JAVA

Table des matières

I- POO (Programmation Orientée Objet)

Paradigme de programmation informatique consistant en la définition et l'interaction de briques logicielles appelées objets

Objet : Concept, idée ou entités du monde physique. Possède une structure interne (Attributs) et un comportement (méthodes), il interagit avec ses pairs.

1- Les 3 Principes du POO

- **Encapsulation** : protéger (ne pas la modifier n'importe comment) l'information (attributs et méthodes) contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet (**accesseur** : getters & setters) :
  - **public** : accessible partout
  - **« »** : accessible uniquement depuis toutes les classes du package
  - **protected** : accessible uniquement depuis les classes dérivées et classes du même package
  - **private** : accessible depuis la classe seulement

- **Héritage** (extends) **:** permet la réutilisabilité des composants et l'adaptabilité des objets (polymorphisme). Une **classe fille** (**dérivée**) peut hériter d'une classe les attributs et méthodes **protected** ou **public**. Si cette dernière est **abstraite**, la classe dérivée doit **définir** les caractéristiques abstraites

- **Polymorphisme :** permettre à un même code d'être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales (permet de manipuler les objets sans connaître leurs types) :
  - **Overrinding** : **Redéfinition** des méthodes (**polymorphes**) avec le même nom et le même type et nombre de paramètres dans les classes filles (**Héritage** – Sous typage)
    - **Varagrs** (Nombre inconnu de paramètres) : method(T...) ⇒ T[]
  - **Overloading (Paramétré, Surcharge) :** permet de définir des méthodes qui peuvent être appliquées sur plusieurs types (**nombres** ou **types** de paramètres **différents**)
  - **Généricité** : Abstraction du typage (**class<T>)**
  - **Covariance** : modifier le **type de retour** (obligatoirement un sous-type du type original) d'une méthode que l'on redéfinit

2- Interface vs Abstract Class

- **Classe abstraite** : une classe non ~~instanciable~~ dont certaines méthodes (abstraites) n'ont pas été ~~implémentées~~ (doivent être implémentées si une classe en hérite), elle sert à **factoriser le code**
  Elle possède un **constructeur**

- **Interface** : on n'y trouve que les **prototypes** de méthodes (déclaration), une classe qui l'implémente doit implémenter ses derniers. Permet de définir des **services** visibles depuis l'extérieur.
  On ne doit pas y mettre des attributs (sinon ils sont final static)
  Ne possède pas de ~~**constructeur**~~

3- static vs final vs volatile vs transient

- **static** :
  - **Variable** ou **Méthode** : appartient à la classe et non à son instance (variable ou méthode de classe)
  - **Bloc de code** : Le mot-clé static devant un bloc de code indique que celui-ci ne sera exécuté qu'une fois. L'exécution se fait lors du chargement de la classe par le ClassLoader. On peut utiliser ces blocs, par exemple, pour initialiser des variables statiques complexes.
- **final** :
  - **Méthode** : ne peut plus être ~~redéfinie~~ (~~override~~) dans une classe dérivée
  - **Classe** : ne peut plus être ~~héritée~~
  - **Variable** : ne peut plus être ~~modifiée~~ après sa 1ère initialisation obligatoire
  - **Paramètre** d'une méthode : empêche l'attribution d'une ~~nouvelle valeur~~ au paramètre
- **Constante** : final static
- **transient :** il permet d'interdire la ~~sérialisation~~ de certaines variables d'une classe

4- Classe interne

Une classe interne n'est ni plus ni moins qu'une classe définie dans une autre classe. Il existe différents types de classes internes.

A- Classe interne simple (inner class)

C'est une classe définie dans une autre.

Intérêt : la classe interne peut avoir accès aux méthodes et attributs de la classe englobante.

```
public class SecondOuterClass {
    private String classAlias = "outer alias";
    public class InnerClass {
        String classAlias = "inner alias";

        public void printAlias() {
            System.out.println("Outer: " +
                               SecondOuterClass.this.classAlias);
            System.out.println("Inner: " + this.classAlias);
        }
    }

    public static void main(String... args) {
        InnerClass innerClass = new SecondOuterClass().new InnerClass();
        innerClass.printAlias();     // will print   Outer: outer alias
                                     //              Inner: inner alias
    }
}
```

Une classe interne, comme pour les attributs et méthodes, a un accès **public**, **protected**, **private**, « », et peut être aussi **static (static nested class).**

B- Classe interne anonyme (anonymous inner class)

Une classe anonyme est une classe sans ~~nom~~, qui est définie par **dérivation** d'une **super classe** ou par implémentation d'une **interface**.

Dérivation d'une super classe	Implémentation d'interface

`SuperClasse c = new SuperClasse(){`
`  // méthodes redéfinies`
`  @Override`
`};`
	`Interface c = new Interface(){`
	`  // Implementation des`
	`  // méthodes de Interface`
	`};`

L'interface **Runnable** en est un exemple courant.

C- Classe locale (local inner class)

C'est une classe interne définie dans un **bloc** et dont la portée est limitée à ce dernier. Elle ne peut pas être ~~**static**~~**.**
PS : Une classe locale peut accéder aux **attributs** de la **classe englobante** ainsi qu'aux **paramètres** et **variable locales** de la méthode où elle est définie, à condition que ceux-ci soit spécifiés ***final***.

5- JVM vs JRE vs JDK

- **JVM** (**J**ava **V**irtual **M**achine) : interpréter et exécuter le byte code
- **JRE** (**J**ava **R**untime **E**nvironment) : JVM + class librairies
- **JDK** (**J**ava **D**evelopement **K**it) : JRE + development tool (javac.exe : compiler, java.exe : launcher, debugger ...)

6- Classe Immuable

C'est une classe dont l'état ne peut pas changer une fois créé.

A- Avantages

- Ils sont garantis **thread-safe**
- Ils peuvent être mis en **cache**
- Ils n'ont besoin ni de ~~**constructeur par copie**~~ (constructeur avec l'objet en paramètre, qui appelle un constructeur avec les valeurs de ce dernier), ni d'implémentation de l'interface ~~**Cloneable**~~ (copie par valeur, implémentation de clone)
- Il n'est pas nécessaire d'en faire une ~~**copie défensive**~~
- Leurs invariants sont **testés** à la **création** seulement
- Ils constituent d'excellentes **clés** pour les **Map** et éléments des **Set** (leur état ne doit pas changer dans la collection)
- **Lazy initilization** et **cache** du **HashCode**
- **Failure atomoticity** (si un objet immuable renvoie une exception, il n'est jamais laissé dans un état indésirable ou indéterminé)

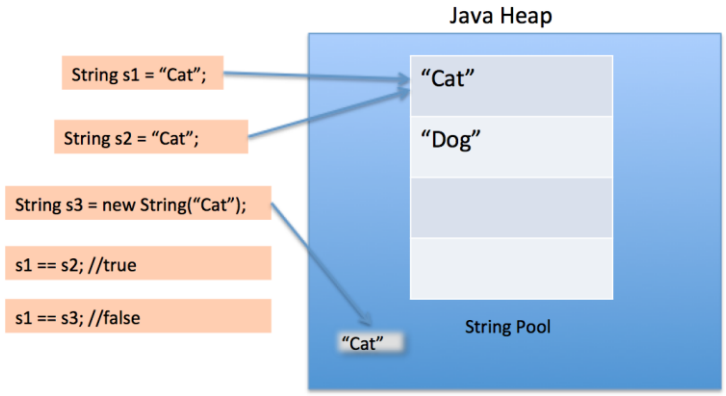
B- Principes

- La **classe** doit être **final**
- Tous les champs doivent être déclaré **private** et **final**
- Ne pas implémenter de méthodes ~~**setter**~~
- La **référence** à **this** ne doit jamais être ~~**exportée**~~

- Retourner des objets copiés en profondeur (**deep cloned objects**) pour tous les champs **muables**

C- String

- **Sécurité** du **JDK**: Les classes java sont chargées avec les noms de classes en paramètres. Si ces derniers n'étaient pas immuables, on pourrait injecter son propre mécanisme de chargement des classes, et hacker rapidement n'importe quelle application
- **Performance** : String Pool



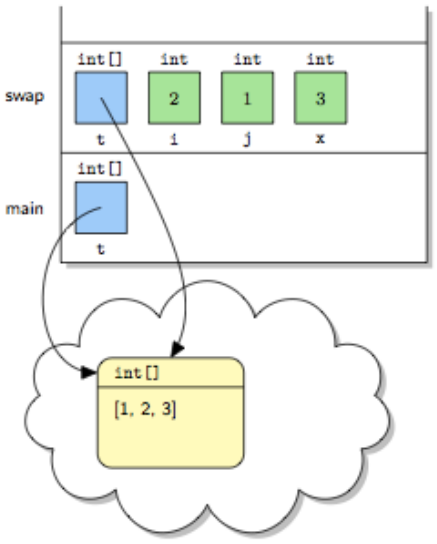
Pour **ajouter** un string créé avec **new** dans le **String Pool**, il faut appeler **String.intern()** : ajoute et retourne le string depuis le String Pool.

- **Thread Safety**

a- String vs StringBuffer vs StringBuilder

- **String** est **~~immuable~~** alors que **StringBuffer** et **StringBuilder** sont **muable**
- **StringBuffer** est **synchronisé** alors que **StringBuilder** est non **~~synchronisé~~**
  ⇒**StringBuilder** est **plus rapide** que **StringBuffer**
- La concaténation des string avec « **+** » est faite avec **StringBuffer**

7- Mémoire JAVA



Pile : Contient les différents environnements (variables) des méthodes

Tas : Contient toutes les **instances** des **objets** crées.

Chaque **Thread** possède sa **propre pile** mais partage le **même tas**

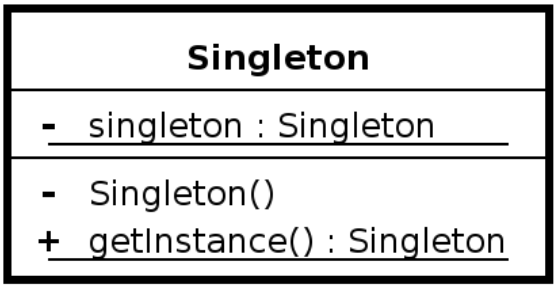
II- Design Pattern

Solutions standard pour des problèmes récurrents d'architecture et de conceptions

1- Création :

A- Singleton

Pour **restreindre** l'**instanciation** d'une classe à un **seul**. Utilisé pour son **efficacité**, sa **rapidité** et son **faible** coût de **mémoire**.



Early Initialization :

```
Public final class Singleton
{
    private static Singleton instance = new Singleton();

    private Singleton(){}

    public static Singleton getInstance()
    {
        return instance;
    }
}
```

Lazy Initialization :

```
public final class Singleton {
     private static volatile Singleton instance = null;

     private Singleton() {}

     public static Singleton getInstance() {
          // Double-Checked Locking
          if (instance == null) {
               synchronized (Singleton.class) {
                    if (instance == null) {
                         instance = new Singleton();
                    }
               }
          }
          return Singleton.instance;
     }
```

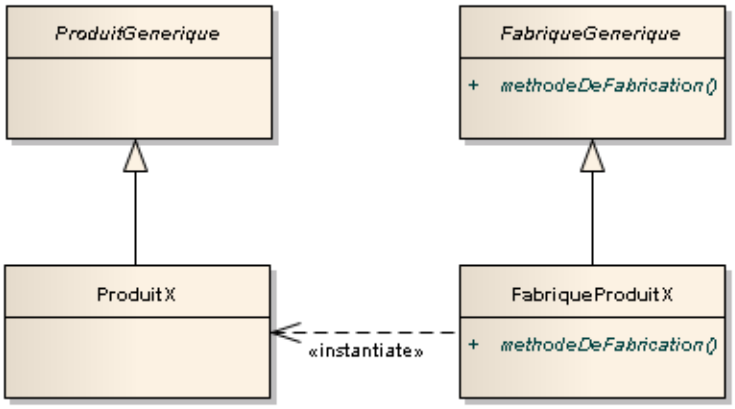
```
}
```

Le ~~**Double-Checked Locking**~~ ne marche pas dans le cas d'**optimisation de byte-code** ou bien dans une architecture **multi-processeurs** à **mémoire partagée** (ex : changement de l'**ordre d'exécution/initialisation** non ~~**atomique**~~).

B- Factory

Définit une **interface** pour la **création** d'un **objet** en **déléguant** à ses sous-classes le **choix** des **classes** à **instancier**.

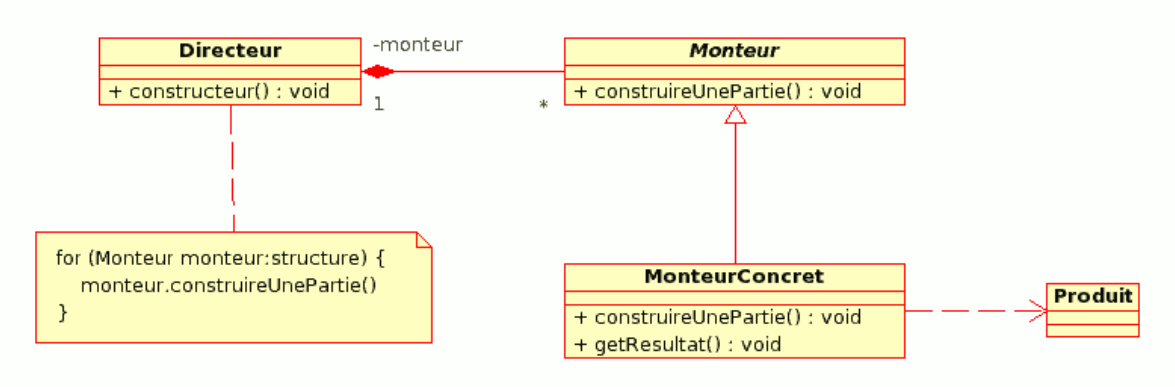
Plusieurs fabriques peuvent être **regroupées** en une **fabrique abstraite** (Interface) permettant d'instancier des objets dérivant de **plusieurs types abstraits différents**




```
public class FabriqueAnimal {
     Animal getAnimal(String typeAnimal) throws ExceptionCreation {
          Animal animal;
          if ("chat".equals(typeAnimal)) {
               animal = new Chat();
          } else if ("chien".equals(typeAnimal)) {
               animal = new Chien();
          } else {
               throw new ExceptionCreation("Impossible de
                    créer un " + typeAnimal);
          }
          return animal;
     }
}
```

C- Builder

Le monteur (builder) est un patron de conception utilisé pour la **création** d'une **variété** d'**objets complexes** à partir d'un **objet source**. L'objet source peut consister en une variété de **parties** contribuant individuellement à la création de chaque objet complet grâce à un **ensemble d'appels** à l'interface commune de la classe abstraite Monteur.



- **Monteur (Builder)** : classe abstraite qui crée les objets (produit)
- **MonteurConcret** :
  - fournit une implémentation de Monteur
  - construit et assemble les différentes parties des produits
- **Directeur** : construit un objet utilisant la méthode de conception Monteur
- **Produit** : l'objet complexe en cours de construction

```
/* Produit */
class Pizza {
	private String pate = "";
	private String sauce = "";
	private String garniture = "";

	public void setPate(String pate) {
		this.pate = pate;
	}

	public void setSauce(String sauce) {
		this.sauce = sauce;
	}

	public void setGarniture(String garniture) {
		this.garniture = garniture;
	}
}

/* Monteur */
abstract class MonteurPizza {
	protected Pizza pizza;

	public Pizza getPizza() {
		return pizza;
	}

	public void creerNouvellePizza() {
		pizza = new Pizza();
	}
```

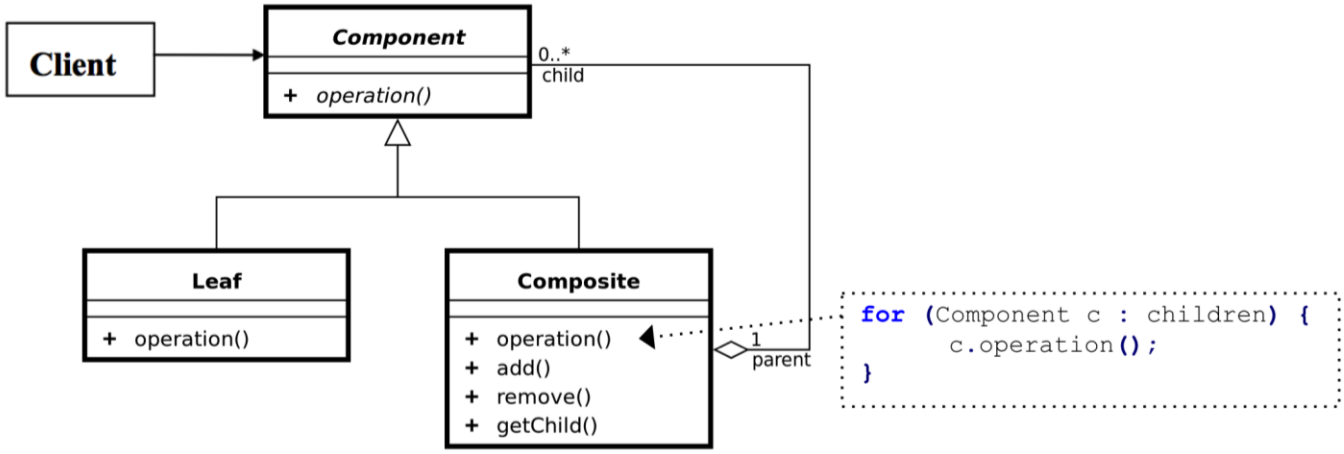
```
	public abstract void monterPate();

	public abstract void monterSauce();

	public abstract void monterGarniture();
}

/* MonteurConcret */
class MonteurPizzaHawaii extends MonteurPizza {
	public void monterPate() {
		pizza.setPate("croisée");
	}

	public void monterSauce() {
		pizza.setSauce("douce");
	}

	public void monterGarniture() {
		pizza.setGarniture("jambon+ananas");
	}
}

/* MonteurConcret */
class MonteurPizzaPiquante extends MonteurPizza {
	public void monterPate() {
		pizza.setPate("feuilletée");
	}

	public void monterSauce() {
		pizza.setSauce("piquante");
	}

	public void monterGarniture() {
		pizza.setGarniture("pepperoni+salami");
	}
}

/* Directeur */
class Serveur {
	private MonteurPizza monteurPizza;

	public void setMonteurPizza(MonteurPizza mp) {
		monteurPizza = mp;
	}

	public Pizza getPizza() {
		return monteurPizza.getPizza();
	}

	public void construirePizza() {
		monteurPizza.creerNouvellePizza();
		monteurPizza.monterPate();
		monteurPizza.monterSauce();
		monteurPizza.monterGarniture();
	}
}

/* Un client commandant une pizza. */
class ExempleMonteur {
	public static void main(String[] args) {
```

```
            Serveur serveur = new Serveur();
            MonteurPizza monteurPizzaHawaii = new MonteurPizzaHawaii();
            MonteurPizza monteurPizzaPiquante = new MonteurPizzaPiquante();

            serveur.setMonteurPizza(monteurPizzaHawaii);
            serveur.construirePizza();

            Pizza pizza = serveur.getPizza();
      }
}
```

2- Structure

A- Composite

Un objet **composite** est constitué d'un ou de **plusieurs objets <u>similaires</u>** (ayant des fonctionnalités similaires). L'idée est de **manipuler** un **groupe** d'**objets** de la même façon que s'il s'agissait d'un **seul objet**. Les objets ainsi regroupés doivent posséder des **opérations communes**, c'est-à-dire un "***<u>dénominateur commun</u>***"



Component (Composant)

- est l'abstraction pour tous les composants, y compris ceux qui sont composés
- déclare l'**interface** pour le **comportement** <u>par défaut</u>

Leaf (Feuille)

- représente un **composant** n'ayant pas de ~~sous-éléments~~
- **implémente** le **comportement** <u>par défaut</u>

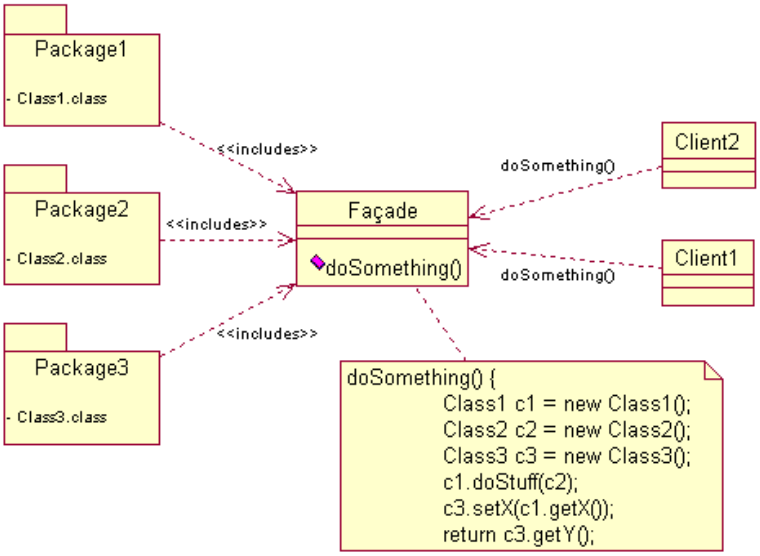
Composite

- représente un **composant** pouvant avoir des **sous-éléments**
- stocke des **composants enfants** et permet d'y accéder
- implémente un **comportement** en utilisant les **enfants**

Client

- manipule les objets de la composition à travers l'interface la classe Composant

B- Façace



A pour but de **cacher** une **conception** et une **interface complexe** difficile à comprendre en fournissant une **interface simple** du sous-système.

Une façade peut être utilisée pour :

- rendre une **bibliothèque** plus facile à utiliser, comprendre et tester;
- rendre une bibliothèque plus lisible;
- **réduire** les **dépendances** entre les clients de la bibliothèque et le fonctionnement interne de celle-ci, ainsi on gagne en **flexibilité** pour les **évolutions futures** du système;
- **assainir** une **API** que l'on ne peut pas ~~modifier~~ si celle-ci est mal conçue, ou mieux **découper** ses **fonctionnalités** si celle-ci n'est pas assez claire.

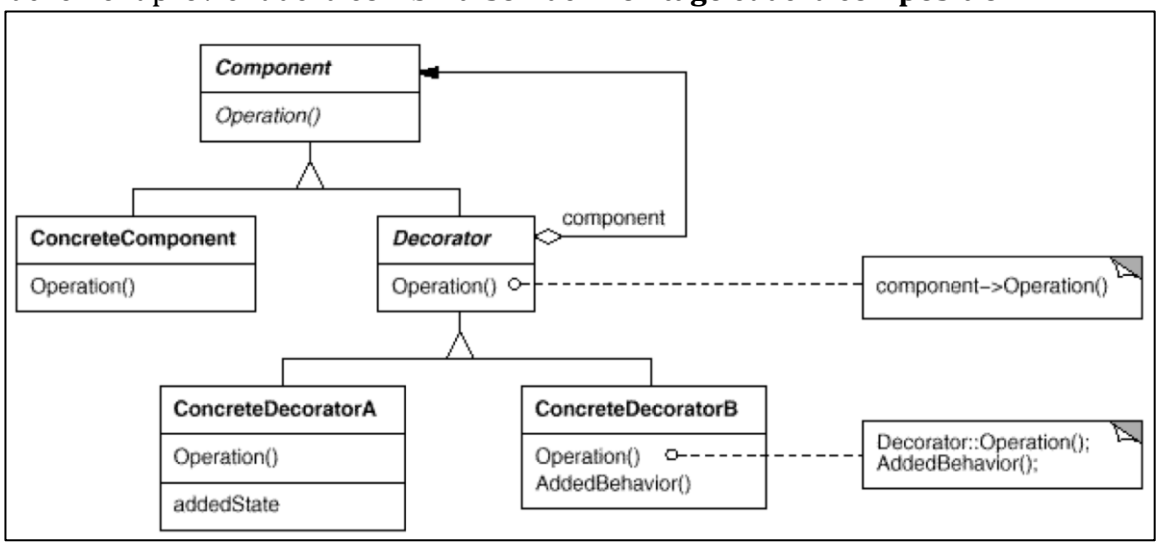
C- Decorator

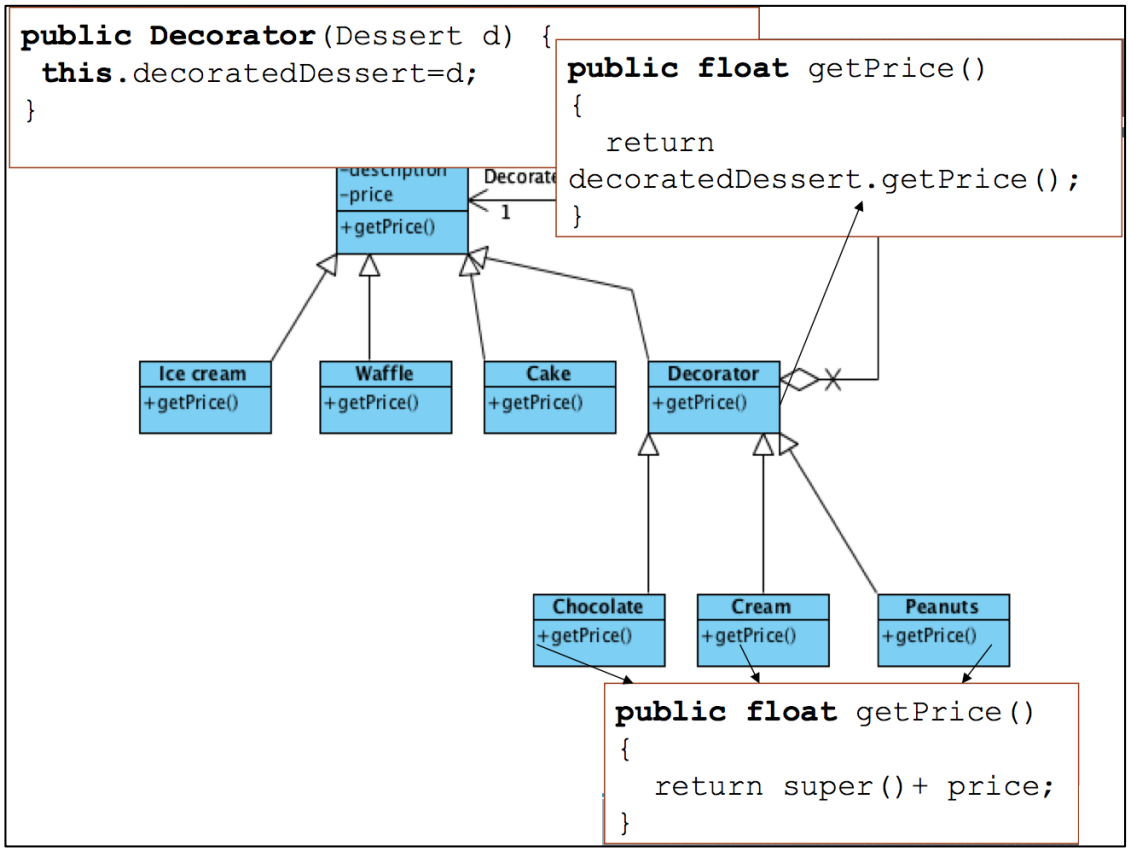
Un décorateur permet d'ajouter **dynamiquement** de **nouvelles <u>fonctionnalités</u>** à un objet. Les décorateurs offrent une alternative assez souple à l'héritage pour composer de nouvelles fonctionnalités.

Problème de l'héritage :

- Modèle **complexe** avec un grand nombre de classe
- ajout de fonctionnalités de façon **statique**

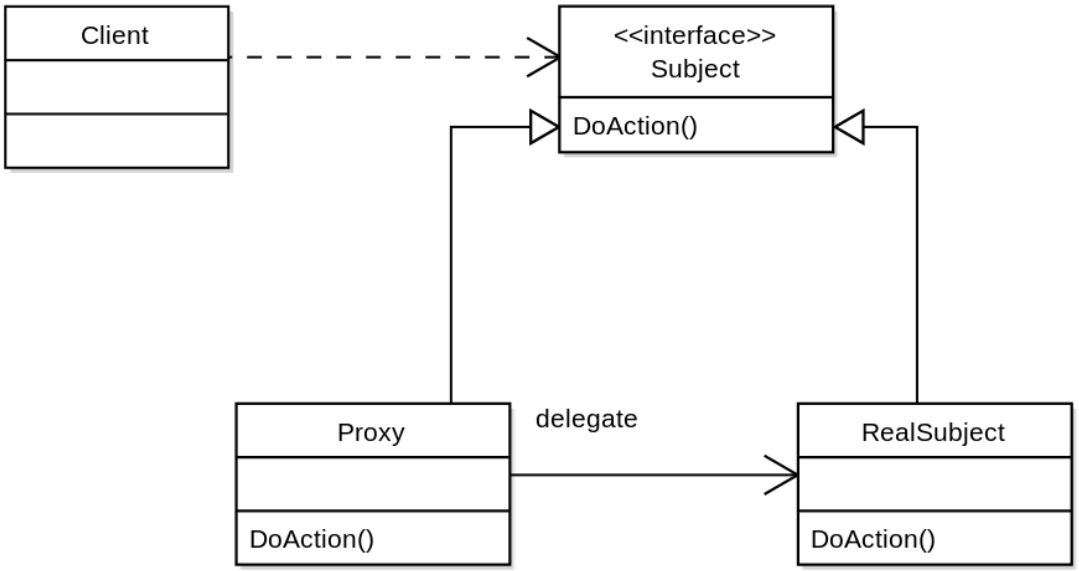
La puissance de ce pattern qui permet d'ajouter (ou modifier) des fonctionnalités facilement provient de la **combinaison** de l'**héritage** et de la **composition**





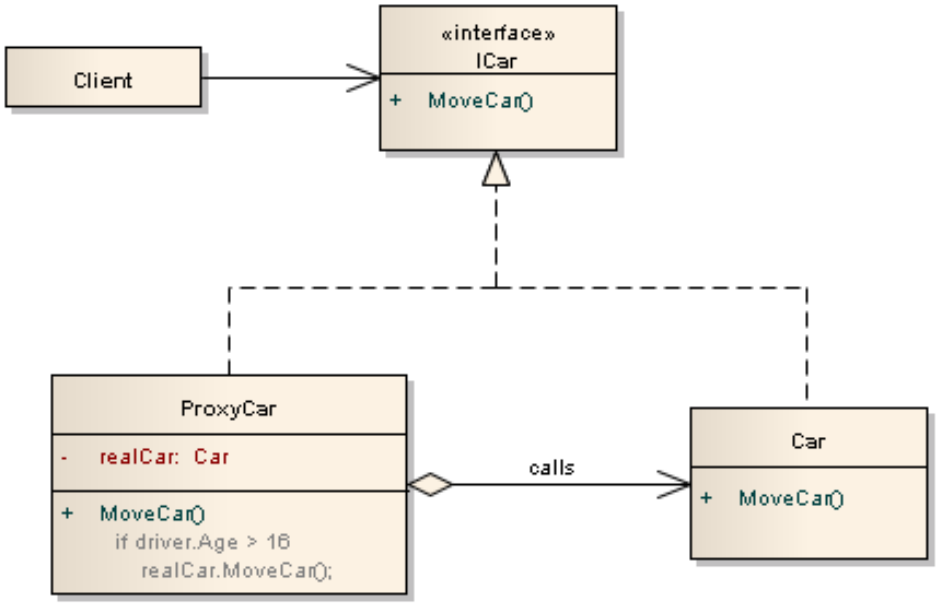
D- Proxy

Un proxy est une **classe** se **substituant** à une **autre classe**. Par convention et simplicité, le proxy **implémente** la même **interface** que la classe à laquelle il se substitue. L'utilisation de ce proxy ajoute une **indirection** à l'utilisation de la **classe** à **substituer**.



Exemple :

Nous avons des membres de la famille et les voitures, et ce sont les classes existantes que nous ne pouvons pas changer. Nous voulons préciser en outre que seules les membres de la famille de **plus de 16 ans** peuvent **conduire** des **voitures**, et nous allons ajouter cette logique dans la classe **proxy**.

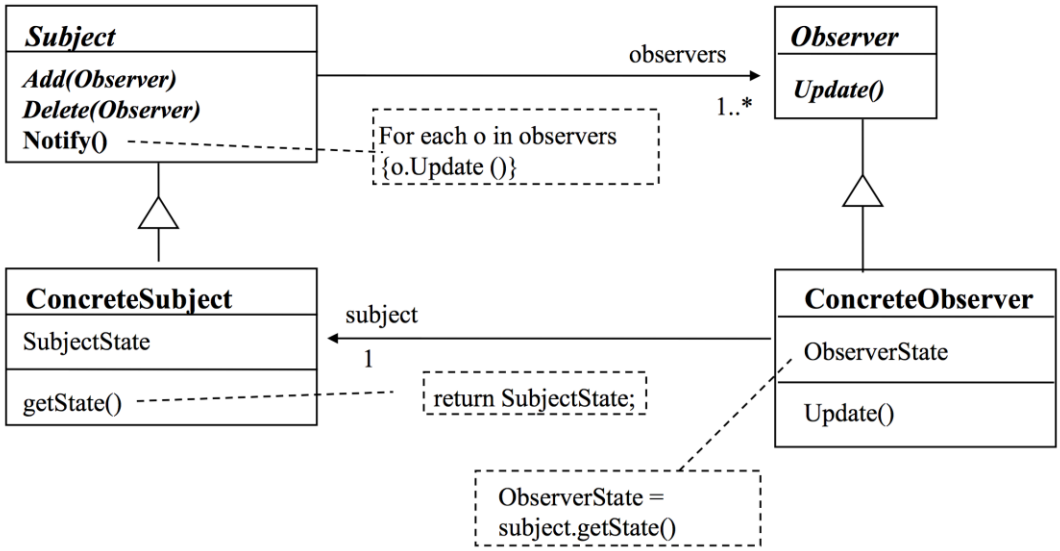


3- Comportement

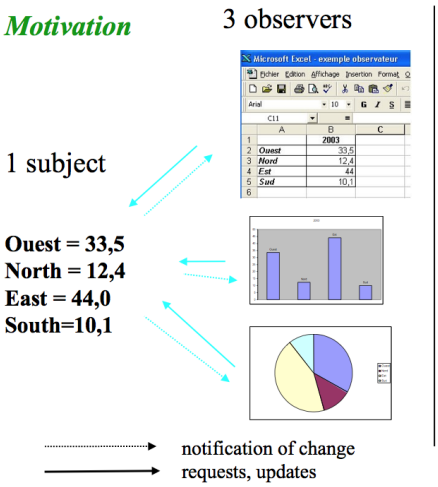
A- Observer

Le pattern Observateur (en anglais Observer) définit une **relation** entre **objets** de type **un-à-plusieurs**, de façon que, si un **objet** change d'état, tous ceux qui en **dépendent** en soient **informés** et **MAJ** automatiquement.

Exemple : les interfaces graphiques



- ***Subject*** : Connais ses **observateurs**, il offre une **interface** pour **ajouter** ou **supprimer** des observateurs
- ***Observer*** : défini une **interface** pour la **MAJ** des **objets** qui doivent être **notifier** des **MAJ** d'un **sujet** (Subject)
- ***ConcreteSubject*** : **Mémorise** les états (**state**) qui sont intéressant pour le **ConcreteObserver**. Envoie une **notification** aux **observateurs** quand ils **changent** d'états
- ***ConcreteObserver*** : gère une **référence** sur un **ConcreteSubject**; **mémorise** l'**état** qui doit rester pertinent pour le sujet; **implémente l'interface** de **MAJ** de l'o**bservateur** afin de mémoriser la cohérence entre l'état et le sujet.



When to use : use this pattern :

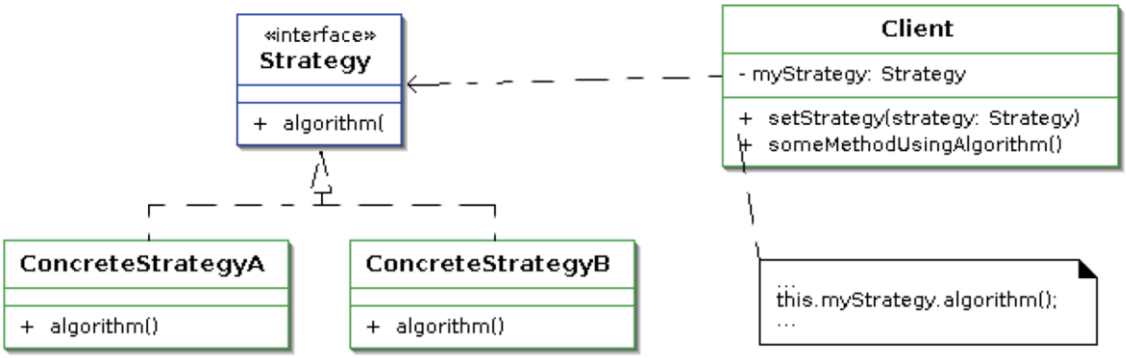
. if a concept has got 2 (or more) representations.

. when the modification of an object implies the modification of other objects whose number is not known.

...

B- Strategy

C'est patron de conception (design pattern) de type **comportemental** grâce auquel des **algorithmes** peuvent être **sélectionnés** à la **volée** au cours de l'**exécution** selon certaines conditions



Eléments caractéristiques :

- **myStrategy** : un attribut représentant la stratégie qui est d'un type abstrait
- l'invocation des méthodes de la stratégie, en faisant **varier** le **type concret** de la **stratégie** on **change** le **comportement** apparent de la classe cliente.

On **remplace** la variation du comportement que l'on peut obtenir par **héritage** par une **variation** du **comportement** par **composition** via la **stratégie**.

Exemple : Dans les api awt/swing, un objet **Container** gère la disposition des différents objets graphiques qu'il contient grâce à un **LayoutManager**

- public interface **LayoutManager** : Defines the interface for classes that know how to lay out Containers.
  Et les classes : java.awt.FlowLayout, java.awt.GridLayout, java.awt.BorderLayout, ...

- Dans **java.awt.Container** : public void **setLayout**(LayoutManager mgr) Sets the layout manager for this container

C- Iterator

Un itérateur est un **objet** qui permet de **parcourir** tous les **éléments contenus** dans un **autre objet**, le plus souvent un **conteneur**, tout en **isolant** l'utilisateur de la **structure interne** du conteneur, potentiellement **complexe**.

Un itérateur ressemble à un **pointeur** disposant essentiellement de deux primitives :

1- **accéder** à l'élément **pointé** en **cours** (dans le conteneur)
2- et se **déplacer** pour **pointer** vers l'**élément suivant**.

En sus, il faut pouvoir créer un **itérateur** pointant sur le **premier** élément; ainsi que **déterminer** à tout moment si l'itérateur a épuisé la **totalité** des éléments du conteneur (**hasNext**).

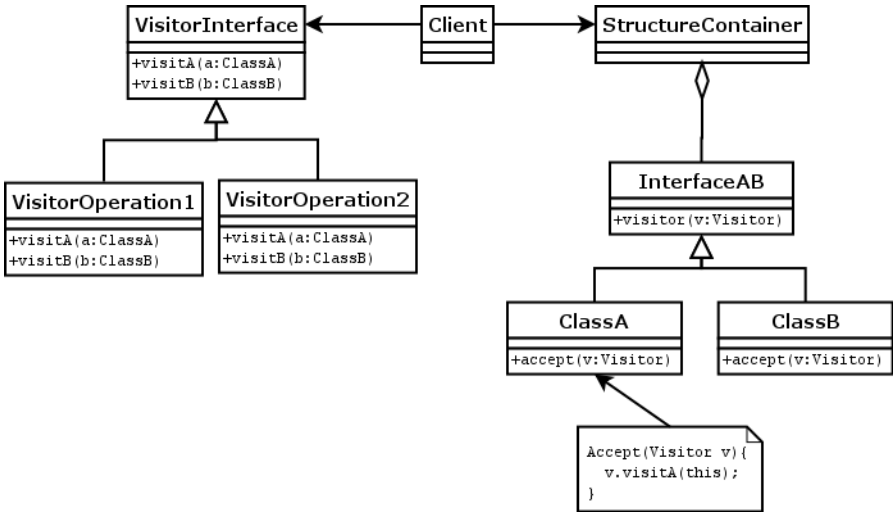
```
public class TestList<T> implements Iterable<T> {
    private T[] arrayList;
    private int currentSize;
    public TestList(T[] newArray) {
        this.arrayList = newArray;
        this.currentSize = arrayList.length;
    }
    // Implement the Iterator Method
    public Iterator<T> iterator() {
            Iterator<T> iterator = new Iterator<T>() {
                  private int currentIndex = 0;
                  public boolean hasNext() {
                        return currentIndex < currentSize &&
                             arrayList[currentIndex] != null;
                  }
                  public T next() {
                        return arrayList[currentIndex++];
                  }
                  public void remove() {
                        // TODO Auto-generated method stub
                  }
            };

            return iterator;
      }

    public static void main(String[] args) {
        // create an array of type Integer
        Integer[] numbers = new Integer[]{1, 2, 3, 4, 5};
        // create your list and hold the values.
        TestList<Integer> testList = new TestList<Integer>(numbers);
        // Iterator
        Iterator<Integer> iterator = testList.iterator();
        while (iterator.hasNext()) {
            Integer value = iterator.next();
            if (iterator.hasNext()) {
                System.out.print(value + ", ");
            } else {
                System.out.print(value);
            }
        }
      }
}
```

ListIterator pour **hasPrevious** et **Previous**

D- Visitor

Le pattern Visiteur Permet de :

- **séparer** un **algorithme** d'une **structure** de **données**.
- **externaliser** des **actions** à effectuer sur des **objets**. Ces actions ne seront pas implémentées directement dans la classe de ces objets, mais dans des classes externes.
- ajouter des **actions** sur un **groupe** d'**objets** (qui ne sont pas forcément du ~~même type~~) tout en **externalisant** le code



Pratique :

- chaque classe pouvant être « **visitée** » doit mettre à disposition une méthode publique « **accepter** » prenant comme argument un objet du type « **visiteur** ».
- La méthode « **accepter** » appellera la méthode « **visite** » de l'objet du type « **visiteur** » avec pour argument l'**objet visité**.

→De cette manière, un **objet visiteur** pourra connaître la **référence** de l'**objet visité** et **appeler** ses **méthodes** publiques pour obtenir les données nécessaires au traitement à effectuer (calcul, génération de rapport, etc.).

```
interface Visitor {
	void visit(ObjetPere objet);
}
interface ObjetPere {
	void accept(Visitor v);
}
public class Objet1 implements ObjetPere {
	public void accept(Visitor v) {
		v.visit(this);
	}
}
public class Objet2 implements ObjetPere {
	public void accept(Visitor v) {
		v.visit(this);
	}
}
public class Objet3 implements ObjetPere {
	public void accept(Visitor v) {
		v.visit(this);
	}
}
```

III- Algorithme de tri

1- Tri à bulles (tri par propagation)

L'algorithme parcourt le tableau, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet du tableau, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est trié. On arrête alors l'algorithme.

```
public static void tri_bulle(int T[]) {
	boolean swaped;
	do {
		swaped = false;
		for (int i = 0; i < T.length - 1; i++) {
			if (T[i] > T[i + 1]) {
				// echanger T[i] et T[i+1]
				int temp = T[i];
				T[i] = T[i + 1];
				T[i + 1] = temp;
				swaped = true;
			}
		}
	} while (swaped);
}
```

Complexité

- **Pire des cas ($O(n^2)$) :** le plus petit élément est à la fin du tableau
- **Moyenne ($O(n^2)$)** : En effet, le nombre d'échanges de paires d'éléments successifs est égal au nombre d'inversions, c'est-à-dire de couples (i,j) tels que i < j et T(i) > T(j). Ce nombre est indépendant de la manière d'organiser les échanges.
- **Meilleur des cas** (**Linéraire (n)**) : une seule itération quand le tableau est déjà trié

2- Tri par insertion

Tri classique utilisé naturellement pour trier les **cartes** en **insérant** chaque carte à sa **place**.

Le tri par insertion est cependant considéré comme le tri le plus efficace sur des entrées de petite taille. Il est aussi très rapide lorsque les données sont déjà presque triées

Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le i-ème élément, les éléments qui le précèdent sont déjà triés. Pour faire l'analogie avec l'exemple du jeu de cartes, lorsqu'on est à la i-ème étape du parcours, le i-ème élément est la carte saisie, les éléments précédents sont la main triée et les éléments suivants correspondent aux cartes encore mélangées sur la table.

L'objectif d'une étape est d'insérer le i-ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont fréquemment effectuées en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

```
public static void triInsertion(int T[]) {
      int elementToInsert,j;
      for (int i = 1; i < T.length; i++) {
            elementToInsert = T[i];
            j = i;
            while (j > 0 && T[j - 1] > elementToInsert) {
                  T[j] = T[j - 1];
                  j = j - 1;
            }
            T[j] = elementToInsert;
      }
}
```

Complexité :

- **Pire des cas ($O(n^2)$)** : atteint lorsque le tableau est **trié** à **l'envers**, l'algorithme effectue de l'ordre de $\mathbf{n^2/2}$ affectations et comparaisons
- **Moyenne ($O(n^2)$)** : Si les éléments sont distincts et que toutes leurs permutations sont équiprobables, alors en moyenne, l'algorithme effectue de l'ordre de $\mathbf{n^2/4}$ affectations et comparaisons
- **Meilleur des cas ($O(n)$)** : Si le tableau est déjà trié, il y a n-1 comparaisons et O(n) affectations.

3- Tri par sélection (extraction)

Le tri par sélection (ou tri par **extraction**) est un algorithme de tri par **comparaison**. Il est particulièrement **simple**, mais **inefficace** sur de **grandes entrées**, car il s'exécute en temps quadratique en le nombre d'éléments à trier.

```
public static void triSelection(int T[]) {
      for (int i = 0; i < T.length - 1; i++) {
            int min = i;
            // Chercher le plus petit element
            for (int j = i + 1; j < T.length; j++)
                  if (T[j] < T[min])
                        min = j;
            if (min != i) {
                  // echanger T[i] et T[min]
                  int temp = T[i];
                  T[i] = T[min];
                  T[min] = temp;
            }
      }
}
```

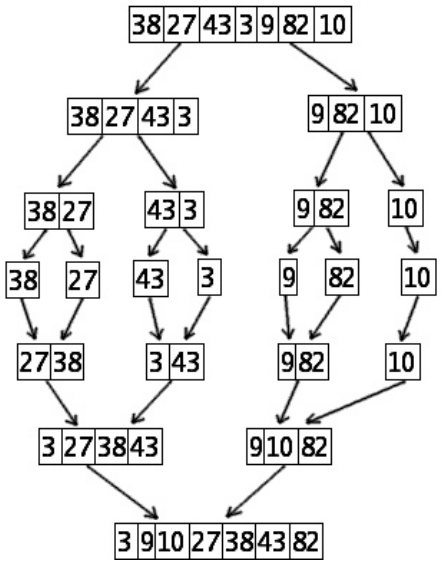
Principe : rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice i - 1 et ainsi de suite.

Complexité :

- **Pire des cas $O(n^2)$**: atteint par exemple lorsqu'on trie la séquence 2,3,…,n,1.
  n-1 échanges

4- Tri par fusion

Le tri fusion est un algorithme de tri par **comparaison stable**. Sa complexité est de **O(n log n)**, ce qui est asymptotiquement optimal. Ce tri est basé sur la technique algorithmique diviser pour régner



Principe :

L'algorithme est naturellement décrit récursivement. Deux points de vue sont possibles. Voici le premier, particulièrement efficace pour **trier** les **listes** :

1. On **trie** les éléments **deux** à **deux**, ce qui donne de petites **listes triées** de **2 éléments** chacune
2. On **fusionne** ces listes deux à deux, ce qui donne de nouvelles listes triées et deux fois plus grandes
3. On **réitère** l'opération précédente jusqu'à obtenir une **seule liste**

Le second point de vue est tout aussi efficace pour le tri des tableaux, mais déconseillé pour les listes :

1. On **coupe** en **deux parties** à peu près égales les données à trier
2. On **trie** les données de **chaque partie** (pour cela, on coupe chaque partie en deux et on trie chacune)
3. On **fusionne** les deux parties

http://laure.gonnord.org/pro/teaching/AlgoProg1213_IMA/complexite_fusion.pdf

5- QuickSort

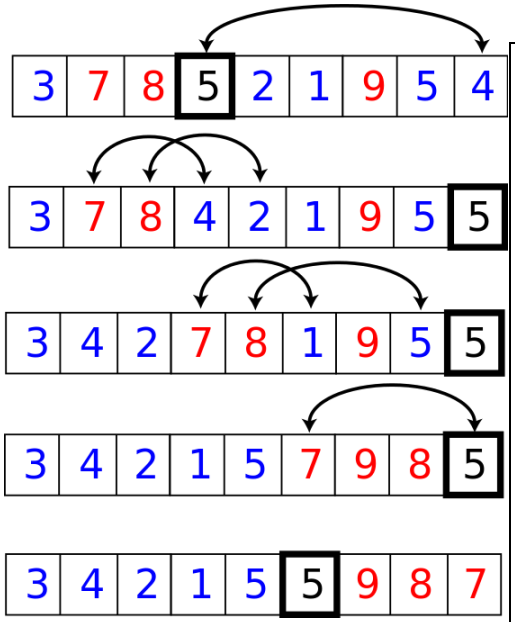
Le tri rapide (en anglais quicksort) est un algorithme de tri fondé sur la méthode de conception **diviser pour régner**.
La **complexité** moyenne du tri rapide pour n éléments est proportionnelle à **n log n**, ce qui est **optimal** pour un **tri par comparaison**, mais la complexité dans le **pire des cas** est **quadratique.**

Principe :
La méthode consiste à **placer** un élément du tableau (appelé **pivot**) à sa place **définitive**, en **permutant tous** les **éléments** de telle sorte que tous ceux qui sont **inférieurs** au pivot soient à sa **gauche** et que tous ceux qui sont **supérieurs** au pivot soient à sa **droite**. (Partitionnement)

Pour chacun des **sous-tableaux**, on définit un **nouveau pivot** et on répète l'opération de **partitionnement**. Ce processus est répété **récursivement**, jusqu'à ce que l'ensemble des éléments soit trié.

Partitionnement :

- on place le **pivot** à la **fin** (arbitrairement), en l'échangeant avec le dernier élément du sous-tableau ;
- on place **tous** les **éléments inférieurs** au pivot en **début** du sous-tableau ;
- on place le **pivot** à la fin des éléments **déplacés**.




```
partitionner(tableau T, premier, dernier, pivot)
   échanger T[pivot] et T[dernier]
   j := premier
   pour i de premier à dernier - 1
      si T[i] <= T[dernier] alors
         échanger T[i] et T[j]
         j := j + 1
   échanger T[dernier] et T[j]
   renvoyer j

 tri_rapide(tableau t, entier premier, entier dernier)
  début
   si premier < dernier alors
     pivot := choix_pivot(t, premier, dernier)
     pivot := partitionner(t, premier, dernier, pivot)
     tri_rapide(t ,premier, pivot-1)
     tri_rapide(t, pivot+1, dernier)
   fin si
  fin
```

IV- Accès Concurrent

1- Thread

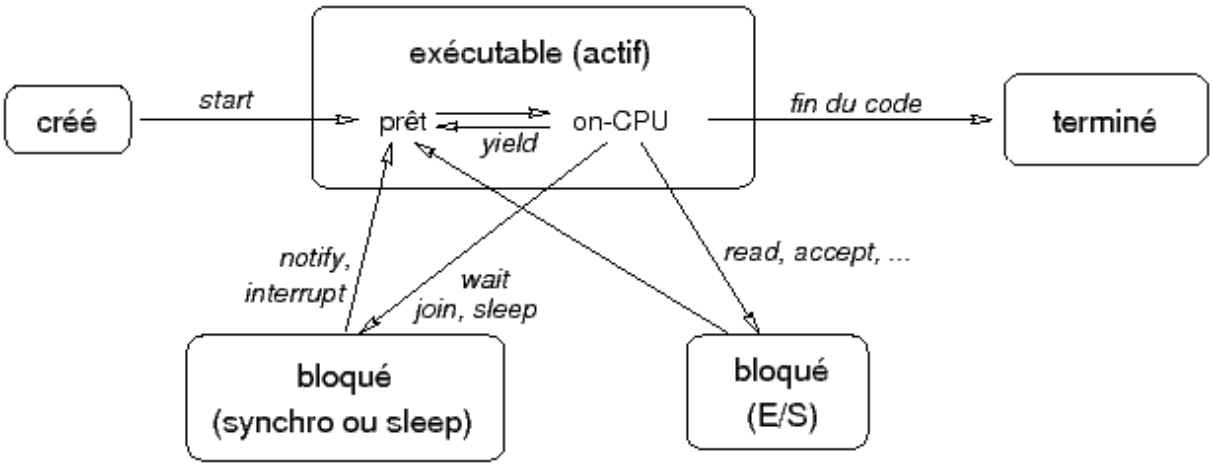
A- Thread vs process

- Un thread est une unité d'exécution faisant partie d'un programme. Plusieurs thread peuvent fonctionner **parallèlement** grâce à un **scheduler**.
- Un thread, c'est un découpage d'un **processus**

B- Création d'un thread

1. Classe héritant de ***Thread*** ou implémentant l'interface ***Runnable*** en implémentant la méthode **run()**
2. **Instanciation** de la classe
3. Appeler la méthode **start()**
4. Appeler la méthode **interrupt()**

C- Cycle de vie



- **New :** instanciation
- **Running :** start()
- **Waiting** : n'exécute aucun traitement et ne consomme aucune ressource CPU
  - appeler la méthode thread.sleep (temps en millisecondes) ;
  - appeler la méthode wait() ;
  - accéder à une ressource bloquante (flux, accès en base de données, etc.) ;
  - accéder à une instance sur laquelle un verrou a été posé ;
  - appeler la méthode suspend() du thread
- **Dead :** sorti de ~~start()~~ de manière naturelle, stop() ou exception

D- ThreadGroup

Permet de regrouper des Threads pour y lancer les mêmes actions (start, stop, ...)

```
ThreadGroup tg = new ThreadGroup("GroupName");
Thread tn = new Thread(tg, new MyThread(),"1");
```

2- Exclusion mutuelle :

A- Thread Safety

C'est la gestion des accès concurrent aux variables partagées (entre plusieurs threads) et mutable (dont la valeur peut changer).

a- Contrôle de l'exécution

Gérer l'ordre d'exécution des instructions du code et quand il doit s'exécuter simultanément.

b- Memory Visibility

La visibilité des modifications de mémoire par les autres threads.

En effet, chaque **CPU** possède plusieurs niveaux de **cache** entre lui et la **main memory** (Heap, RAM), et par conséquent, les threads peuvent voir la mémoire différemment.

La **JMM** (**J**ava **M**emory **M**odel) définie une **Memory Barrier** :

- **Read Barrier** : invalide la ~~mémoire locale~~ (Cache, Registre, ...) et impose la **lecture** à partir de la **main memory**
  ⇒ les modifications effectuées par les autres threads deviennent visibles par le thread courant.
- **Write Barrier** : **copie** la **mémoire locale** dans la **main memory**
  ⇒ les modifications effectuées par le thread courant deviennent visibles par les autres threads

La JMM effectue un **Read Barrier** en **entrant** dans un block **synchronized**, suivi d'un **Write Barrier** en **sortant** de ce dernier.

Pour les variables **volatiles**, une **lecture** déclenche un **read barrier** et une **écriture** déclenche un **write barrier**, synchronisant ainsi les opérations qui suivent même si les variables ne sont pas volatiles.

B- Locks

a- Intrinsic Locks

i- synchronized

1. Permet d'acquérir le **lock** sur un **objet** :
   - **bloc** de **code** : l'**objet** lui même
   - **méthode** : l'objet **appelant**,
   - méthode **statique** : la **classe**

   Empêche les autres threads à ~~exécuter le code en parallèle~~, jusqu'à ce qu'il **relâche** le **lock.**
   ⇒ Assure le **contrôle** de l'**exécution**
2. Toutes les modifications effectuées par un thread sont reflétées sur la **main memory** avant que d'autres threads n'y accèdent.
   ⇒ **flush** des caches CPU à l'**acquisition** et après le **relâchement** d'un **lock**
   ⇒ Assure la **visibilité de la mémoire**

ii- Volatile vs Synchronized

Volatile : Indique qu'une **variable** va être **modifiée** par **plusieurs threads**
La JVM garantie alors que l'on récupérera toujours une **valeur** à **jour**, en désactivant l'utilisation du ~~cache CPU~~.

Characteristic	Synchronized	Volatile

Type of variable	Object	Object or primitive
Null allowed?	No (synchronisation on the Object)	Yes (synchronisation on the Reference)
Can block?	Yes	No
All **cached** variables **synchronized** on **access**?	Yes	From Java 5 onwards
When synchronization happens	When you explicitly enter/exit a synchronized block	Whenever a volatile variable is accessed.
Can be used to combined several operations into an **atomic** operation?	**Yes**	Pre-Java 5 : **no**. Java 5 : Atomic get-set of volatiles possible

PS : Une variable **volatile** n'assure la **synchronisation** que pour une opération **atomique**, et par conséquent la ~~synchronisation~~ de **x++** (read, increment & set) qui est une opération composée n'est pas assurée. (il faut utiliser **Atomic Classes**)

iii- Variable Atomique

L'API Concurrent (`java.util.concurrent.atomic`), offre un certain nombre de **classe atomique** qui permettent de réaliser des opérations atomiques (++, ...) sans utiliser la ~~synchronisation~~ ⇒ très **performante**.

- **AtomicBoolean**, **AtomicInteger**, **AtomicLong** et **AtomicReference** : permettent de gérer les booléens, les entiers 32 bits et 64 bits, ainsi que les références sur d'autres objets.
- **AtomicIntegerArray**, **AtomicLongArray** et **AtomicReferenceArray** : permettent de gérer des tableaux d'entiers 32 et 64 bits, et les tableaux de références sur d'autres objets.

get(), ***intValue()***, ***longValue()***, ***floatValue()***, ***doubleValue()***
set(int newValue)
incrementAndGet() *:* ++i, ***decrementAndGet()*** *:* --i ***et addAndGet(int delta)*** : i+=delta
getAndIncrement() *:* i++, ***getAndDecrement()*** *:* i--, *et* ***getAndAdd(int delta)*** : i+=delta
compareAndSet(int expect, int value) : compare la valeur encapsulée avec expect, si ces deux valeurs sont égales, alors la valeur encapsulée devient value. Dans le cas d'objets, la comparaison utilise == et non pas la méthode equals().

b- Explicit Locks

Ce sont des alternatives plus flexibles et configurables pour les blocs `synchronized`

i- Semaphor

Permet de nous assurer, en toute sécurité, que seuls **n threads** peuvent accéder à certaines **ressources** à un moment donné.

- Compteur c
- **P** : **acquire** ⇒ if (c>0) c-- else wait
- **V** : **release** ⇒ c++ notify

ii- Monitor

destroy() : met fin brutalement au thread

join(long) : attend la fin de l'exécution du thread ou que le délai de garde soit écoulé.
sleep(long) : suspend l'exécution du thread appelant pendant la durée indiquée ou jusqu'à ce que l'activité soit interrompue (throws InterruptedException)
wait() : permet au thread qui l'exécute de se mettre en attente dans le wait-set de l'objet récepteur. Ainsi un thread peut décider de suspendre sa propre exécution.
notify() : permet à un thread de réveiller un des threads qui s'était mis en attente dans le wait-set de l'objet récepteur.
notifyAll() : ressemble à notify avec la différence qu'il réveille tous les threads qui était en attente au moment de son exécution.

iii- ReentrantLocks

- **Lock** : Le cas le plus simple d'un verrou qui peut être acquis et libéré
- **ReadWriteLock** : une implémentation du lock qui possède à la fois les verrous de lecture et écriture. plusieurs verrous de lecture peuvent être tenus à un moment à moins que le verrou exclusif en écriture est maintenu

Reentrant : Si un thread **acquiert** un **verrou** sur un **objet**, il peut **accéder** à **tous** les **blocs** de code **synchronisés** sur ce **dernier**.
synchronized est un **ReentrantLock**.
Les **Locks** réalisent une **Memory Barrier**

```
Lock l = new ReentrantLock();
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
Lock r = rwl.readLock();
Lock w = rwl.writeLock();
(l|r|w).lock();
try {
    // some business logic that needs to be synchronized
} finally {
    (l|r|w).unlock();
}
```

Méthodes :

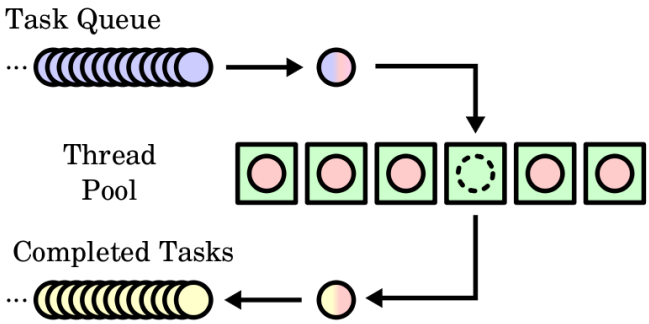
- ***lock()*** : tente d'acquérir ce verrou. Cette méthode rend la main quand le verrou a été acquis. Donc si un autre *thread* possède ce verrou, le *thread* courant attendra que cet autre *thread* le rende.
- ***lockInterruptibility()*** : tente d'acquérir ce verrou, sauf si le *thread* courant est dans l'état INTERRUPTED.
- ***tryLock()*** et ***tryLock(long time, TimeUnit unit)*** : tente d'acquérir ce verrou. S'il est immédiatement disponible, alors ces deux méthodes retournent true. S'il ne l'est pas, la première retourne immédiatement false, la seconde attend le temps indiqué en paramètre qu'il se libère.
- ***unlock()*** : libère ce verrou. Il est indispensable que cette méthode soit appelée dans tous les cas. Il faut donc appeler cette méthode dans une clause **finally**.

c- DeadLocks

Eviter les ~~DeadLocks~~ :

- Eviter les locks mutliples
- Utiliser un ordre pour acquérir les locks

3- Thread Pool (Executors)

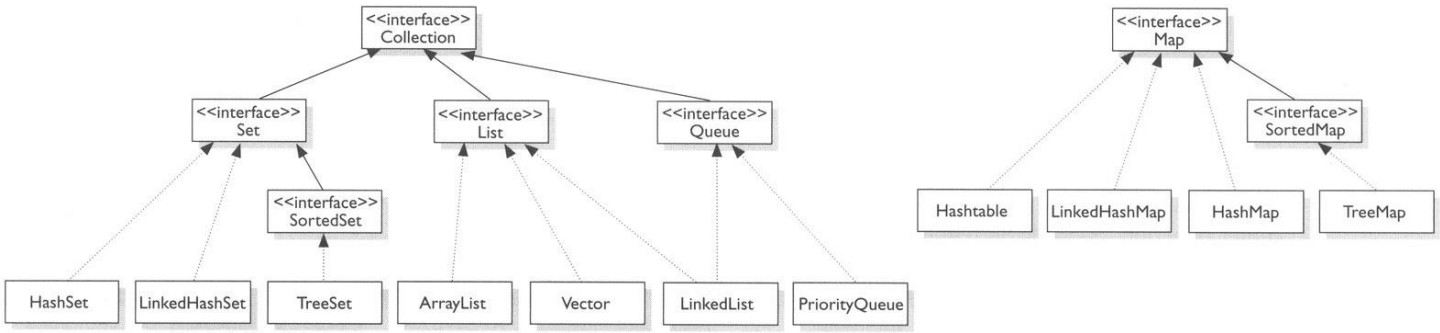


C'est un **design pattern** qui permet de créer **n threads** au lancement qui seront en charge d'exécuter des **tâches** mises en **files d'attente**.

En JAVA, c'est la classe **Executors** qui permet de créer un thread pool.

- **newCachedThreadPool()** et **newFixedThreadPool(int nThreads)** : crée un thread pool capable de créer des nouveaux threads à la demande, sans limite pour la 1ère, et au max **nThreads** pour la deuxième.
- **newScheduledThreadPool(int corePoolSize)** : permet de créer un thread pool permettant de réaliser des tâches périodiques :
  - **schedule(Runnable command, long delay, TimeUnit unit)** : crée une tâche qui ne se lancera qu'une seule fois, après le délai indiqué par le paramètre delay
  - **scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)** : lance la tâche passée en paramètre à intervalles de temps réguliers, indépendamment du temps que cette tâche met à s'exécuter
  - **scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)** : lance cette tâche, puis attend le délai initialDelay avant de la lancer à nouveau, de façon périodique
- **newSingleThreadExecutor()** et **newSingleThreadScheduledExecutor()** : Crée un thread pool avec un unique thread.

V- Collections :



Les **Iterators** des Collections sont **fail-fast** : Si après la création de l'iterator la liste a changée (sauf avec les méthodes remove et add de ce dernier), il va renvoyé une exception (`ConcurrentModificationException`)

1- List

Une liste est une collection **ordonnée**. L'utilisateur de celle-ci a un contrôle complet sur les éléments qu'il insère dedans, et il peut y accéder par l'entier de leur index.

- **Vector** (tableau dynamique) : synchronisé, ~~null~~, capacity (10), capacityIncrement (0)
- **ArrayList** (tableau dynamique) : ~~Thread Safe~~, null, capacity (10), capacityIncrement (amortized time cost, depend on JDK implementation)
- **LinkedList** (Liste doublement chaîné): ~~Thread Safe~~, ~~null~~

2- Set

Un Set est une collection qui n'accepte pas les ~~**doublons**~~, et au maximum une fois **null.** Il n'est pas ~~**ThreadSafe**~~

- **HashSet** : **+**rapide (Complexité O(1)), ~~ordonnée~~, implémentée avec **HashMap** (les valeurs sont stockées dans les clés de cette dernière)
- **TreeSet** : **-**rapide (Complexité O(log n)), ordonnée (Comparator et Comparable), implémenté avec **TreeMap**
- **LinkedHashSet** : entre HashSet et TreeSet, implémenté avec **HashTable** et **LinkedList**, garantissant ainsi l'**ordre** et la **rapidité**

3- Map

Une map est une collection qui associe une clé à une valeur.

- **HashTable** : synchronisé, ~~null~~, ~~ordonnée~~
- **HashMap** : ~~ThreadSafe~~, null, ~~ordonnée~~
- **LinkedHashMap** : ~~ThreadSafe~~, hérite de **HashMap** et implémente **LinkedList** : ordonnée (ordre d'insertion)
- **TreeMap** : ~~ThreadSafe~~, (hérite de SortedMap) ordonnée avec les clés (Comparable et Comparator)

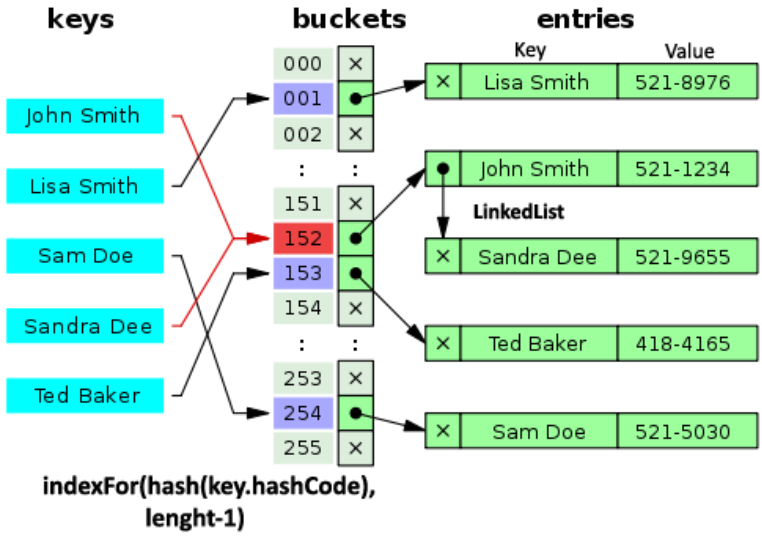
LoadFactor (0,75): pourcentage du remplissage de la map avant d'augmenter sa capacité (initial : 16), par défaut au double de sa taille

Map.keySet() → Set<K>
Map.entrySet() → EntrySet<K,V>

PS : ne pas oublier de surcharger **hashCode** et **equals** !

A- Fonctionnement de HashMap

http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html



- ✓ Si les deux **clés** ont le même **hashCode** :
  - S'ils sont **equals** ⇒ écrasement de la pair <Key,Value>
  - Sinon ⇒ enregistrement dans une liste chaînée d'Entry
- ✓ Quand une HashMap arrive à sa limite de taille (**LoadFactor**), elle se **rehash** : Allocation d'une nouvelle zone mémoire, recalcul du hash (dépends de la taille du tableau initial), et recopie de tout les pairs <Key,Value> en inversant les listes chaînées (copie plus rapide)

4- Queue

5- Synchronisation

L'API concurrent de JAVA propose deux sortes d'implémentation des collections :

- Collections **synchronisés** : Vector, HashTable (== synchronizedMap), Collections.synchronizedXXX(x)
  PS : L'**iterator** doit être **synchronisé** sur la collection

- Collections **concurrentes** :
  - **CopyOnWriteArrayList**, **CopyOnWriteArraySet** : Tableau recopié intégralement à chaque modification. Seules les modifications sont synchronisées.
  - **ConcurrentHashMap**
  - **ConcurrentSkipListSet**, **ConcurrentSkipListMap** : la version qui conserve l'**ordre** de ConcurrentHashMap

  Ajoute des méthodes **atomiques** : put(add)IfAbsent, remove, replace

	HashTable	**ConcurrentHashMaps**
Null	—	—
Thread-Safe	+	+
Synchronisation	Toute la Map Toutes opérations	Chaque partition de la Map (ConcurrencyLevel) Update seulement
Performance	—	+
Consistance des données	préservée	L'itération peut présenter des données non ~~MAJ~~
Iterator	Fail-fast	Fail-safe

* **fail-safe** : fait une copie des données (snapshot) avant d'itérer, ne renvoyant pas ainsi ~~ConcurrentModificationException~~. Les modifications sur les itérateurs (remove, set et add) ne sont pas prises en charge.

6- Comparable, Comparator

Résultat obtenu	**Comparator.*compare*(Object o1, Object o2)**	**Comparable.*compareTo*(Object o2) [Object o1 = objet de la classe qui implémente l'interface Comparable)**
-1 (entier négatif)	o1 < o2	o1 < o2
0 (égalité)	o1 == o2	o1 == o2
+1 (entier positif)	o1 > o2	o1 > o2

Java implémente l'**ordre naturel** pour certains de ses composants (grâce à l'interface ***Comparable***) : Byte, Short, Integer, Long, Float, Char, File, String, Date

Afin d'utiliser un **ordre plus personnalisé** (par exemple selon **plusieurs relations** : trier d'abords le nom, puis le prénom), il est conseillé de faire appel à l'interface ***Comparator***

Collections.sort(List list, Comparator comparator) : Si prend seulement list, elle utilise **comparable**, sinon, **comparator**

7- Implémentation de LinkedList

```
public class Link<T> {
	private T data;
	private Link<T> next;

	public Link(T data) {
		this.data = data;
		this.next = null;
	}
	// Getters & Setters
}
```

```
import java.util.Iterator;

public class LinkedList<T> implements Iterable<T> {
	private Link<T> first;
	public LinkedList() {
		first = null;
	}
	public boolean isEmpty() {
		return (first == null);
	}
	public void add(T element) {
		addLast(element);
	}
	public void addFirst(T element) {
		Link<T> link = new Link<T>(element);
		link.setNext(first);
```

```
            first = link;
      }
      public void addLast(T element) {
            // Parcourir la liste
            if (first != null) {
                  Link<T> link = first;
                  while (link.getNext() != null)
                        link = link.getNext();
                  // Insérer à la fin
                  Link<T> newLink = new Link<T>(element);
                  link.setNext(newLink);
            } else
                  addFirst(element);
      }
      public void clear() {
            first = null;
      }
      public T getFirst() {
            return first.getData();
      }
      public T getLast() {
            // Parcourir la liste
            Link<T> link = first;
            while (link.getNext() != null)
                  link = link.getNext();
            return link.getData();
      }
      public int size() {
            int size = 0;
            // Parcourir la liste
            Link<T> link = first;
            if (link != null)
            {
                  size++;
                  while (link.getNext() != null) {
                        link = link.getNext();
                        size++;
                  }
            }
            return size;
      }
      public T get(int index) {
            int i = 0;
            if (index >= this.size())
                  throw new IndexOutOfBoundsException();
            // Parcourir la liste
            Link<T> link = first;
            while (i < index) {
                  link = link.getNext();
                  i++;
            }
            return link.getData();
      }
      public Iterator<T> iterator() {
            return new Iterator<T>() {
                  private Link<T> cursor = first;

                  public boolean hasNext() {
                        return cursor.getNext()!=null;
                  }
                  public T next() {
```

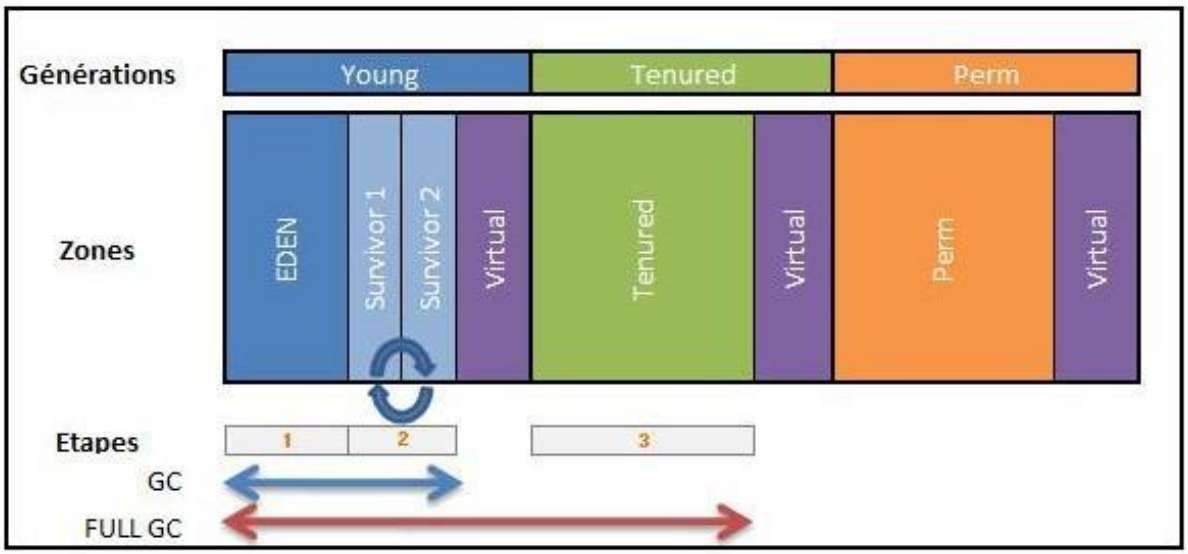
```
                        cursor = cursor.getNext();
                        return cursor.getData();
                    }
                    public void remove() {
//                      cursor.setNext(cursor.getNext());
                    }
                };
        }
}
```

VI- Garbage Collector

La JVM (machine virtuelle Java) utilise un processus de **récupération automatique** de la **mémoire** des **objets inutilisés** nommé **ramasse miettes.**
Il existe 3 types de GC :

- **Comptage de référence** : problème avec les cycles
- **Traversant :** algorithme de coloriage
- **Générationnel :** les données n'ont pas la même durée de vie

1- GC Générationnels (jdk6)

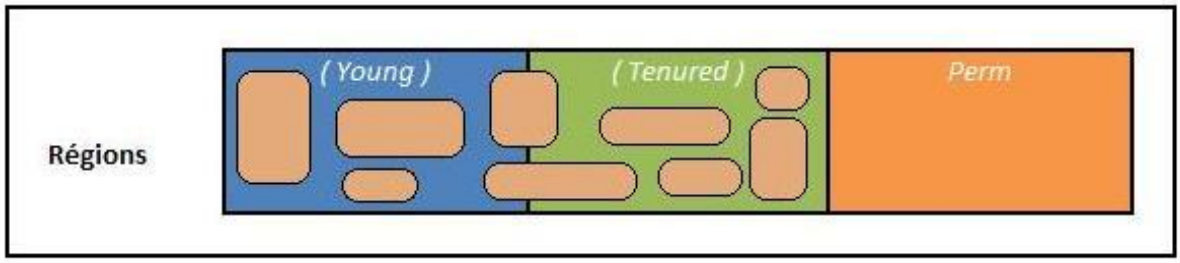


Les objets sont créés dans l'**EDEN** (1), puis déplacé dans le **survivor** (2) après un **GC**, puis dans le **Tenured** (3) puis **FullGC** quand s'est saturé.
Le GC effectue deux types de traitements :

- Un **nettoyage rapide** lorsque l'**EDEN** est rempli
- Un **nettoyage approfondi** (**FullGC**), plus gourmand en ressources, lorsque le **Tenured** est saturé. Il faut libérer la mémoire explicitement, et éventuellement la compacter (donc déplacer les objets et mettre à jour les références en conséquence)

- **Perm** : stocke les objets n'ayant pas besoin d'être ~~nettoyés~~ par le **GC**, comme les classes chargées par le **class loader** lors de l'exécution d'un programme. Ces objets resteront **disponibles** pendant toute la **durée de vie** de la **JVM**
- **Virtual** : ne sont pas utilisées par la **JVM** pour stocker des données. Elles correspondent à l'écart entre la **taille réelle** et la **taille max**. Leur réservation n'a pas encore été faite auprès système (Taille Heap Min (-Xms) < Taille Heap Max (-Xmx)).

2- GC Garbage First (G1) (jdk7)



http://blog.xebia.fr/2008/03/12/gc-generationnels-traditionnels-jdk6-vs-gc-garbage-first-jdk7/

L'idée de l'algorithme G1 (Garbage-First) est d'utiliser les spécificités des environnements **multiprocesseurs** afin d'approcher des performances '**temps réel'** tout en maintenant un **débit élevé**.
Pour parvenir à ce but, le tas n'est plus ~~divisé en générations~~, mais en un nombre beaucoup plus grand de **petites régions**.

Avantages :

- Meilleure **prédiction** des **temps** de **pauses** destinées au GC.
- **Courte pause** sans ~~fragmentation~~.
- Nettoyage acceptant la **parallélisassions** et la **concurrence**.
- Meilleure **utilisation** du **tas**.

VII- Types Primitifs

Type	**Signification**	**Taille (en octets)**	**Plage de valeurs acceptées**
char	Caractère Unicode	2	'\u0000' ? '\uffff' (0 à 65535)
byte	Entier très court	1	-128 ? +127
short	Entier court	2	-32 768 ? +32 767
int	Entier	4	-2^{31}?$-2{,}147\times10^{9}$? $+2^{31}-1$?$2{,}147\times10^{9}$
long	Entier long (L, sinon int)	8	-2^{63}?$-9{,}223\times10^{18}$? $+2^{63}-1$?$9{,}223\times10^{18}$
float	Nombre réel simple (f, sinon par défaut double)	4	$\pm2^{-149}$?1.4×10^{-45} ? $\pm2^{128}-2^{104}$?3.4×10^{38}
double	Nombre réel double	8	$\pm2^{-1074}$?$4{,}9\times10^{-324}$? $\pm2^{1024}-2^{971}$?$1{,}8\times10^{308}$
boolean	Valeur logique (booléen)	1	true (vrai), ou false (faux)
octal	Commence avec 0		0-7
hex	Commence avec 0x		0-F

binaire	Commence avec 0b/0B		
BigDecimal	unscaledValue × 10^{-scale}		

Le codage des entiers se fait en binaire pour les entiers positifs et en binaire en complément à deux pour les entiers négatifs :

- 9 est codé sur un octet en 00001001
- -9 est codé sur un octet en 11110111
- -1 est codé sur un octet en 11111111
- -128 est codé sur un octet en 10000000
- 127 est codé sur un octet en 01111111

VIII- Cycle de vie d'une classe dans la JVM

1. **chargement** (loading) : permet de lire le bytecode dans la machine virtuelle
2. **liaison** (linking) : permet de rendre utilisable le bytecode. Cette étape est composée de trois processus (vérification, préparation, résolution)
    1. La **vérification** (verify) permet de s'assurer que le bytecode est compatible avec la machine virtuelle
    2. La **préparation** (prepare) effectue l'allocation mémoire nécessaire à la classe
    3. La **résolution** (resolve) transforme les références symboliques du constant pool en références mémoire.
3. **initialisation** (initialization) : initialisation des valeurs des variables
4. **instanciation** (instantiation)
5. **récupération de la mémoire** (garbage collection)
6. **finalisation** (finalization)
7. **déchargement** (unloading)

1- ClassLoaders

La JVM **recherche** et **charge** les classes requises dans un **ordre** bien précis grâce à la **délégation** des **classloaders** :

1. **bootstrap classes** : Les classes de bootstrap qui sont les classes fournies avec la plate-forme **Java SE** dans le fichier **rt.jar**
2. **extension classes** : Les classes d'extension qui sont packagées sous forme de fichiers .**jar** et stockées dans le répertoire **lib/ext** du **JRE**
3. **user classes** : Les classes d'utilisateurs qui sont écrites par les **développeurs** ou des **tiers**

Méthode de la classe ClassLoader	**Rôle**
loadClass()	charger une classe en demandant au préalable au classloader père de réaliser l'opération
findClass()	charger une classe
defineClass()	Ajouter le bytecode de la classe dans la machine virtuelle

```
ClassLoader.getSystemClassLoader()
ClassLoader cl = TestClassLoader1a.class.getClassLoader()
cl.loadClass("java.lang.Number"));

Class.forName("java.lang.Number") ;

Number.class ;

URLClassLoader loader;
try {
      loader = new URLClassLoader(new URL[] {
                  new URL("file:///C:/Program
                              Files/Java/jre1.6.0_03/lib/rt.jar") });
      Class<?> stringClass = loader.loadClass("java.lang.String");
      System.out.println(stringClass.getClassLoader());
      System.out.println(String.class.getClassLoader());
}catch (MalformedURLException e) {
e.printStackTrace();
} catch (ClassNotFoundException e) {
e.printStackTrace();
```

IX- J2EE

1- JavaBean

Des **composants réutilisables** sont des objets autonomes qui doivent pouvoir être facilement **assemblés** entres eux pour créer un programme

Les **beans** sont des classes JAVA respectant :

- **Constructeur par défaut** (sans ~~paramètre~~) **public**
- Les **propriétés** (attributs **privées**) accessibles avec des **Accesseurs public** avec des règles de nomages
- implémente l'interface **serializable**
- Emettent des **événements** en gérant une liste de **listeners** qui s'y abonnent via des méthodes :
  - (add|remove)xxxListener(xxxListener listener)
  - (add|remove)PropertyChangeListener(PropertyChangeListener listener)

2- Base de Données

A- Jointure

Interne	Externe
SELECT * FROM A **[INNER] JOIN** B **ON** A.a = B.b	SELECT * FROM A **(LEFT|RIGHT) JOIN** B **ON** A.a = B.b
Retourne **uniquement** les lignes pour lesquelles A.a et B.b correspondent	Retourne en **plus** les lignes non ~~**correspondantes**~~ de la table (**gauche**|**droite**)

B- Index

CREATE INDEX 'nom' ON 'table' ;
+ Temps de réponse des requêtes select
- MAJ/Insertion

C- Procédure stockée

Permer de stoquer des scripts pour pouvoir les réutiliser facilement

CREATE PROCEDURE nom(**[**parametre, ...**]**)
BEGIN
Select ...
END

CALL nom(**@**variable)

3- JDBC

JDBC (**J**ava **D**ataBase **C**onnectivity), c'est une **API Java** standard qui permet de s'interconnecter avec un large éventail de BD **SQL**, mais aussi avec d'autres sources de données tabulaires (feuilles de calcul, fichiers plats, ...).

```
try{
      //ETAPE 1: Charger les pilotes JDBC
      Class.forName("com.mysql.jdbc.Driver");
      //ETAPE 2: Ouvrir une connexion
      Connection conn =
                  DriverManager.getConnection(DB_URL,USER,PASS);
      Statement stmt = conn.createStatement();
      //ETAPE 3: Executer une requête
      ResultSet rs = stmt.executeQuery(sql)|executeUpdate(sql);

      //ETAPE 2-3 bis: PreparedStatement (la requête est
      précompliée)
      PreparedStatement stmt = conn.prepareStatement(sql);
      ResultSet rs = stmt.executeQuery()|executeUpdate();

      //ETAPE 4: Extraire les données de ResultSet
      while(rs.next()){
            Xxx id = rs.getXxx("N°|id");
      }
      rs.close();
// Erreur JDBC
} catch(SQLException se){
      se.printStackTrace();
// Erreur Class.forName
} catch(Exception e){
      e.printStackTrace();
} finally{
      //ETAPE 5: Fermer la connexion
      stmt.close();
      conn.close();
 }
```

A- ResultSet vs RowSet

	ResultSet	JdbcRowSet	CachedRowSet

Connexion maintenue	✓	✓	—
Sérialisation	—	—	✓
Performance	+	+	- (Données préchargées)
JavaBean	—	✓	✓

- **RowSet** hérite de **ResultSet**
- Quand la BD est compatible, les deux peuvent être **scrollable** : le **curseur** peut se déplacer vers l'**avant** et l'**arrière** et dans une **position absolue**.

B- DataSource vs DataManager

	DataManager	DataSource
Version	Avant JDBC 3.0	JDBC 3.0
Chargement des Drivers	✓	—
Portabilité	Codé en dur, recompilation	Fichier de configuration
Connection Pool	— (framework, implémentation)	✓
Transactions Distribués	—	✓
Connexion	Host, Port, user, pass + Driver	JNDI

C- Connection Pool

Mécanisme permettant de **réutiliser** les **connexions créées** pour éviter la création systématique de nouvelles instances de *Connection* réduisant ainsi la consommation de ressources.

Un pool de connexions ne ~~**ferme**~~ pas les connexions lors de l'appel à la méthode ***close()***. Au lieu de fermer directement la connexion, celle-ci est "**retournée**" au **pool** et peut être utilisée ultérieurement.

Les Pool de connexion peuvent être gérés par les serveurs d'applications.

D- Transaction

Une connexion est par défaut en mode **AutoCommit** : chaque **statement** terminé est comité.

```
try{
      conn.setAutoCommit(false);
      ...
      Savepoint save1 = con.setSavepoint();
      ...
      conn.commit();
} catch (Exception e) {
      conn.rollback()|rollback(save1);
}
```

4- Spring

C'est un **conteneur léger** qui se charge de la **création** et la **mise** en **relation** d'**objets**.

Le Framework Spring permet de **construire** et de **définir** l'**infrastructure** d'une application **J2EE**. C'est un framework **multi-couche** pouvant **s'insérer** au niveau de

toutes les couches (**MVC**), ainsi il permet d'insérer **Hibernate** dans la couche de **persistance** et **Struts** dans la couche **présentation**.

Spring offre les fonctionnalités suivantes :

- **IOC (I**nversion **O**f **C**ontrol): C'est un design-pattern permettant de passer le **contrôle** de l'**exécution** de l'application au **framework**, permettant ainsi de **découpler** les modules :
    - **Recherche de dépendances**
    - **Injection de dépendances** : les dépendances entre les différentes classes sont déterminés dynamiquement.
- **AOP** (**A**spect **O**riented **P**rogramming) : Un paradigme de programmation permettant de **séparer** la logique **métier** de la **technique** (log, trace, ...)
- Spring **MVC** web application et **RESTful** web service framework

```
public interface MessageService {
    String getMessage();
}
```

```
@Component
public class MessagePrinter {

    @Autowired // demande a spring de chercher le bean à instancier
    private MessageService service;

    public void printMessage() {
        System.out.println(this.service.getMessage());
    }
}
```

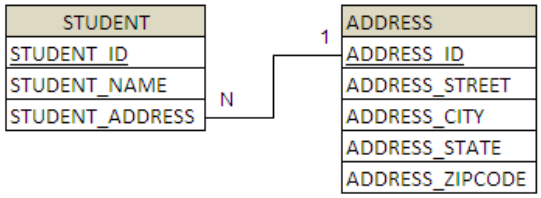
```
@Configuration
@ComponentScan
public class Application {


    @Bean // Bean à injecter
    MessageService mockMessageService() {
        return new MessageService() {
            public String getMessage() {
              return "Hello World!";
            }
        };
    }

  public static void main(String[] args) {
      ApplicationContext context =
          new AnnotationConfigApplicationContext(Application.class);
      MessagePrinter printer = context.getBean(MessagePrinter.class);
      printer.printMessage();
  }
}
```

5- Hibernate

Hibernate est un framework **ORM** (**O**bject **R**elationnal **M**apping) qui permet de gérer la **persistance** des **objets** en **BDR** (**B**ase de **D**onnées **R**elationnel) en **remplaçant** les **accès** à la BD par des **appels** à des **méthodes** objet de **haut niveau**.

Hibernate implémente **JPA** + de nouvelles fonctionnalités (**HQL**, **Criteria**)

A- Configuration




```java
@Entity
@Table(name = "STUDENT")
public class Student {
	@Id
	@GeneratedValue
	@Column(name = "STUDENT_ID")
	private long studentId;

	@Column(name = "STUDENT_NAME", nullable = false, length = 100)
	private String studentName;

	// Eager : Charger le tout en même
	// Lazy : Charger à la demande
	@ManyToOne(fetch = fetchtype.lazy)
	private Address studentAddress;

	public Student() {
	}
}
```

```java
@Entity
@Table(name = "ADDRESS")
public class Address {
	@Id
	@GeneratedValue
	@Column(name = "ADDRESS_ID")
	private long addressId;

	@Column(name = "ADDRESS_STREET", nullable = false, length=250)
	private String street;

	@Column(name = "ADDRESS_CITY", nullable = false, length=50)
	private String city;

	@Column(name = "ADDRESS_STATE", nullable = false, length=50)
	private String state;

	@Column(name = "ADDRESS_ZIPCODE", nullable = false, length=10)
	private String zipcode;

	public Address() {
	}
}
```

B- Implémentation d'Hibernate :

- DTO (**D**ata **T**ransfert **O**bject) : Son but est de simplifier les transferts de données entre les sous-systèmes d'une application logiciel. Les objets de transfert de données sont souvent utilisés en conjonction des objets d'accès aux données.

- DAO (**D**ata **A**ccess **O**bject) : permet de **regrouper** les **accès** aux **données persistantes** dans des **classes** à **part**. Il s'agit surtout de ne pas ~~écrire~~ ces accès dans les **classes** "**métier**", qui ne seront modifiées que si les règles de gestion métier changent. Il complète le modèle **MVC**.

- Façace pour l'entityManager

- Relfection pour faire le mapping entre les entity et les noms de tables
Annotations
Factory pour les sessions

C- Cache

Hibernate fonctionne avec deux niveaux de caches :

- **Cache 1er niveau** : est lié à la ***session* Hibernate**, et les objets qui sont cachés ne sont visibles que pour une seule transaction.
- **Cache 2ème niveau** : lié à la ***session factory*** d'Hibernate, Les objets cachés sont donc visibles depuis l'**ensemble** des **transactions**.
  Permet de cacher des **entités**, ainsi que des **associations** de type **collection**.
  (Non activé par défaut)

X- Socket

1- Définition

Interface logicielle avec les services du Système d'Exploitation qui **masque** le travail nécessaire pour de **gestion** du **réseau** pour **établir** une **connexion**, puis d'**envoyer** et **recevoir** des données.

2- Type de Socket

- **Connecté** (**TCP**) : **+** fiable **-** performant accusé de réception
- **Non ~~Connecté~~** (**UDP**) : **-** fiable **+** performant ~~accusé de réception~~

3- JAVA NIO (Non-Blocking IO)

http://www.onjava.com/pub/a/onjava/2002/09/04/nio.html

XI- Utilitaire de développement

JUnit/Jenkins/Sonar/TDD
Scrum

1- JUnit

JUnit est un framework de **test unitaire** pour le langage de programmation Java

Test Unitaire :

2- Scrum