);
int n = stream.reduce(0, Integer::sum);
```

Les Stream sont paresseux. Les opérations sont exécutées uniquement si on a besoin du résultat, par exemple pour calculer un résultat. Puisque les Streams sont paresseux, ils peuvent être utilisés dans des situations où on ne connaît pas à l'avance la longueur du Stream.

Bon à savoir : On ne peut passer qu'une seule fois par le même courant. Une fois qu'un élément a été traité, il n'est plus possible d'y accéder à nouveau.

▪ This will not work:

```
Stream<Integer> stream = Stream.of(1,2,3,4,5);

int n = stream.reduce(0, (a,b)->a+b);

Stream<Integer> stream2 = stream.map( (i)-> i+1 );
```

Error: "stream has already been operated upon or closed"

Une **Classe Optionnelle** est une classe qui stocke les objets pour nous et teste s'ils sont nuls avant d'effectuer une opération. Si l'objet est nul, rien ne se passe.

```
Optional<String> s=Optional.of(someMethodThatReturnsAString());
s = s.map( String::toUpperCase );

...
// later in the code:

s = s.map( (t)-> t+"/" );
```

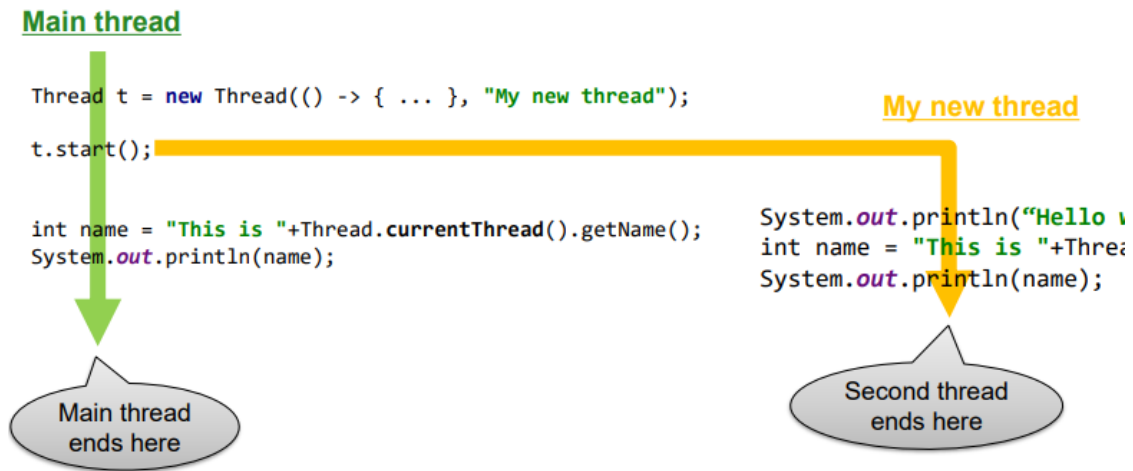
- Of course, at the end you would like to have the object stored in the Optional:

```
String name = s.orElse("empty");
```

```
if(s!=null)
    name=s;
else
    name="empty";
```

Module 6

Un programme que l'on exécute a un thread qui démarre à la méthode `main()`. Mais ce programme peut aussi créer de nouveaux threads en parallèle.



Chaque process a un *Main thread*. C'est ce qui permet l'exécution de notre programme.

Jusqu'à maintenant, on a toujours travaillé avec ce Main thread. On va voir dans ce module comment travailler avec des threads différents de Main

Thread = un fil en anglais → **fil d'exécution du programme**

Un programme se termine lorsque tous ses threads ont fini de s'exécuter

La méthode `.join()` permet de dire d'attendre que le premier thread finisse pour démarrer le second thread.

`.isAlive()` permet de nous dire si un thread a terminé (true) ou non (false)

Les Threads sont intéressants car :

1. S'il on veut faire qqch qui demande beaucoup de temps et on ne veut pas bloquer le reste du programme
2. Accélérer l'exécution d'un programme

⇒ Les Threads n'améliorent la vitesse d'un programme que si les tâches pour les threads sont plus longues que le coût de leur création et de leur gestion.

Un **Thread Pool** est un groupe de threads qui sont prêts à s'exécuter. Il permet de faire attendre les threads avant de s'exécuter

```
ExecutorService executor = Executors.newFixedThreadPool(2);
```

Ici le code crée un thread pool composé de 2 threads.

Future donne le feu vert pour s'exécuter et de lancer leur programme respectifs.

Le mot-clé `.get()` permet d'obtenir le résultat du programme lancé par le thread. Si pas encore fini la méthode va attendre.

Si l'on a pas besoin du thread pool, on peut le supprimer en faisant `executor.shutdown()`

On peut comparer les threads dans les threadpool comme des cuisiniers d'un restaurant qui attendent d'avoir des commandes pour commencer à se mettre au travail.

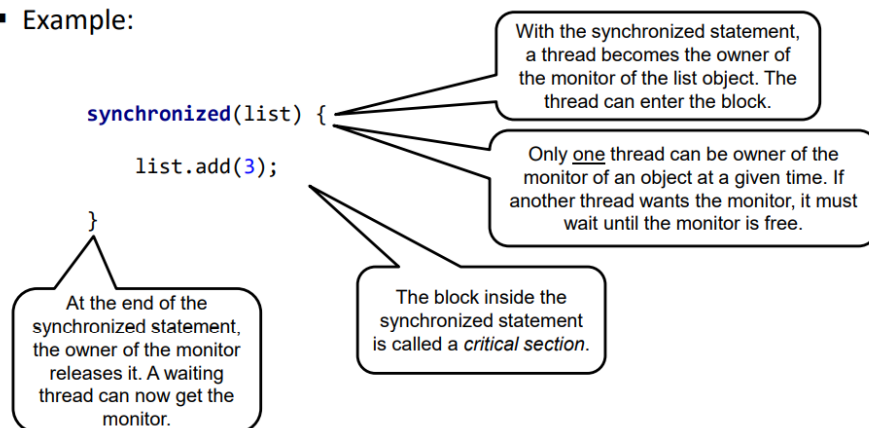
Race Condition (*situation de compétition*) = situation dans laquelle différents threads sont exécutés au même moment. Nous avons aucune garantie pour savoir lequel des différents threads finira son exécution en premier.

⇒ Lorsque 2 threads tournent en parallèle (en même temps), il est possible que l'exécution soit entrelacée : mélange des lignes de code des 2 threads entre eux

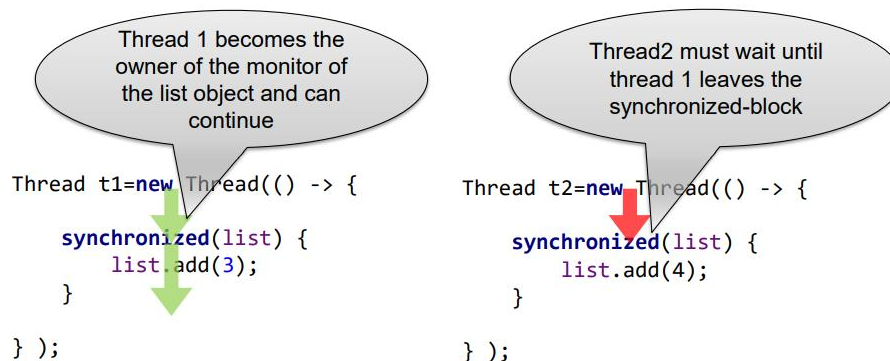
Notre objectif est alors d'empêcher 2 threads d'accéder à une variable en même temps.

En Java, chaque objet a un moniteur. Un **moniteur** permet d'empêcher d'autres threads à accéder à une partie du code.

▪ Example:



La Critical Section : partie de notre code que l'on veut protéger



Note: In this example, we have assumed that thread 1 first enters the critical section. It can also happen that thread 2 enters first. Then thread 1 would have to wait.

Les threads peuvent utiliser le moniteur de n'importe quel objet pour la synchronisation. Il peut même s'agir d'un objet spécifiquement créé dans ce but. Bien sûr, les deux threads doivent utiliser le même objet pour se synchroniser

Il est également possible en Java de marquer la méthode entière comme *synchronized*

Avec un `synchronized` devant la méthode, la **méthode** entière est **synchronized**. Ce qui veut dire que la méthode est protégée par un verrou, ce qui signifie qu'une seule thread peut y accéder à la fois.

Attention, la plupart des structures de données en `java.util.*` ne sont pas *thread-safe* :

Des races conditions peuvent se produire !

On doit protéger les méthodes que l'on utilise.

Cependant il existe des méthodes intégrées pour nous protéger de cela (`synchronizedList`,...)

La méthode `.wait()` permet de « mettre en pause » un bout de code, donc consommer moins de mémoire et être plus rapide

La méthode `.notify()` permet de le « réveiller » justement.

On doit donc checker si l'objet qui est en `wait()` est maintenant réveillé, pour que l'on puisse exécuter ce que l'on veut et passer à la suite du programme.

- A thread can only call `wait()` or `notify()` on an object if it owns the monitor of the object, i.e. `wait()` and `notify()` must be always inside a `synchronized` block

```
// T1:
synchronized(someObject) {
    someObject.wait();
    i++;
}

// T2:
synchronized(someObject) {
    someObject.notify();
}
```

- When a thread calls `wait()`, the thread releases the monitor
- When a waiting thread is waken up, it waits until the thread that called `notify()` releases the monitor
- Example: Let's assume T1 first gets the monitor of `someObject`
 1. T1 calls `wait()` and releases the monitor of `someObject`
 2. T2 can enter the monitor and calls `notify()`
 3. T1 has to wait until T2 leaves the monitor
 4. When T2 has left, T1 can execute `i++`