

# Bienvenue !

Pierre-Julien VILLOUD



✉ [pjvilloud@protonmail.com](mailto:pjvilloud@protonmail.com)

📄 <https://github.com/pjvilloud>

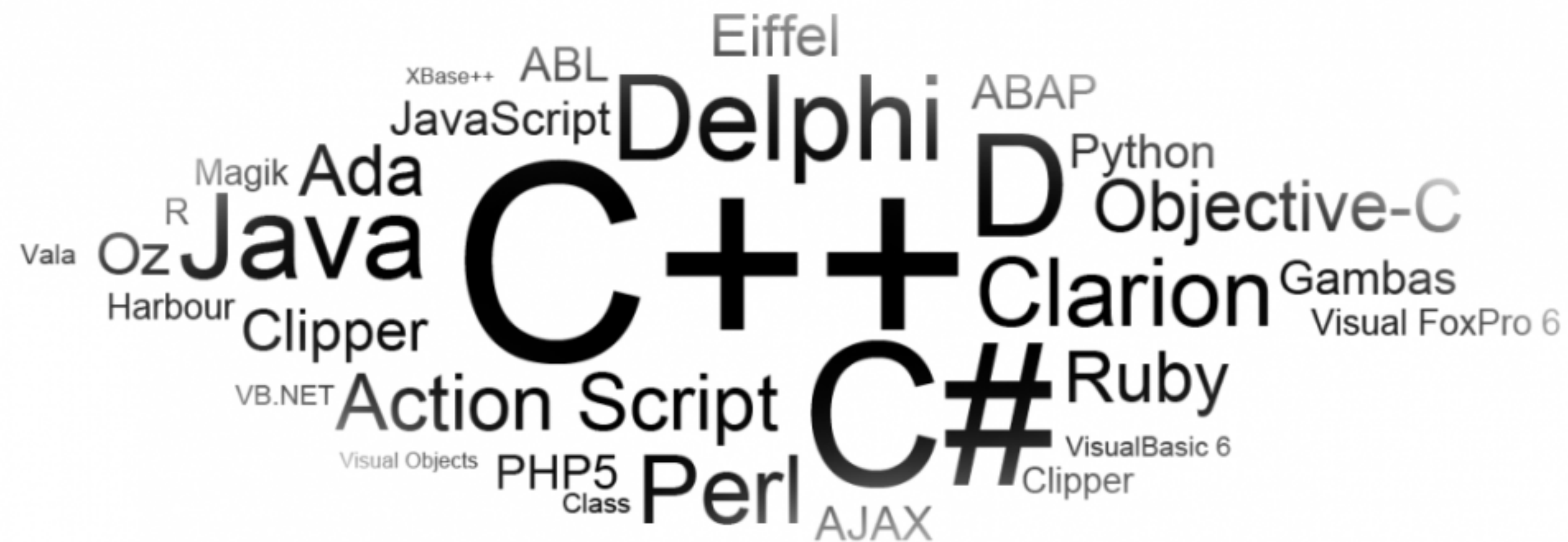
in <https://linkedin.com/pjvilloud>

🌐 <http://pjvilloud.github.io>

# Introduction

⚠ Le cours [Java 210](#) est un pré-requis à ce cours.

ℹ La programmation objet, ou POO est de nos jours le principal paradigme de programmation.



ℹ Java est un langage totalement orienté objet

# Vocabulaire de la POO

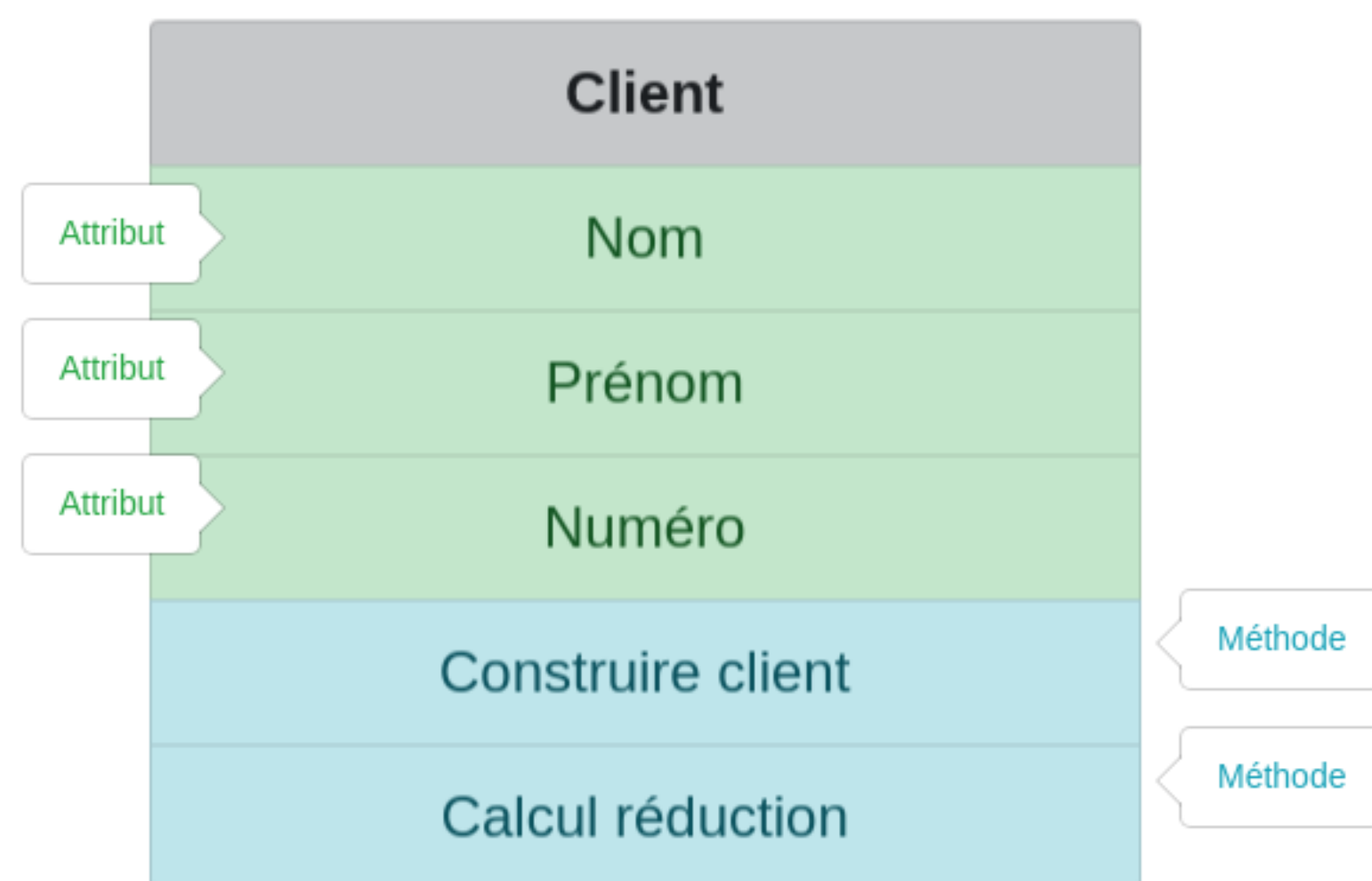
Objet
Instance
Classe
Attributs
Méthodes

# Les notions principales

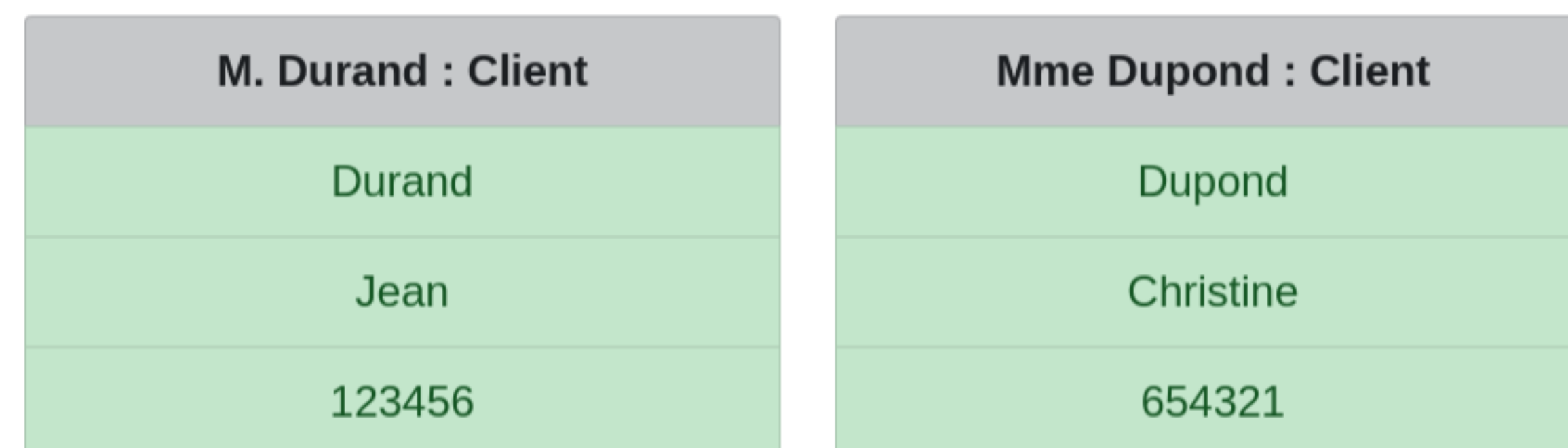
📘 Une *classe* est un modèle regroupant des données et des fonctions internes ou accessibles depuis l'extérieur.

📘 Un objet est une *instance* d'une classe. Il est *construit* à partir de la *classe*. C'est en réalité une *référence* vers un espace mémoire contenant l'objet en lui-même.

## La classe



## Les objets (ou instances)



📘 Les *attributs* représentent les données des objets.

📘 Les *méthodes* représentent les fonctions et traitements effectués sur les objets.



# Relations entre les classes

Les classes ont trois types principaux de relations entre elles : **dépendance**, **agrégation** et **héritage**. Exemple avec les classes `Produit`, `Commande`, `CommandeSpeciale` et `Client` d'un système de gestion de commandes.

Relation	Équivalent	Exemple
Dépendance	<i>utilise</i>	Une <code>Commande</code> va utiliser <code>Client</code> afin de savoir si le compte est suffisamment crédité. En revanche, <code>Client</code> ne dépend pas de <code>Commande</code> .
Agrégation	<i>possède</i>	Une <code>Commande</code> possède ou contient des instances de <code>Produit</code> .
Héritage	<i>est</i>	Une <code>CommandeSpeciale</code> est une (ou hérite de) <code>Commande</code> . Toutes les données et comportements de <code>Commande</code> sont passés à <code>CommandeSpeciale</code> . <code>CommandeSpeciale</code> peut définir de nouvelles données et méthodes ou redéfinir des comportements dont elle a hérité.

# Comparaison avec la programmation procédurale

## Programmation procédurale

Identification des tâches et découpage en sous-tâches
Écriture de procédures pour résoudre ces tâches
Convient pour des problèmes simples
Données globales

## Programmation orientée objet

Identification des objets-métier en classes
Définition des méthodes dans chaque classe responsable
Permet de modéliser des problèmes plus complexes
Données protégées (principe de responsabilité)



```
rotation(forme, typeForme){
  if(typeForme == "Carré") {
    rotationCarre(forme);
  } else if(typeForme == "Rond") {
    //On ne fait rien
  } else if(typeForme == "Triangle") {
    rotationTriangle(forme);
  }
}
```

```
rotationCarre(forme) {...}
rotationTriangle(forme) {...}
```

On peut appeler `rotationCarre` avec une forme `rond...`

La classe `Triangle` est responsable de sa méthode `rotation`

```
class Forme {
  void rotation() {}
}

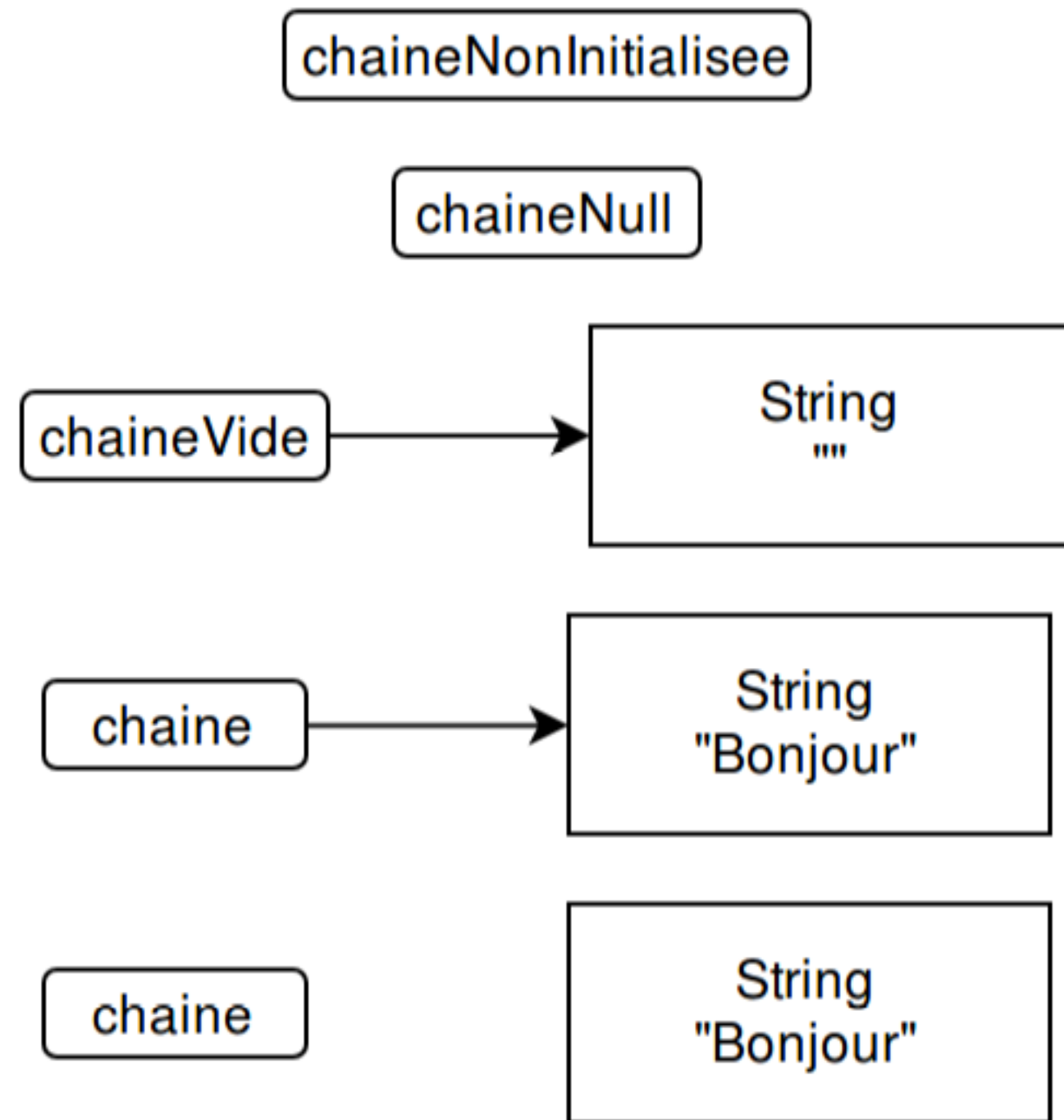
class Carre extends Forme {
  void rotation(){ /* Rotation Carre*/ }
}

class Rond extends Forme {
}

class Triangle extends Forme {
  void rotation(){ /* Rotation Triangle*/ }
}
```

# La notion de référence

📌 Lorsqu'on construit un objet, les données le constituant sont stockées dans un endroit en mémoire qui n'est pas directement accessible. Pour y accéder, on utilise une **référence** qui correspond au *point d'accès* en mémoire de cet objet. C'est cette référence qui est stockée dans une variable.



```
String chaineNonInitialisee;
```

```
String chaineNull = null;
```

```
String chaineVide = "";
```

```
String chaineVide = new String("");
```

```
String chaine = "Bonjour";
```

```
chaine = null;
```

Ces deux notations sont équivalentes d'un point de vue mémoire mais diffèrent au niveau programmation. `chaineNonInitialisee` est une variable qui n'est pas initialisée, `chaineNull` est une variable initialisée à la valeur `null`.

Ces deux notations sont équivalentes. Un espace mémoire est alloué dans lequel est mis une chaîne de caractère vide. La variable `chaineVide` est une référence vers cet espace mémoire.

📌 Une référence peut être `null`, c'est-à-dire qu'elle ne fait référence à aucun objet. Une variable de type objet non initialisée n'a pas automatiquement la valeur `null`. Le *ramasse-miette* est un mécanisme interne à Java qui s'occupe à l'exécution, de gérer les espaces mémoires qui ne sont plus référencés.



# Les classes standard

**i** Java 8 propose nativement plus de 4000 classes permettant la construction d'interfaces graphiques, la manipulation de fichiers, les collections ou encore la sécurité, les bases de données...

String

Math

Classes enveloppes

Collections



# La classe `String`

**i** Déjà vu dans le [cours précédent](#), la classe standard `String` contient 4 attributs, 67 méthodes, 15 constructeurs, 3169 lignes...

# La classe `Math`

La classe `Math` permet d'effectuer des opérations mathématiques avancées.

## Opération

## Méthode → retour

## Exemple

Arrondi

```
round(float ou double) → int ou long  
ceil(double) → double  
floor(double) → double
```

```
Math.round(4.56); //5  
Math.ceil(4.23); //5  
Math.floor(5.56); //5
```

Puissance

```
pow(double base, double exposant) → double
```

```
Math.pow(2, 3); //8
```

Minimum/Maximum

```
min(numeric, numeric) → numeric  
max(numeric, numeric) → numeric
```

```
Math.min(4, 5); //4  
Math.max(4, 5); //5
```

...

...

...

# Classes enveloppes

**i** Il existe une classe enveloppe par type primitif. Ces classes encapsulent les types primitifs et permettent de manipuler ces types comme des objets ordinaires. Elles offrent également de nombreuses méthodes en accès statique.

`boolean` → Boolean

`Boolean.logicalXor(a, b);` //Ou exclusif logique entre deux booléens

`char` → Character

`Character.isDigit('5');` //true; Test si le caractère est chiffre

`byte` → Byte

`Byte.decode("0x6b");` //107; Décodage de valeurs hexadécimales

`short` → Short

`Short.parseShort("5");` //Renvoie le Short 5

`int` → Integer

`Integer.max(4, 5);` //5

`long` → Long

`Long.numberOfTrailingZeros(500);` //2

`float` → Float

`Float.isNaN(Float.POSITIVE_INFINITY - Float.POSITIVE_INFINITY);` //true

`double` → Double

`Double.isInfinite(Double.NEGATIVE_INFINITY / 0.0);` //true

# Autoboxing et Unboxing

**i** Afin de faciliter l'utilisation des classes enveloppes numériques, Java a mis en place deux mécanismes : l'*Autoboxing* et l'*Unboxing*

```
Integer monInteger = 127;
```

## Autoboxing

L'autoboxing est la conversion automatique que le compilateur Java effectue entre les types primitifs et les classes enveloppes correspondantes. Ainsi, cette ligne est équivalente à `monInteger = new Integer(127)`

```
monInteger += 1;
```

## Unboxing

L'unboxing est l'inverse de l'autoboxing, c'est-à-dire la conversion automatique entre une instance d'une classe enveloppe et sa valeur dans le type primitif correspondant. L'opérateur `+=` s'applique normalement à des `int`.

**⚠ Attention à ces mécanismes qui, lorsqu'ils ne sont pas compris peuvent produire des résultats inattendus !**

## Vrai

Il y a Unboxing. Les valeurs primitives `int` des deux instances de `Integer` sont récupérées pour pouvoir être utilisées avec l'opérateur `<=`. 128 est bien inférieur ou égal à 128.

```
Integer monInteger = 128;  
Integer monInteger2 = 128;  
monInteger <= monInteger2;  
monInteger == monInteger2;  
monInteger.equals(monInteger2)
```

## Faux

Il n'y a pas Unboxing. Ce sont les références des objets qui sont comparées. Les deux objets étant des instances distinctes, la comparaison avec l'opérateur `==` renvoie `false`.

## Vrai

La méthode `equals` de la classe `Integer` est appelée et cette dernière compare bien les valeurs, et non les références.

```
Integer monInteger = 127;  
Integer monInteger2 = 127;  
monInteger == monInteger2
```

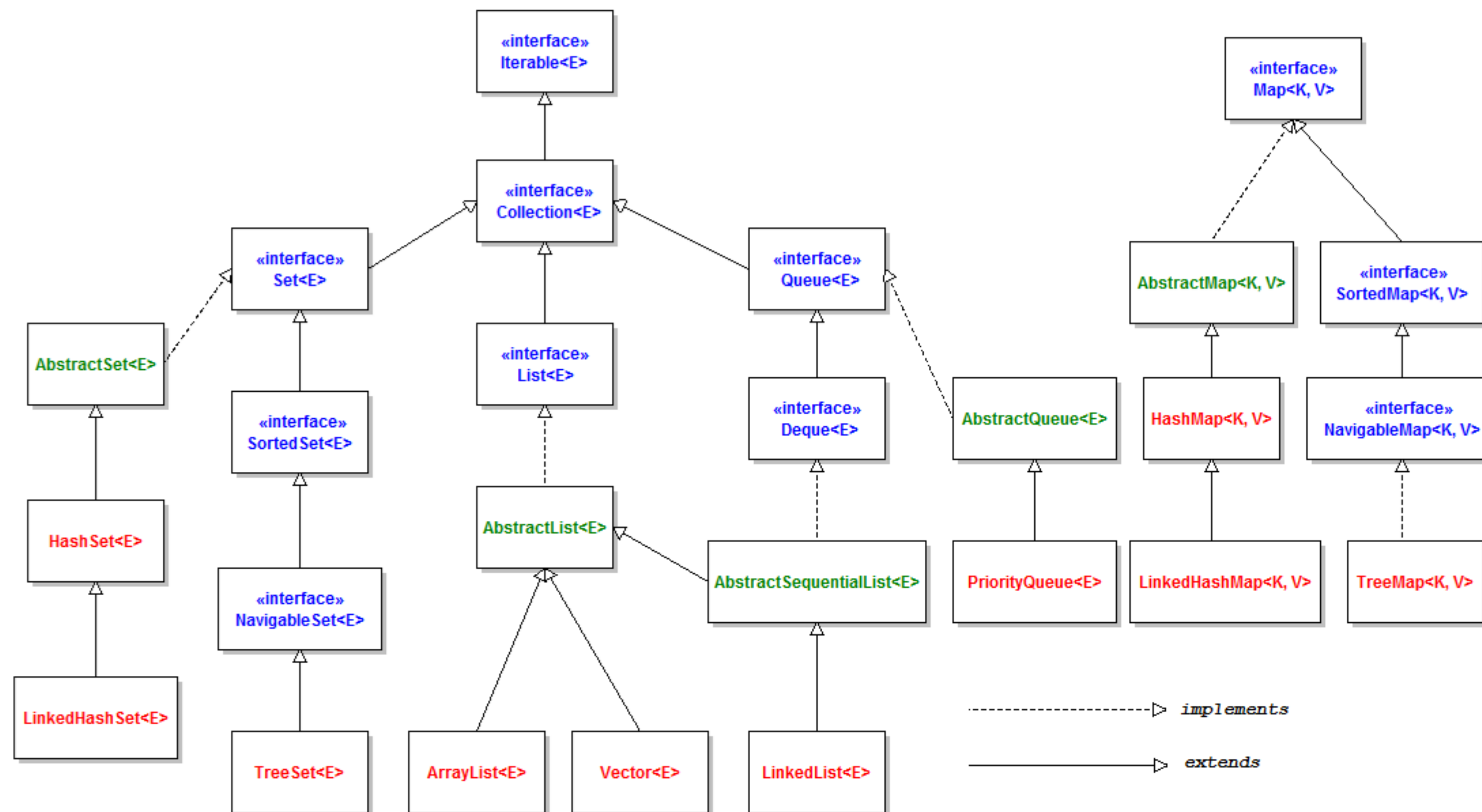
## Vrai

Il n'y a pas Unboxing mais la JVM met en cache les `Integer` entre -127 et 127 pour des raisons de performances. Ainsi même en créant deux instances différentes avec la valeur 127, ce seront en fait deux références vers le même objet en cache...



# Les collections

📘 Java propose un certain nombre de classes pour gérer des collections d'objet.



📘 Pour savoir quelle collection choisir par rapport à vos besoins, c'est par [ici](#).

# Les listes `ArrayList`

Les listes `ArrayList` permettent de gérer un ensemble ordonné d'objets de même type sans nécessité de déclarer une taille.

```
ArrayList<String> noms = new ArrayList();
noms.add("Jean");//[ "Jean" ]
noms.add(0, "Pierre");//[ "Pierre", "Jean" ]
noms.addAll(Arrays.asList("Simon", "Jacques"));//[ "Pierre", "Jean", "Simon", "Jacques" ]
noms.isEmpty();//false
noms.contains("Simon");//true
noms.indexOf("Pierre");//0
noms.size();//4
noms.get(2);//"Simon"
noms.remove(1);//[ "Pierre", "Simon", "Jacques" ]
noms.remove("Pierre");//[ "Simon", "Jacques" ]
noms.sort(String::compareTo);//Notation Java 8 [ "Jacques", "Simon" ]
noms.sort(new Comparator<String>() {
    public int compare(String o1, String o2) {
        return o1.compareTo(o2);
    }
});//Notation Java < 8 [ "Jacques", "Simon" ]
for (String prenom : noms){
    System.out.println(prenom);
}
```

Déclaration et initialisation d'une liste de `String`

Ajout d'éléments  
Ajout d'un élément en fin de liste ou à un indice donné ou concaténation avec une autre liste

Recherche  
Recherche d'un élément dans la liste

Suppression d'éléments  
Suppression d'éléments par index ou par élément

Tri  
Tri d'une liste avec déclaration explicite d'un comparateur ou avec utilisation de lambda

Parcours d'une liste

⚠ La classe `Vector` est à éviter pour des raisons notamment de performances. Utiliser la classe `ArrayList` à la place.

# Les ensembles **HashSet**

**i** Les ensembles **HashSet** permettent de gérer un ensemble non ordonné d'objets **distincts** de même type sans nécessité de déclarer une taille.

```
HashSet<String> setPrenoms = new HashSet();
```

Déclaration et initialisation d'une liste de **String**

```
setPrenoms.add("Jean");//[ "Jean" ]
```

```
setPrenoms.add("Jean");//N'a aucun effet car l'item est déjà présent !
```

```
setPrenoms.add("Pierre");[ "Pierre", "Jean" ]
```

```
setPrenoms.addAll(Arrays.asList("Simon", "Jacques"));//[ "Pierre", "Jean", "Simon", "Jacques" ]
```

```
setPrenoms.isEmpty();//false
```

```
setPrenoms.contains("Simon");//true
```

```
setPrenoms.size();//4
```

```
setPrenoms.remove("Pierre");//[ "Jean", "Simon", "Jacques" ]
```

Ajout d'éléments

Ajout d'un élément en fin de liste ou à un indice donné ou concaténation avec une autre liste

Recherche

Recherche d'un élément dans la liste

Suppression d'éléments

Suppression d'éléments par élément

```
for (String prenom : setPrenoms){
```

```
    System.out.println(prenom);//L'ordre d'affichage est potentiellement différent de l'ordre d'insertion.
```

```
}
```

Parcours d'une ensemble



# Les tables de hachage **HashMap**

Les tables de hachage **HashMap** permettent gérer un ensemble de couple clé/valeur. Les clés sont **distinctes** mais pas les valeurs.

```
HashMap<Integer, String> mapNombres = new HashMap();  
  
mapNombres.put(1, "Un");//[ 1 => "Un" ]  
mapNombres.put(3, "Trois");//[ 1 => "Un", 3 => "Trois" ]  
mapNombres.put(2, "DEUX");//[ 1 => "Un", 3 => "Trois", 2 => "DEUX" ]  
mapNombres.put(2, "Deux");//[ 1 => "Un", 3 => "Trois", 2 => "Deux" ]  
mapNombres.putIfAbsent(2, "AAA");//[ 1 => "Un", 3 => "Trois", 2 => "Deux" ]  
mapNombres.isEmpty();//false  
mapNombres.containsKey(2);//true  
mapNombres.containsValue("Un");//true  
mapNombres.size();//3  
mapNombres.remove(2);//[ 1 => "Un", 3 => "Trois" ]  
  
for (Integer n : mapNombres.keySet()){  
    System.out.println(mapNombres.get(n));//Affiche "Un" et "Trois"  
}
```

Déclaration et initialisation d'une liste de **String**

Ajout d'éléments

Ajout de couple clé/valeur. L'ajout d'une clé déjà existante remplace la valeur précédente (sauf utilisation de **putIfAbsent**)

Recherche

Recherche d'un élément dans la table par clé ou par valeur

Suppression d'éléments

Suppression d'éléments par clé

Parcours d'une table de hachage



# Créer ses propres classes

📘 Voici les différentes étapes pour créer sa propre classe

Définir son nom et la placer dans un **package**

Définir ses **attributs** et ses relations d'**agrégation**

Implémenter un ou plusieurs **constructeurs**

Implémenter des **méthodes**

Gérer la **visibilité** des attributs et des méthodes

# La notion de package java ▾

**i** Java permet de regrouper les classes en un ensemble appelé **package** qui fonctionne comme une arborescence de fichiers

Pour ajouter une classe dans un package, on met l'instruction **package** suivi du chemin du package dont chaque niveau est séparé par un **.**  
Le fichier doit être situé dans le répertoire `com / ipiecoles / java / java220` à partir du répertoire racine des sources.

Il est également possible d'importer des classes statiques, mais également des méthodes et des champs statiques avec l'instruction **import static**. Ici le champ statique `out` est accessible directement.

```
package com.ipiecoles.java.java220;
```

```
import java.util.Date;
```

L'import d'une classe se fait à l'aide de l'instruction **import** suivi du chemin de la classe. Ici, la classe `Date` peut être directement utilisée dans le programme.

```
import java.io.*;
```

Il est également possible d'importer toutes les classes d'un package avec le caractère **\***. Ici toutes les classe du package `java.io` sont accessibles dans le programme (comme la classe `File`). De manière générale, on évite d'utiliser cette notation pour des raisons de conflits éventuels (`import java.util.*; import java.sql.*;` lancera une erreur de compilation à l'utilisation de la classe `Date`).

```
import static java.lang.System.out;
```

```
public class Exemple {
```

```
    Date dateJava;
```

```
    java.sql.Date dateSql;
```

```
    File fichier;
```

```
    void bonjour(){
```

```
        out.print("Bonjour !");
```

```
    }
```

```
}
```

Les package permettent d'éviter les conflits d'unicité de nom de classe. Ici, on peut utiliser la classe `java.util.Date` et la classe `java.sql.Date` dans la même classe. Il est cependant nécessaire d'utiliser le nom canonique pour l'une des deux classes afin de les différencier dans le programme. A noter qu'il est possible d'utiliser le nom canonique d'une classe directement au lieu de l'importer. Cela fait cependant perdre le programme en lisibilité.

**i** Une pratique communément adoptée est de se baser sur le nom de domaine de l'entreprise (mais écrit dans le sens inverse) pour nommer les packages et d'utiliser des sous-packages pour les différents projets. Exemple `com.ipiecoles.java.java220`

# Les attributs

Les attributs (parfois appelés champs d'instances ou propriétés) représentent les données de la classe. C'est ce qui va caractériser une instance de classe.

Commande
Prochain Numéro
Numéro
Client
Priorité
Items
Est réglée ?
...

```
class Commmande {
    static long prochainNumero = 1;
    final Long numero;
    Client client;
    int priorite;
    HashMap<Integer, Item> items;
    Boolean estReglee = false;
}
```

Attribut `static` de type `Long` initialisé à la valeur `1L`. Un attribut `static` a la particulier d'être unique au niveau de la classe (pas au niveau de l'instance).

Attribut `final` de type `Long`. Par défaut s'il n'est pas défini explicitement, il prendra la valeur `null`. Une fois initialisé à la construction de l'objet, il ne sera plus possible de le modifier.

Attribut de type `Client` qui est une classe `maison`.

Attribut de type `int` (type primitif). Par défaut s'il n'est pas défini explicitement, il prendra la valeur `0`.

Agrégation : une commande possède une liste d'items.

Attribut de type `Boolean` avec une valeur par défaut à `false`.

Les attributs sont typés et peuvent avoir une valeur par défaut (qui sera affectée à chaque instance).



# Les méthodes

❗ Les méthodes représentent les *fonctions* ou *traitements* effectués sur les instances des classes. Elles sont généralement nommées avec un verbe d'action et manipulent les attributs de la classe.

Commande
Calcul d'un délai de livraison
Régler une commande
Ajouter un item
...

Méthode `static` retournant un `Integer` qui manipule le paramètre `priorite`. Elle peut être appelée directement à partir de la classe :  
`Commande.calculDelaiLivraison(2);` sans qu'il soit nécessaire d'initialiser un objet.

```
class Commande {
    static Integer calculDelaiLivraison(int priorite){
        //Livraison plus rapide en fonction de la priorité
        return priorite == 1 ? 3 : 5;
    }

    void reglerCommande(Double montant) {
        //Règlement de la commande
        this.montantRegle -= montant;
        this.client.metAJourSolde(montant);
    }

    void ajouterItem(Item item) {
        //Ajout d'une item dans la map
        this.items.put(item.getId(), item);
    }
}
```

Méthode qui permet de régler tout ou partie d'une commande prenant en paramètre le montant réglé. La méthode `metAJourSolde` est implémentée dans la classe `Client`.

Méthode qui permet d'ajouter une nouvelle commande à la map de commandes.

❗ Elles ont un type de retour et potentiellement des paramètres d'entrée (en dehors des attributs de la classe). Pour manipuler les attributs de la classe, on peut les préfixer par `this` pour éviter de les mélanger avec des paramètres d'entrée.



# Les constructeurs

❗ Pour instancier des objets d'une classe, il est nécessaire d'utiliser des **constructeurs**. Les constructeurs sont des méthodes comme les autres à l'exception de leur *signature* qui est un peu particulière, le type de retour et le nom de la méthode ne faisant qu'un.

```
Commande(){
}
```

Constructeur par défaut qui crée une instance sans initialiser aucun attribut (sauf ceux définis explicitement au niveau de la classe ainsi que les types primitifs)

```
Commande(Long numero, Client client) {
    this.numero = numero;
    this.client = client;
}
```

Constructeur prenant deux paramètres `numero` et `client` qui seront affectés aux attributs `numero` et `client` de l'instance créée. L'utilisation de `this` permet de distinguer les paramètres des attributs.

```
Commande(Long numero, Client client, int maPriorite) {
    this(numero, client);
    priorite = maPriorite;
}
```

Constructeur prenant trois paramètres et qui appelle le constructeur précédent avec l'instruction `this(numero, client);`. Il est possible d'utiliser directement le nom d'attribut si aucun paramètre n'éclipse l'attribut.

```
Commande c = new Commande();
```

Appel du constructeur par défaut. Les attributs *objets* ont `null` pour valeur, les types primitifs numériques valent `0` ou `false` pour les booléens.

```
c = new Commande(12345L, new Client());
```

Appel du deuxième constructeur. Les attributs `numero` et `client` sont initialisés avec les valeurs passées en paramètres. La variable `c` est une *référence* vers l'espace mémoire contenant l'objet effectif.

❗ On déclare généralement uniquement les constructeurs permettant d'instancier des objets cohérents. Exemple : Une commande doit obligatoirement posséder au minimum un numéro et un client. On ne déclare alors pas de constructeur par défaut et tous les constructeurs doivent au moins avoir ces deux paramètres.

# Visibilité

**i** La notion de **visibilité** est au cœur de la POO. Elle est la base du principe d'*encapsulation* et d'*interface*. En Java, il existe quatre *modificateurs de visibilité* permettant de définir la visibilité des méthodes et attributs au sein d'une classe. Ils sont spécifiés avant la déclaration des attributs ou des méthodes.

Modificateur	Effet	Utilisation
<code>private</code>	Visible uniquement par la classe	Attributs des classes, méthodes internes
<code>public</code>	Visible par toutes les classes	Constructeurs (!), getters et setters (pour les champs pouvant être manipulés), constantes, méthodes en lecture seule ou devant être utilisées en dehors de la classe.
<code>protected</code>	Visible par toutes les sous-classes et tout le package	<b>Sauf cas particulier, ne pas déclarer les attributs en <code>protected</code></b> . En revanche les méthodes protégées sont plus courantes.
(par défaut)	Visible par tout le package	<b>Éviter de ne pas spécifier la visibilité d'une méthode ou d'un attribut !</b>

**i** De manière générale, il est recommandé de restreindre autant que possible l'accès aux attributs et méthodes.



# Encapsulation

Grâce aux *modificateurs de visibilité*, il est possible de gérer l'**encapsulation** des classes. Cela revient à cacher les détails d'implémentation à tout ce qui est extérieur à la classe, et à exposer uniquement ce qui est nécessaire afin d'assurer l'intégrité des données.

```

public class Bonjour {
    public static final String BASE_MESSAGE = "bonjour";
    private String nom;
    private Date dateCourante;

    public Bonjour(String nom){
        this.nom = nom;
        this.dateCourante = new Date();
    }

    public void ditBonjour(){ out.println(formaterMessage()); }
    private String formaterMessage(){ return BASE_MESSAGE + " " + this.nom; }

    public String getNom() { return nom; }
    public void setNom(String nom) { this.nom = nom; }
    public Date getDateCourante() { return dateCourante; }
}
    
```

Cette classe est publique, donc visible par n'importe qui. Si on n'avait pas indiqué `public`, elle serait alors visible uniquement au niveau du package.

Cette constante est ici publique mais pourrait être également `private` si l'on juge qu'il n'est pas utile de l'exposer.

Les attributs sont privés et ne sont accessibles en lecture que par leur *getters*, et en écriture que par leur *setters*.

Sauf cas particuliers, un constructeur doit être `public` pour qu'il soit possible de créer des instances.

Cette méthode peut être appelée dans n'importe quel contexte. Cette classe peut être redéfinie dans les sous-classes.

Cette méthode ne peut être qu'appelée dans la classe. C'est un détail d'implémentation qu'il n'est pas nécessaire d'exposer. En le mettant `private`, cela empêche également les sous-classes de redéfinir son comportement.

Les *getters* et *setters* sont normalement `public`. Si l'on veut qu'un attribut ne puisse pas être mis à jour en dehors de la classe (comme la `dateCourante`) dans notre exemple, on n'implémente pas le *setter* (ou on le déclare `private`).

Un attribut ou une méthode peut être déclarée `static`. Dans ce cas, il est possible d'utiliser l'attribut ou d'appeler la méthode sans avoir d'instance de l'objet. Il faut cependant que l'attribut ou la méthode n'ait pas besoin d'accéder à des attributs non statiques.

# Héritage

**i** L'héritage est une notion centrale de la POO. Ce mécanisme permet de créer des classes à partir de classes existantes. Les attributs et méthodes de la *superclasse* sont transmis à la *sous-classe* (on dit qu'une sous-classe *hérite* d'une superclasse) qui peut alors définir d'autres attributs et méthodes ou redéfinir certains comportements (à condition que la *superclasse* l'autorise).

```
public class Client {
    private long numero;
    private double solde;
    private String adresseLivraison, adresseFacturation;

    public Client(long numero, double solde, String adresseLivraison, String adresseFacturation) {
        this.numero = numero; this.solde = solde;
        this.adresseLivraison = adresseLivraison;
        this.adresseFacturation = adresseFacturation;
    }
    public final double calculMontant(double montantHT){
        return montantHT * tauxTva();
    }
    protected double tauxTva(){
        return 1.2; //Par défaut on paye avec 20% de TVA
    }
}
```

Pour créer un objet de type `Client`, on appelle le constructeur :

```
Client c = new Client(1L, 2000d, "3 rue de la Paix", "3 rue de la Paix");
```

Lorsqu'on va appeler `c.calculMontantCommande(100)` cela va renvoyer  $100 * 1.2 = 120$ . Cette méthode est `final`, ce qui signifie qu'elle ne pourra pas être redéfinie dans d'éventuelles sous-classes.

Lorsqu'on va appeler `c.tauxTva()` cela va renvoyer 1.2

```
public class Professionnel extends Client {
    private String raisonSociale;
    private String siret;
```

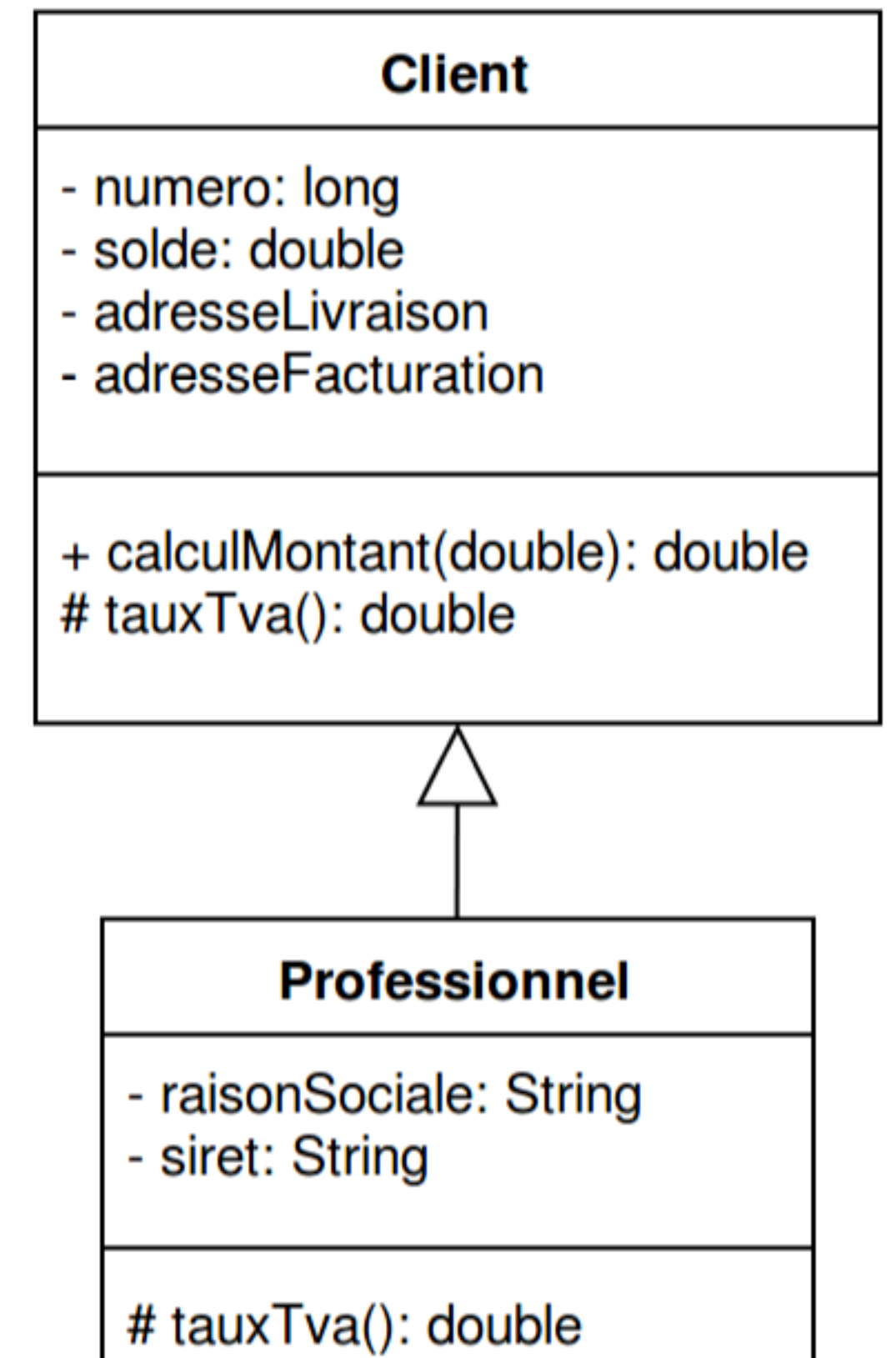
Les attributs `numero`, `solde`, `adresseLivraison` et `adresseFacturation` sont hérités de la superclasse

```
public Professionnel(long numero, double solde, String adresseLivraison, String adresseFacturation, String raisonSociale, String siret) {
    super(numero, solde, adresseLivraison, adresseFacturation);
    this.raisonSociale = raisonSociale; this.siret = siret;
}
protected double tauxTva(){
    return 1.0; //Les pros payent en HT
}
```

Il n'y a pas d'héritage de constructeurs, mais on peut appeler le constructeur de la superclasse (dans ce cas, cela doit être la première instruction du constructeur) avec `super`. Pour créer un objet de type `Professionnel`, on appelle le

La méthode `calculMontant` est héritée, la méthode `tauxTva` est redéfinie. Lorsqu'on va appeler `p.calculMontantCommande(100)` cela va renvoyer  $100 * 1.0 = 100$

```
Professionnel p = new Professionnel(1L, 2000d, "...", "...", "IPI", "1234");
```





# Hiérarchie d'héritage et classe **Object**

❗ L'héritage n'est pas un limité à un niveau. Ainsi, dans l'exemple précédent, on peut faire hériter une autre classe de **Professionnel**. En particulier, toutes les classes (même les vôtres héritent de la superclasse cosmique **Object**). En revanche, il n'est pas possible en Java, d'hériter de deux classes différentes.

Par défaut, la méthode `equals` compare les références des objets mais elle peut être redéfinie pour comparer en plus les valeurs de chaque attributs par exemple.

`equals` doit être *reflexive*, *symétrique*, *transitive* et *cohérente*

La méthode `hashCode` a pour but de donner une signature unique à un objet en fonction de la valeur de ses attributs sous forme d'entier. Si la méthode `equals` est redéfinie, et qu'on veut pouvoir stocker des instances dans une `HashMap`, cette méthode doit être également redéfinie

La méthode `toString` est intéressante pour retourner un équivalent textuel d'un objet, pour des besoins de journalisation par exemple, ou même des besoins métiers. La méthode par défaut renvoie

`NomClasse@hashcode(hex)`

Object

`equals`

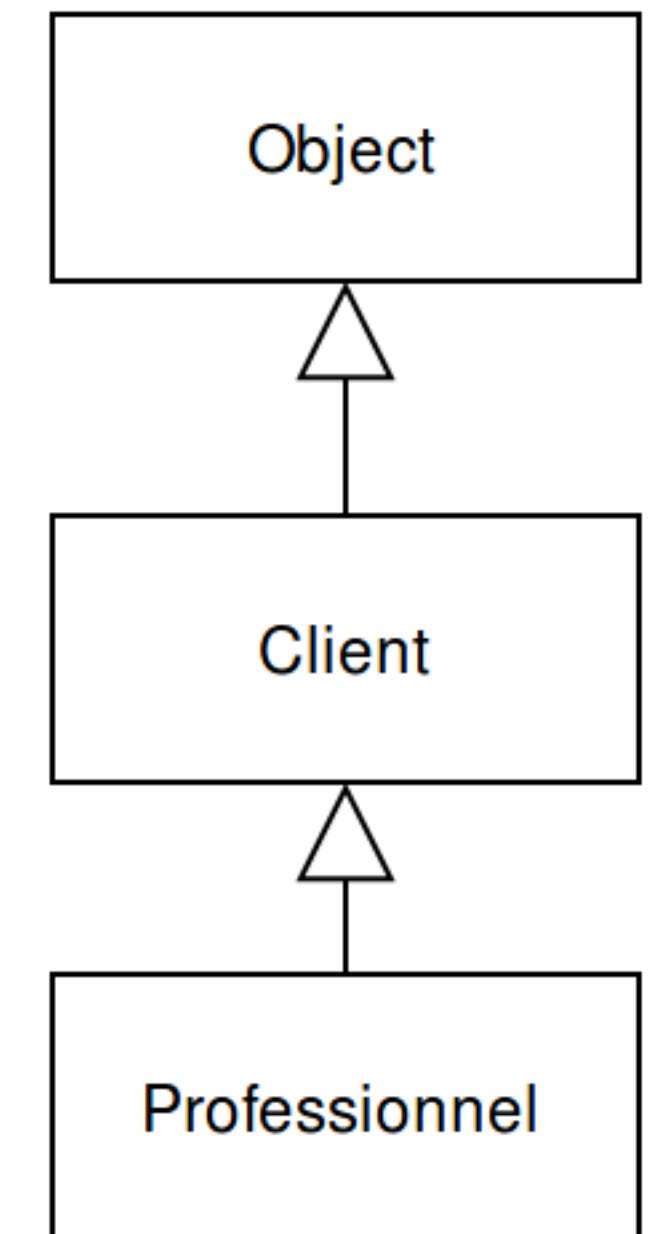
`hashCode`

`toString`

...

```
class Client {
    ...
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null) return false;
        if (getClass() != o.getClass()) return false;
        Client c = (Client) o;
        return numero == c.numero &&
            Double.compare(c.solde, solde) == 0 &&
            Objects.equals(adresseLivr, c.adresseLivr) &&
            Objects.equals(adresseFact, c.adresseFact);
    }

    public String toString() {
        return "Client{" +
            "numero=" + numero + ", solde=" + solde +
            ", adresseLivr=" + adresseLivr + '\n' +
            ", adresseFact=" + adresseFact + '\n' +
            '}';
    }
}
```



# Polymorphisme

❗ Pour savoir si l'héritage est le bon concept à implémenter entre deux classes, il suffit de se poser la question *est*. Si tous les objets de la sous-classe *sont* des objets de la superclasse, alors l'héritage est le bon concept. Une instance d'une sous-classe est donc également une instance de la superclasse (l'inverse est faux). On appelle ce mécanisme, le **polymorphisme**.

```
Boolean estClient, estProfessionnel, estObject;
Client c; Professionnel p;

c = new Client();
estClient = c instanceof Client; //true
estObject = c instanceof Object; //true (toujours...)

c = new Professionnel();
estClient = c instanceof Client; //true
estProfessionnel = c instanceof Professionnel; //true

p = new Professionnel();
estClient = p instanceof Client; //true
estProfessionnel = p instanceof Professionnel; //true

p = new Client(); //Erreur de compilation
```

❗ Un des gros intérêts du **polymorphisme** est de pouvoir traiter un objet comme instance de sa classe, ou de n'importe quelle superclasse (y compris `Object`).



# Exemple d'usage du polymorphisme

❗ Il est possible de stocker des collections d'objets du type d'une superclasse est d'y stocker des objets de la superclasse et de n'importe quelle sous-classe.

```
Client c = new Client();
Professionnel p = new Professionnel();
Client[] clients = new Client[2];
Professionnel[] pros = new Professionnel[1];
```

On ne peut stocker que de objets de la classe `Professionnel` (et ses éventuelles sous-classes) dans un `Professionnel[]`

Un `Professionnel` étant un `Client`, il est possible de transformer un tableau de référence de sous-classe en tableau de référence de superclasse. Cependant, le tableau se *souvient* du type de base utilisé lors de sa création. Lorsqu'on essaye de mettre un `Client` dans le tableau, la compilation ne lève pas d'erreurs mais une exception `ArrayStoreException` est levée à l'exécution.

On peut transtyper explicitement un objet lorsqu'on connaît son type (on peut également le tester en utilisant `instanceof`). Cela permet de retrouver la classe de base est d'exécuter les méthodes propres à la classe. Lorsqu'un transtypage échoue, une exception `ClassCastException` est levée;

```
clients[0] = c; clients[1] = p;
pros[0] = p;
```

On peut stocker un `Client` ou un `Professionnel` dans un `Client[]`

```
boolean txTva = p.tauxTva() == clients[1].tauxTva(); //true
Client[] clientsPros = pros; //OK
clientsPros[0] = new Client(); //Erreur à l'exécution
```

Même s'il est stocké en tant que `Client` dans le tableau, l'instance se comporte bien comme un `Professionnel`

```
ArrayList<Object> objects = new ArrayList();
objects.add(c); objects.add(p);
objects.add(new Integer(5));
objects.get(1).tauxTva() //Erreur de compilation
```

Un `ArrayList` d'`Object` peut contenir n'importe quel objet ! Il n'est d'ailleurs pas utile de préciser que l'`ArrayList` contient des `Object` : `ArrayList objects = new ArrayList();`. En revanche, ils se comportent comme des `Object`

```
(Client)objects.get(1).tauxTva(); //OK
(Professionnel)objects.get(1).tauxTva(); //OK
(Professionnel)objects.get(0).tauxTva(); //Erreur à l'exécution
```



# Surcharge et redéfinition

- La **surcharge** permet de définir plusieurs méthodes avec le même nom, mais dont la *signature* (paramètres et/ou type de retour) est différente.
- La **redéfinition** est le fait de changer le code d'une méthode issue de la superclasse.

```
public class Client {
  //Surchages
  public void ajouterItem(Item item) {
    ...
  }
  public void ajouterItem(String nom, int quantite){
    ...
  }

  //Redéfinition
  public String toString(){
    super.toString() + ...;
  }
}
```

La redéfinition remplace le code de la superclasse. On peut cependant l'appeler explicitement si nécessaire avec `super /`

- La **résolution de surcharge** est fait par le compilateur qui choisit quelle fonction est appelée en fonction des paramètres et du type de retour.
- La résolution d'une **méthode redéfinie** est faite à l'exécution par la JVM qui sélectionne la méthode à exécuter en fonction du type réel de l'objet.

# Classes abstraites

Il arrive que les superclasses deviennent de plus en plus abstraites et servent de *moule* pour les sous-classes permettant de mettre en commun des attributs et des méthodes.

```
public abstract class Client {
    private long numero;
    public Client(long numero) {
        this.numero = numero;
    }
    public double calculMontant(double montantHT){
        return montantHT * tauxTva();
    }
    public abstract double tauxTva();
}
```

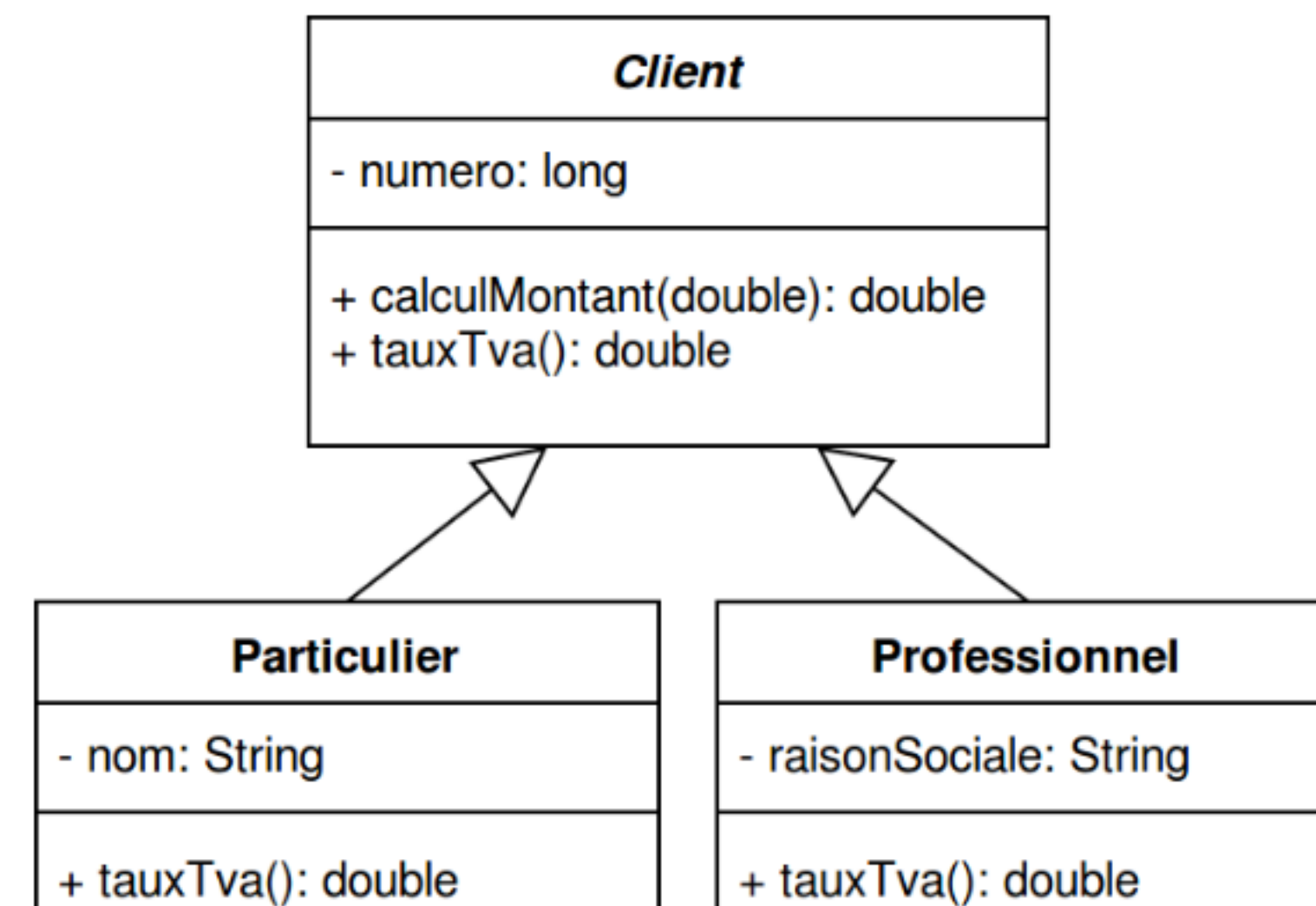
La classe `Client` devient `abstract`. Ainsi, il ne devient plus possible d'appeler le constructeur de `Client` ailleurs que dans les constructeurs des sous-classes (à l'aide de `super`). On peut également déclarer `abstract` une méthode. Dans ce cas, elle n'a pas d'implémentation dans la classe `abstract`, par contre les sous-classes sont **obligées** de redéfinir cette méthode.

```
public class Professionnel extends Client {
    private String raisonSociale;
    public Professionnel(long numero, String raisonSociale) {
        super(numero);
        this.raisonSociale = raisonSociale;
    }
    public double tauxTva(){
        return 1.0; //Les pros payent en HT
    }
}
```

Dans le constructeur, on fait référence au constructeur de la superclasse avec `super`. La méthode `calculMontant` est héritée normalement.

```
public class Particulier extends Client {
    private String nom;
    public Particulier(long numero, String nom) {
        super(numero);
        this.nom = nom;
    }
    public double tauxTva(){
        return 1.2; //Les particuliers payent en TT
    }
}
```

Chaque sous-classe doit implémenter la méthode `tauxTva`



Un `Client` est soit un `Professionnel`, soit un `Particulier`, cela n'a pas vraiment de sens d'avoir une instance de type `Client`. On peut donc la rendre **abstraite**. Il ne sera possible alors que d'instancier une sous-classe. On peut continuer à utiliser le type `Client` dans des listes ou des tableaux



# Bonnes pratiques

**i** Quelques **bonnes pratiques** à suivre lorsqu'on utilise les mécanisme d'héritage.

Mettre les attributs et méthodes communs dans la superclasse.

Ne pas utiliser `protected` sur les attributs et bien réfléchir si c'est utile lors de l'utilisation sur une méthode.

Utiliser l'héritage que lorsqu'il y a une relation d'appartenance *est*.

Ne pas utiliser l'héritage à moins que toutes les méthodes et attributs aient du sens pour les sous-classes.

Ne pas modifier le comportement attendu d'une méthode lors de sa redéfinition

Utiliser le polymorphisme à la place de tests `instanceof`

Lorsqu'une sous-classe est créée et qu'une méthode accède directement aux attributs, cela casse le principe d'encapsulation. De plus, toutes les classes issues du même package auront accès à ces champs `protected` qu'elles soient ou non des sous-classes

Il peut être tentant d'utiliser l'héritage lorsqu'il y a des similitudes entre classes. Par exemple, faire hériter une classe `Prestataire` de `Professionnel` puisqu'ils ont tous les deux un numéro, une raison sociale... Cependant, un `Prestataire` n'est pas un `Client`, il sera donc nécessaire de définir ou redéfinir des méthodes et il en résultera plus de code que si l'héritage n'avait pas été utilisé.

Créer une classe `Achat` qui hérite de `Commande` (un achat d'un fournisseur est une commande) n'est pas une bonne idée car la méthode `calculDelaiLivraison` sera héritée et elle n'a pas de sens dans le contexte d'un achat fournisseur.

Par exemple, redéfinir `tauxTva` dans la classe `Professionnel` pour renvoyer une exception précisant qu'un `Professionnel` ne paye pas de TVA est une mauvaise pratique, puisque cela fera échouer d'autres méthodes (ici `calculMontant`).

Si vous vous retrouvez à faire

```
if(object instanceof Client){
    methode1();
} else if (object instanceof Professionnel) {
    methode2();
}
```

Il y a de grandes chances que le polymorphisme puisse être utilisé.



# Classes `final`

📘 Une classe peut être déclarée `final` afin d'empêcher toute autre classe d'hériter de celle-ci. Les méthodes d'une classe `final` sont automatiquement `final` contrairement aux attributs.

```
final class CommandeSpeciale extends Commande {  
    ...  
}  
  
public class CommandePlusSpeciale extends CommandeSpeciale { //Erreur de compilation  
}
```

📘 La classe `String` est `final`. Il n'est pas possible d'en hériter. Cela permet notamment de s'assurer qu'une référence à `String` correspond bien au type de base. Ce mécanisme n'est à utiliser que s'il est vraiment nécessaire.

# Classe générique

❗ La *programmation générique* consiste à écrire du code qui puisse être utilisé par des types différents. Une **classe générique** est une classe paramétrée par une ou plusieurs *variables de type*.

```
public class Paire<T, U> {
    private T premier;
    private U second;

    public Paire(T premier, U second) {
        this.premier = premier;
        this.second = second;
    }

    public U getSecond() { return second; }
    public T getPremier() { return premier; }
    public void setSecond(U second) { this.second = second; }
    public void setPremier(T premier) { this.premier = premier; }
}
```

La classe `Paire` introduit deux variables de type qui sont `T` et

`T` et `U` sont remplacés respectivement par `String` et `Integer`. La deuxième instruction utilise l'inférence de type pour alléger la syntaxe.

Dans cet exemple, on n'utilise pas `<>`. C'est donc le *type brut* qui est utilisé : `Object`, `Object`, il en résulte un *warning* à la compilation. Cependant, comme la variable `p2` explicite clairement les types, la deuxième instruction échoue à la compilation.

Les deux instructions sont valides (bien que générant un *warning*), mais la deuxième instruction lèvera une exception à l'exécution puisqu'il y aura une tentative de transtypage de `Boolean` vers `Integer`.

T et U sont ensuite utilisés dans la classe pour spécifier les types des attributs ou des variables locales, et des types de retour des méthodes.

```
Paire<String, Integer> p = new Paire<String, Integer>("Un", 1);
Paire<String, Integer> p = new Paire<>("Un", 1); //Java 7+
String s = p.getPremier();
p.setSecond(1);
```

```
Paire<String, Integer> p2 = new Paire("Un", 1); //Warning
p2.setPremier(1); //Erreur de compilation
```

```
Paire<String, Integer> p2 = new Paire(5, false); //Warning
Integer i = p2.getSecond(); //ClassCastException
```

```
Paire p3 = new Paire("Un", 1); //Warning
p3.setPremier(false); //Warning
```

Enfin dans ce cas, c'est également le *type brut* qui est utilisé sauf qu'aucun type n'est explicite dans la déclaration. On peut donc faire tout et n'importe quoi...

❗ C'est ce principe qui est utilisé dans les collections comme `ArrayList<T>`, `HashSet<T>` ou `HashMap<K, V>`.



# Méthodes génériques

📘 Il est aussi possible de déclarer une **méthode générique** dans une classe classique.

```
public class ArrayIpi {
    public static <T> T getMiddle(T[] a)
    {
        return a[a.length / 2];
    }
}
```

<> indique que la méthode est générique. Comme pour les classes, il est possible de spécifier plusieurs variables de types. Ensuite, le type `T` peut être utilisé comme type de variable ou de retour.

```
public static <T> T min(T[] a) {
    if(a == null || a.length == 0) {
        return null;
    }
    T min = a[0];
    for (int i = 1; i < a.length; i++)
        if(min.compareTo(a[i]) > 0){
            min = a[i];
        }
    return min;
}
```

Grâce à cette notation, on ne pourra appeler cette méthode que si `T` étend `Comparable`. C'est le cas de `String`.

Ici, `compareTo` est une méthode de `Comparable`. Pour que l'on puisse l'appeler, il faudrait que `T` soit un type qui étend `Comparable` ...

Les deux instructions sont équivalentes. En effet, dans la deuxième instruction, le type `String` est inféré à partir du type de la variable `milieu`.

```
String[] noms = {"Jean", "Pierre", "François"};
String milieu = ArrayIpi.<String>getMiddle(noms);
String milieu = ArrayIpi.getMiddle(noms); //François
```

Comme `noms` est un tableau de `String`, cette méthode sait que `T` correspond au type `String`. Il appelle donc la méthode `compareTo` de la classe `String` pour sortir le minimum du tableau.

📘 Lorsqu'on veut qu'une variable de type *étende* plusieurs types, on utilise la notation `T extends Comparable & Serializable`



# Interfaces

📘 Lorsqu'une classe abstraite ne possède aucun attribut (en dehors de constantes) et aucune méthode non statique, on peut la remplacer par une **interface**.

Déclaration d'une interface avec le mot-clé `interface`. Il ne peut y avoir que des méthodes abstraites et des constantes

```
public interface Client {
    public static final double PRIX_BASE = 1.0;
    public abstract double tauxTva();
    public default void affichePrixBase(){
        System.out.println("Prix de base : " + PRIX_BASE)
    }
}
```

Avec Java 8, il est possible de mettre déclarer des méthodes avec des implémentations par défaut. Lorsque c'est le cas, les classes implémentant l'interface ne sont pas obligées de définir leur propre implémentation.

```
public class Particulier implements Client {
    public double tauxTva(){
        return PRIX_BASE + 0.2;
    }
}
```

La classe `Particulier` implémente l'interface `Client`. Il est obligatoire de définir la méthode `tauxTva`

```
Client client = new Particulier();//OK
Client client2 = new Client();//Erreur de compilation
```

📘 Il est possible de déclarer des variables dont le type est une **interface**.

📘 Une classe peut implémenter **plusieurs interfaces**.

📘 Une **interface** peut *hériter* d'une autre **interface**.

# Implémentations multiples

📘 L'intérêt principal des interfaces est l'**implémentation multiple** (alors que l'héritage est simple). Une classe peut ainsi implémenter plusieurs interfaces.

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

```
public class Particulier implements Client, Comparable<Client> {  
    private double solde;  
    public Particulier(double solde) { this.solde = solde; }  
    public int compareTo(Client o) {  
        return Double.compare(solde, o.solde);  
    }  
    public double tauxTva(){  
        return 1.2;  
    }  
}
```

Implémenter l'interface `Comparable`  
demande de définir la méthode  
`compareTo`.

Implémenter l'interface `Client`  
demande de définir la méthode  
`tauxTva`.

```
Particulier[] clients = new Particulier[3];  
clients[0] = new Particulier(3500.0); //Client 1  
clients[1] = new Particulier(6500.0); //Client 2  
clients[2] = new Particulier(5500.0);  
Arrays.sort(clients); //Client 1, 3, 2
```

La classe `Arrays` possède une méthode `sort` qui permet de trier un tableau d'éléments implémentant l'interface `Comparable`. Le tri fait appel à la méthode `compareTo` de la classe.

📘 De nombreuses interfaces *standard* existent, comme ici `Comparable`



# Classes internes

Une **classe interne** est une classe qui est définie à l'intérieur d'une classe. Ce mécanisme est intéressant notamment dans le cadre de la *programmation événementielle*. Voici quelques avantages à l'utilisation des **classes internes** :

Elles ne sont pas visibles des autres classes du package de la classe externe.

Les méthodes des classes internes peuvent accéder aux attributs et méthodes de la classe interne (même s'ils sont `private`).

C'est ce mécanisme qui est utilisé dans les *classes anonymes*

Lors de l'instanciation d'un objet d'une classe interne, l'objet de la classe externe qui lui a *donné naissance* lui est systématiquement associé.

⚠ La définition d'une classe interne n'implique pas automatiquement d'attribut de ce type dans la classe externe. Il faut déclarer explicitement un attribut lorsqu'on en a besoin.

⚠ Les classes internes ont une syntaxe complexe et ne doit être utilisé que lorsqu'il est nécessaire et maîtrisé.

```

public class Commande {
    private List<Item> items;
    private double tva = 1.2;
    public void ajouterItem(String nm, int qte, double px) {
        Item item = new Item();
        item.nom = nm;
        item.qte = qte;
        item.prix = px;
        items.add(item);
    }
    private class Item {
        private int qte;
        private double prix;
        private String nom;
        public double prixItem(){ return qte * prix * tva; }
    }
    public double prixTotal(){
        return items.stream().map(Item::getPrixLig)
            .reduce(0d, Double::sum);
    }
}
...
Commande commande = new Commande();
Commande.Item item = commande.new Item();
  
```

On peut déclarer un attribut utilisant le type de la classe interne `Item`

La classe `Item` et ses attributs sont accessibles directement.

La classe `Item` est privée. Elle n'est pas visible à l'extérieur de la classe `Commande`. La seule manière de créer une instance de `Item` est ici dans la méthode `ajouterItem`.

On peut accéder à l'attribut `tva` de la classe externe `Commande`.

La méthode `prixItem` est également accessible depuis la classe `Commande`.



# Classes internes locales

📘 On peut déclarer une **classe interne locale** à une méthode. L'instanciation de cette **classe** n'est alors possible que dans la méthode en question.

```
public class Item {
    private String nom;
    private int quantite;
    private double prix;

    public void afficheItem(){
        final String AFFICHAGE = "Affichage de l'item : ";
        boolean accessibleAfficheur = false;
        class Afficheur {
            private String texte;
            public Afficheur() {
                this.texte = AFFICHAGE + nom + ", qte:" + quantite + ", prix:" + prix;
            }
            public void affiche(){ System.out.println(texte); }
        }
    }
}

new Item("Produit", 5, 12.0).afficheItem();
//"Affichage de l'item : Produit, qte: 5, prix: 12.0"
```

Cette variable n'est pas accessible dans la classe `Afficheur` car elle n'est pas `final`.

Cette variable est accessible dans `Afficheur` car elle est `final`.

Accès possible aux attributs de la classe externe `Item`.

📘 En plus des accès classiques à la classe externe, la **classe interne locale** a également accès à toutes les variables locales `final` de la méthode.

# Les classes internes anonymes

📘 Il est possible de déclarer ponctuellement une classe, sans lui donner de nom lorsque son usage est très ponctuel, on parle de **classe interne anonyme**. Ce mécanisme est beaucoup utilisé en *programmation événementielle*.

```
public class Particulier implements Client, Comparable<Client> {  
  
    private double solde;  
  
    public Particulier(double solde) { this.solde = solde; }  
  
    public int compareTo(Client o) {  
        Comparator<Particulier> comparator = new Comparator<Particulier>() {  
            public int compare(Particulier p1, Particulier p2) {  
                return p1.solde.compareTo(p2.solde) ;  
            }  
        };  
        return comparator.compare(this, o) ;  
    }  
}
```

Le constructeur par défaut de `Comparator` est utilisé et l'utilisation des `{ }` permet de redéfinir directement la méthode `compare` de `Comparator`. A noter le `;` obligatoire après l'appel à `new`.

📘 Cette notation permet de se rapprocher du paradigme de *programmation fonctionnelle*.

# Les lambdas

**i** Utilisables à partir de Java 8, les **lambdas** sont une nouvelle façon d'écrire du code, de manière plus concise.

```
Comparator<Particulier> comparator = new Comparator<Particulier>() {  
    public int compare(Particulier p1, Particulier p2) {  
        return p1.getSolde().compareTo(p2.getSolde());  
    }  
};
```

```
Comparator<Particulier> comparator = (Particulier p1, Particulier p2) -> o1.getSolde().compareTo(o2.getSolde());
```

```
Comparator<Particulier> comparator = (Particulier p1, Particulier p2) -> {  
    Double s1 = o1.getSolde(); Double s2 = o2.getSolde();  
    return s1.compareTo(s2);  
});
```

**i** **Comparator** ne définit qu'une seule méthode, on parle d'**interface fonctionnelle**. La JVM sait donc que c'est la méthode **compareTo** qui est redéfinie. Plus concis, les **lambdas** sont cependant moins pratique à *debugger*.



# Les références de méthodes

📌 Les **références de méthodes** sont une notation introduite également par les **lambdas** qui permettent de simplifier encore plus les *classes anonymes*.

```
public class Particulier {
    private Long numero;
    private Double solde;
    public static int compareSolde(Particulier a, Particulier b) {
        return a.solde.compareTo(b.solde);
    }
}
...

```

La référence de la méthode statique `compareSolde` de la classe `Particulier` est utilisée pour chaque élément du tableau `clients`.

La référence de la méthode `compareSolde` de l'instance `comp` de la classe `ComparaisonClient` est utilisée pour chaque élément du tableau `clients`.

```
public class ComparaisonClient {
    public int compareSolde(Particulier a, Particulier b) {
        return a.getSolde().compareTo(b.getSolde());
    }
}
...

```

La référence de la méthode `compareToIgnoreCase` de `String` est utilisée pour chaque élément du tableau `myArray`.

```
public interface ParticulierFactory {
    public abstract Particulier getParticulier(Long numero, Double solde);
}

```

La référence de constructeur de `Particulier` est affectée à une instance de `ParticulierFactory` appelé `partFactory`. Cela est possible car la signature du constructeur de `Particulier` est la même que la méthode `getParticulier` de `ParticulierFactory` : `(Long, Double) -> Particulier`. Ensuite, la méthode `getParticulier` de `partFactory` est appelée avec le numéro 123 et le solde 250.0, ce qui appelle le constructeur de `Particulier`. Une nouvelle instance de `Particulier` est créée.

```
Particulier[] clients = { new Particulier(1L, 1200.0), ... };
//Référence à une méthode statique
Arrays.sort(particuliers, Particulier::compareSolde);

//Référence d'une méthode d'instance d'un objet spécifique
ComparaisonClient comp = new ComparaisonClient();
Arrays.sort(clients, comp::compareSolde);

//Référence de la méthode d'instance compareToIgnoreCase
//d'un objet spécifique du type particulier String
String[] myArray = {"one", "two", "three", "four"};
Arrays.sort(myArray, String::compareToIgnoreCase);

```

```
//Référence à un constructeur
ParticulierFactory partFactory = Particulier::new;
Particulier p = partFactory.getParticulier(123L, 250.0);

```

📌 La notation `::` peut faire référence à une méthode statique, une méthode d'instance d'un objet spécifique, une méthode d'instance d'un objet spécifique d'un type particulier, ou d'un constructeur.

# Les Streams

📌 Cette API est disponible à partir de Java 8 et permet de rendre plus aisée et concise la gestion des **Collections**. Elle s'appuie largement sur les **lambda**.

```
List<String> noms = Arrays.asList("pierre", "jean", "jeremy", "marc", "jennifer");
```

Passage d'une collection **List** à un **Stream** avec `stream()`.

```
noms.stream()
```

```
.filter(x -> x.contains("je"))
```

Filtrage uniquement des valeurs contenant la chaîne de caractères "je"

```
.map(x -> x.substring(0, 1).toUpperCase() + x.substring(1))
```

On met en majuscule la première lettre de chaque élément avec `map` qui permet de modifier chaque valeur.

```
.sorted()
```

On trie par ordre alphabétique avec `sorted` qui retourne la liste des éléments triés selon l'ordre naturel.

```
// Affichage :
```

```
// Jean
```

```
// Jennifer
```

```
// Jeremy
```

```
.forEach( System.out::println );
```

On affiche chaque élément avec `forEach` qui permet d'effectuer un traitement pour chaque élément.

```
List<Commande> commandes = ... ;
```

On passe d'une liste de **Commande** à une liste de **Client**

```
Stream<Client> clients = commandes.stream()
```

```
.map( c -> c.getClient() )
```

```
.collect( Collectors.toList() );
```

On passe d'un **Stream** à une **List** avec `collect` et `Collectors.toList()`