

# Programmation Orientée Objet (POO)

---



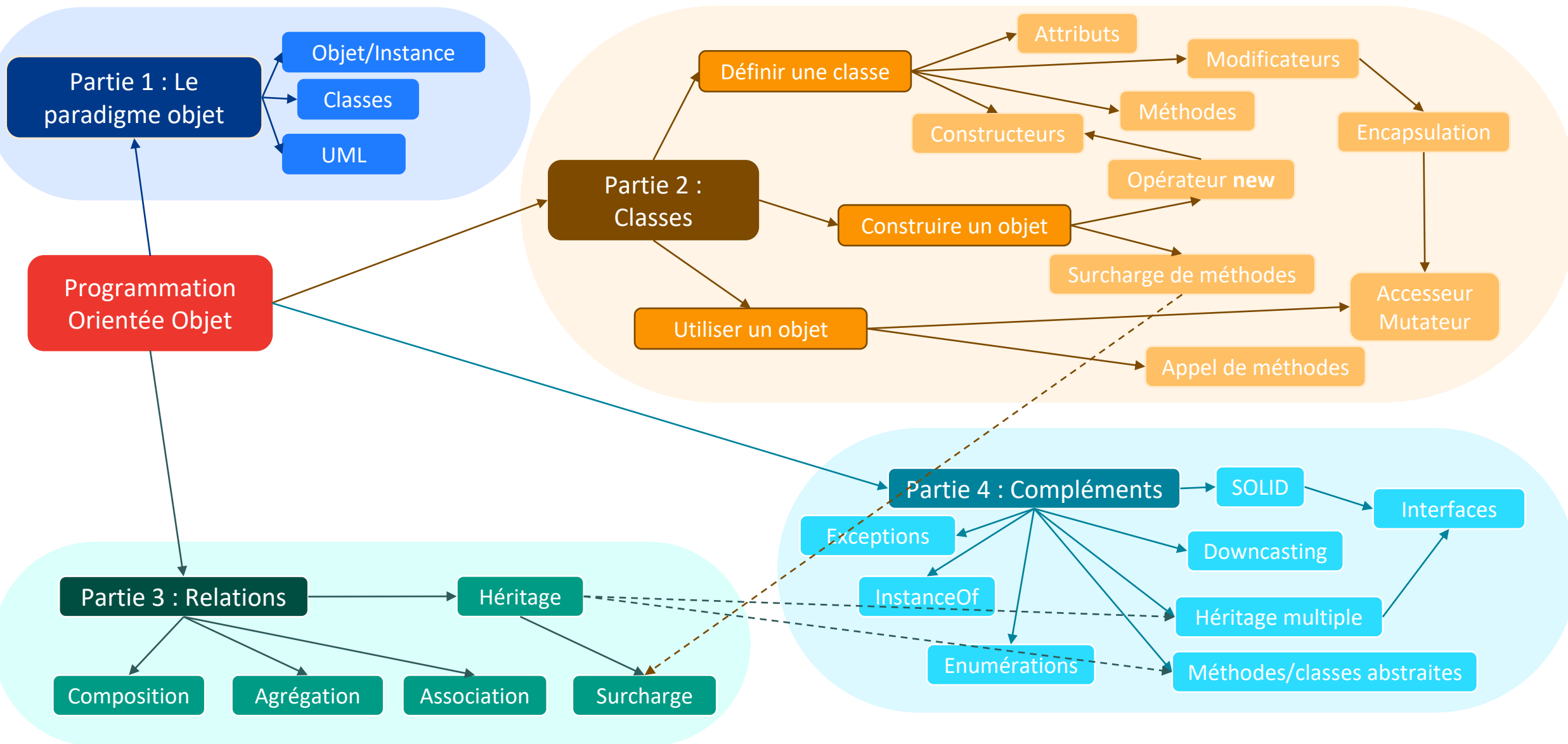
# Introduction : le paradigme objet

Une Classe ???

Un Objet ???

A quoi sert la programmation orientée objet ?

# Plan du cours







01

# Le paradigme objet

# Introduction : Les grands types de programmation

## Programmation procédurale

Découpage du code en procédures (fonction), en partant de l'opération complexe et en découpant jusqu'à l'opération élémentaire.

- Structure simple
- Peu de concepts à apprendre
- Exécution efficace
- Ne permet pas de traiter les problèmes complexes
- Difficilement maintenable

## Programmation fonctionnelle

Comme la programmation procédurale, essentiellement basée sur les fonctions, mais avec la contrainte de n'utiliser que des fonctions pures (immutabilité des données, Ex : sum, map, reduce).

- Elimination des effets de bord
- Adapté aux données IA et Big Data
- Plus facile à paralléliser
- Ne permet pas de traiter les problèmes complexes
- Contraintes rigides
- Difficulté à traiter des données dynamiques

## Programmation orientée objet

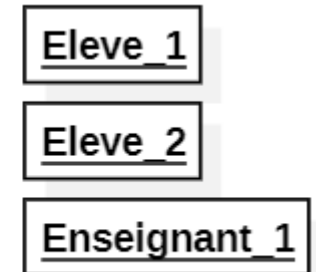
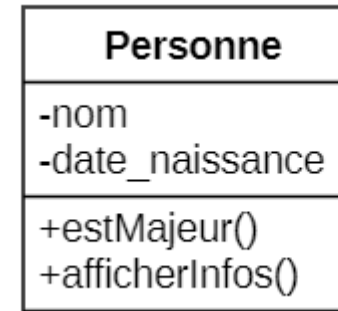
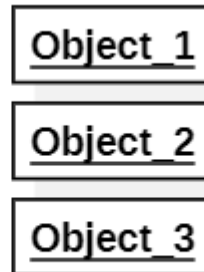
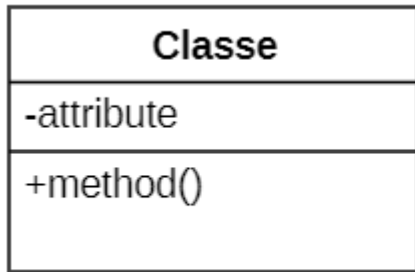
Se base sur le paradigme objet. Intègre les services/fonctions au sein des objets (méthodes). Permet de définir des comportements de façon compartimentée.

- Convient aux problèmes complexes
- Modulable
- Evolutif
- Favorise l'abstraction
- Sollicite des concepts plus abstraits
- Structure lourde, moins adaptée pour des programmes simples
- Verbosité

**Delaporte, X.** (2024). *Esthétique du code*. Dans **Le code a changé** [Podcast]. France Inter.

**Azoulai, N.** (2024). *Python*. P.O.L.

# Le paradigme objet



Une classe représente une abstraction, un modèle, une représentation générique des objets du monde réel.

Elle comporte :

- Un **nom** qui permet de nommer les objets représentés par la classe.
- Des **attributs** qui permettent de conférer des propriétés aux classes (ex : couleur, âge, localisation).
- Des **méthodes** qui sont des fonctions définies pour la classe, qui permettent d'animer les objets instanciés.

Un **objet** (ou **instance de classe**) est relié à **une** classe (et **une seule** !) mais une classe peut être utilisée pour créer **plusieurs** objets.

Pour faire un parallèle avec les variables de type primitif (int, double, float, boolean) :

Type  
↑ a pour type  
Variable

int  
↑ a pour type  
indice\_boucle

Classe  
↑ a pour classe  
Objet

Personne  
↑ a pour classe  
Eleve1



**Dark Oak Logs**  
x 522



**Stone Bricks**  
x 136



**Glass Pane**  
x 35



**Spruce Slab**  
x 259



**Spruce Stairs**  
x 127



**Stone Brick Wall**  
x 25



**Stripped Birch Log**  
x 245



**Oak Leaves**  
x 115



**Dirt**  
x 16



**Spruce Planks**  
x 182



**Spruce Trap Door**  
x 109



**Barrel**  
x 13



**Dark Oak Fence**  
x 167



**Lantern**  
x 54



**Torch**  
x 10



**Stone Brick Stairs**  
x 140



**Stone Brick Slab**  
x 43



**Campfire**  
x 6

<https://minecraft.wonderhowto.com/forum/build-house-minecraft-0224754/>









<https://simscommunity.info/2015/12/07/the-sims-4-get-together-build-items-overview/>

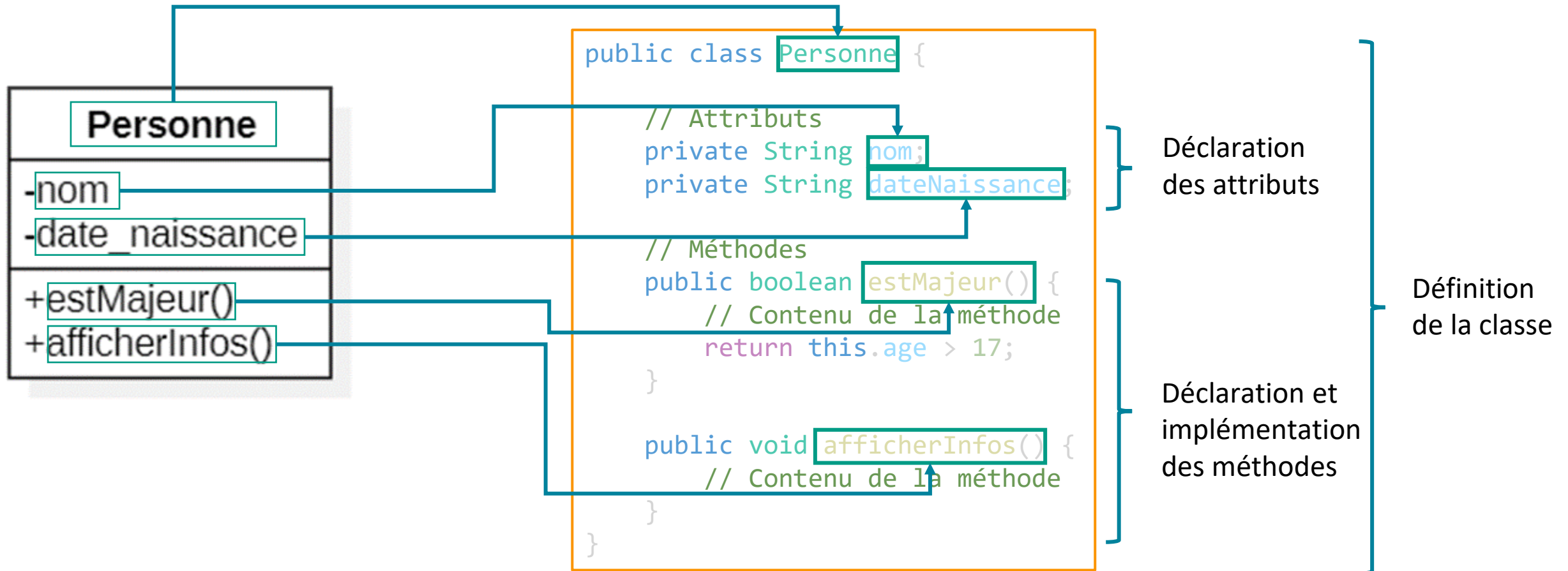




## 02

# Classes, objets, attributs et méthodes

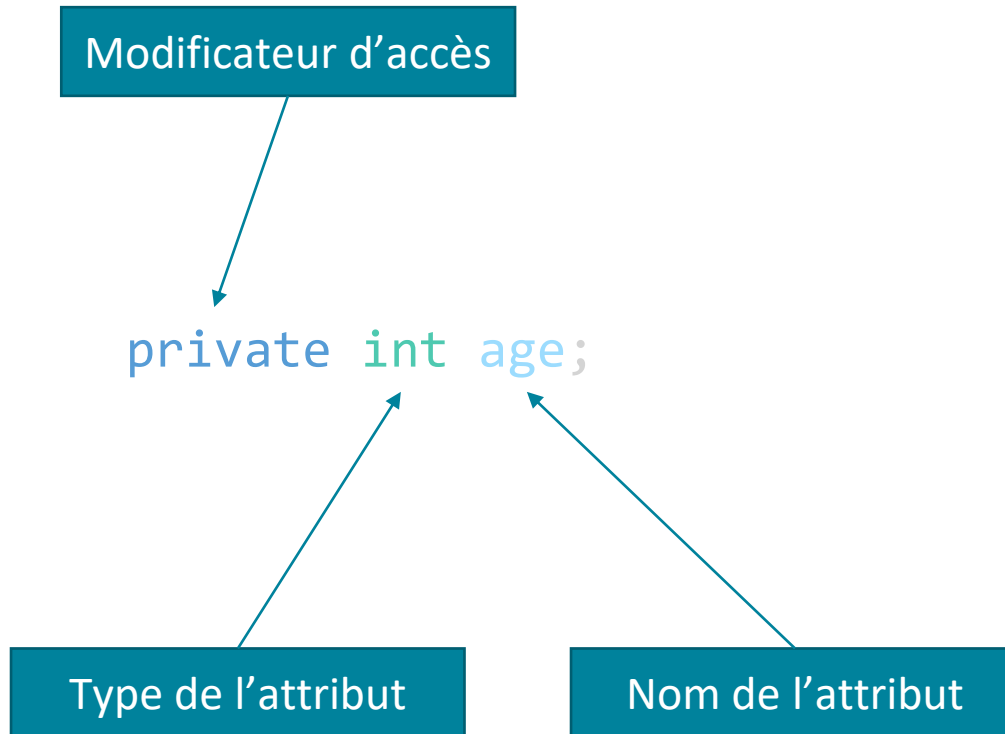
# Définition d'une classe (attributs et méthodes)



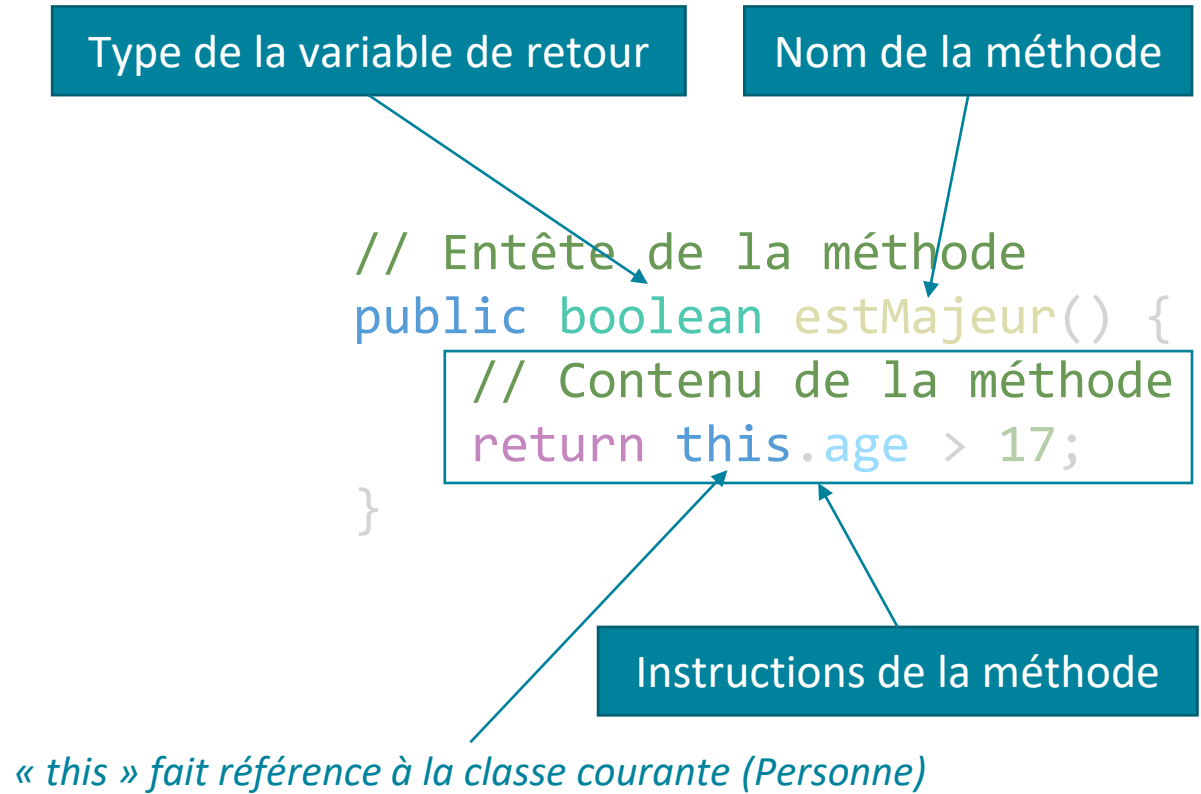


# Définition d'une classe (attributs et méthodes)

## Déclarer un attribut



## Déclarer et implémenter une méthode



# Définition d'une classe (modificateurs d'accès)

Lorsqu'on déclare des attributs ou des méthodes, on leur associe en général un **modificateur d'accès**, qui indique d'où sont visibles/accessibles les méthodes et attributs. Il existe trois modificateurs d'accès :

- **public** : visible depuis n'importe quel package, et n'importe quelle classe;
- **protected** : visible par la classe courante et par les classes héritées et les classes du package;
- **private** : visible uniquement par la classe courante.

```
public class Personne {  
  
    // Attributs  
    private String nom;  
    private String dateNaissance;  
  
    // Méthodes  
    public boolean estMajeur() {  
        // Contenu de la méthode  
        return this.age > 17;  
    }  
  
    public void afficherInfos() {  
        // Contenu de la méthode  
    }  
}
```

En l'absence de modificateur les éléments sont accessibles uniquement par la classe courante et les classes du même package.

On déclare généralement en **private** les attributs dont on veut contrôler l'accès et la modification, et les méthodes qui ne sont utilisées qu'à l'intérieur de la classe.

Pour attribuer une classe à un package, on ajoute l'instruction suivante en tête de fichier :  
**package** nom\_package;

# Définition d'une classe - Encapsulation (Accesseur / Getters)

Même si des attributs sont définis en `private`, on doit pouvoir lire leur valeur depuis l'extérieur de la classe. Pour cela, on définit des `accesseurs`, qui permettent de récupérer la valeur d'un attribut, même si celui-ci est déclaré `private`.

On nomme par convention les accesseurs de la façon suivante : `getNomattribut`.

Le type de retour d'un `accesseur` est forcément celui de l'attribut concerné.

Dans certains cas, comme dans l'exemple ci-contre, les `accesseurs` permettent de vérifier que les droits d'accès sont vérifiés avant de retourner la valeur de l'attribut.

```
public class Personne {  
  
    // Attributs  
    private String nom;  
    private int age;  
  
    // Accesseurs-Getters  
    public int getAge(){  
        return this.age;  
    }  
  
    public int getAge(User user){  
  
        if (verifierAcces(user, "lecture")) {  
            return this.age;  
        } else {  
            System.out.println("Accès refusé");  
            return -1;  
        }  
    }  
}
```



# Définition d'une classe - Encapsulation (Mutateurs / Setters)

De la même manière, un attribut défini en `private` doit tout de même pouvoir être modifié depuis l'extérieur de la classe. Pour cela, on définit des **mutateurs** qui prennent en paramètre une valeur du type de l'attribut concerné, et qui l'affecte à cet attribut. En revanche, comme il ne fait qu'affecter la valeur, **un mutateur ne retourne aucune valeur** (`void`).

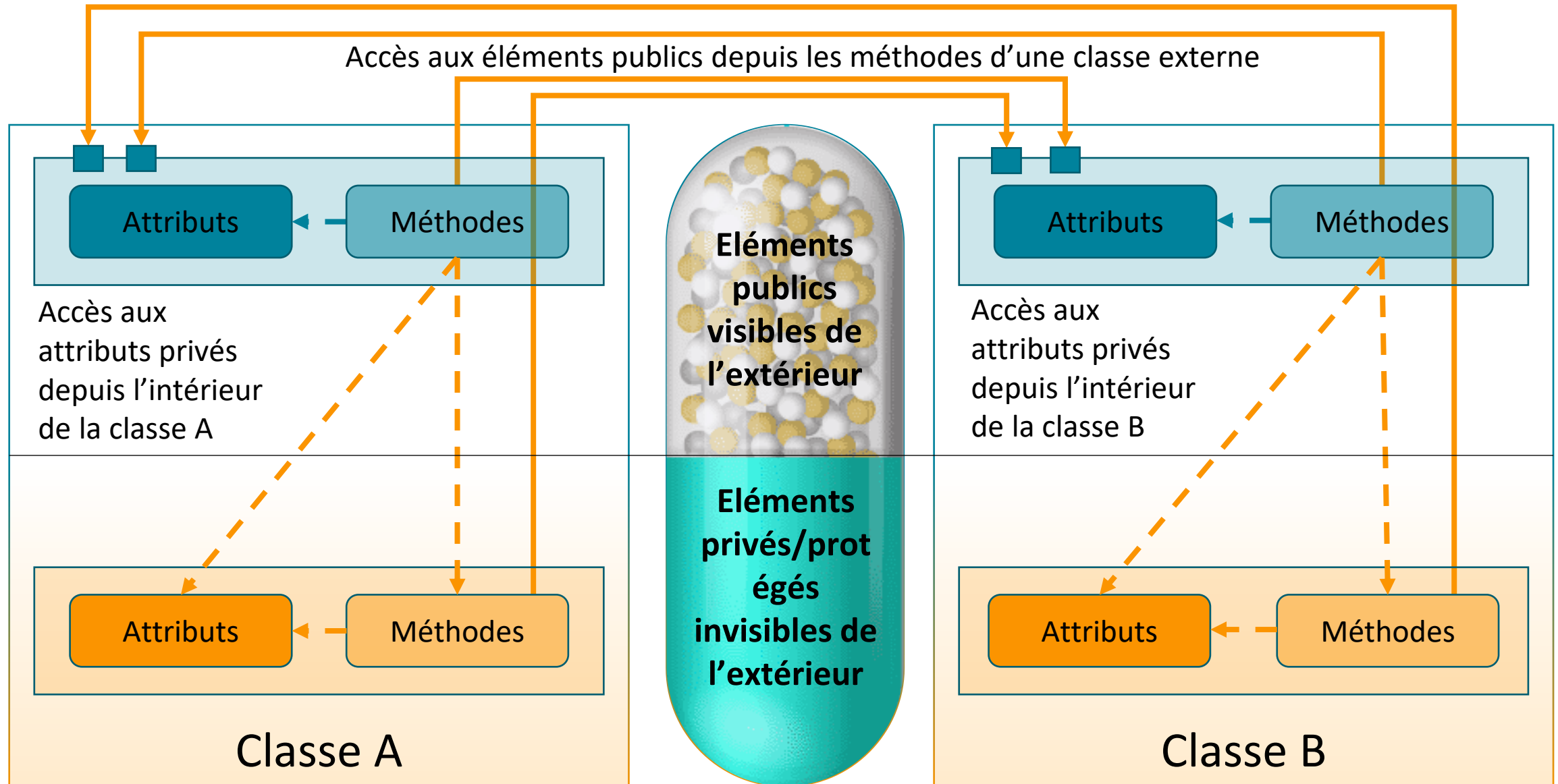
Un **mutateur** permet également de vérifier les conditions d'accès en écriture à un attribut. C'est l'intérêt du **principe d'encapsulation**.

On nomme par convention les accesseurs de la façon suivante : **setNomattribut**.

Un accesseur ou un mutateur est forcément une **méthode publique**, car elle doit pouvoir être appelée depuis les classes qui n'ont pas accès aux attributs privés.

```
public class Personne {  
  
    // Attributs  
    private String nom;  
    private int age;  
  
    // Mutateurs - Setters  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    public void setAge(User user, int age) {  
  
        if (verifierAcces(user, "écriture")) {  
            this.age = age;  
        } else {  
            System.out.println("Accès refusé");  
        }  
    }  
}
```

# Définition d'une classe - Encapsulation



# Définition d'une classe (attributs constants et attributs de classe)

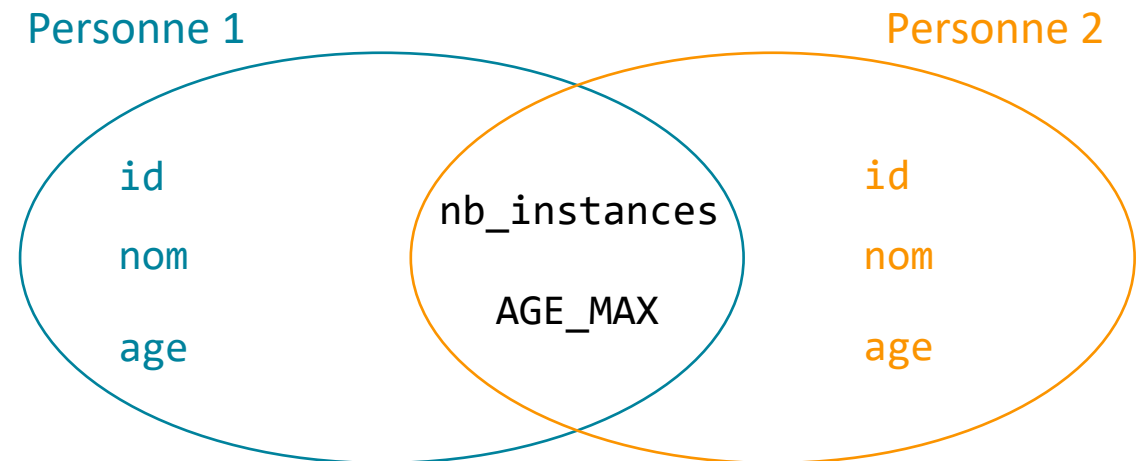
Il existe deux groupes d'attributs spécifiques qui sont les **attributs constants**, et les **attributs de classe**.

On déclare un **attribut constant**, avec le modificateur **final** lorsque celui-ci n'est pas voué à changer au cours de la vie d'un objet.

On déclare un **attribut de classe**, avec le modificateur **static**, quand celui-ci est partagé par tous les objets d'une même classe (compteur d'objets, par exemple).

Un attribut peut être **à la fois un attribut de classe et une constante**. C'est souvent le cas des variables globales.

```
public class Personne {  
    // Attribut constant  
    private final int id;  
  
    // Attribut de classe  
    private static int nb_instances;  
  
    // Attribut constant de classe  
    private static final int AGE_MAX = 3000;  
}
```





# Définition d'une classe (Constructeurs de la classe)

Pour obtenir une instance d'une classe, il faut « construire » l'instance. Pour construire une instance, il faut un **constructeur**.

Un constructeur est une méthode spécifique, qu'on retrouve dans toutes les classes et dont l'objectif est de **donner une valeur** aux attributs de l'objet instancié. Pour cela, les valeurs à attribuer sont passées en **paramètres** du constructeur.

Un constructeur **porte le nom de la classe** qu'il est censé construire et on ne lui spécifie **pas de type de retour**.

C'est le constructeur qui est appelé lorsqu'on utilise l'opérateur **new** pour créer un objet (détail dans le slide **Créer et manipuler des objets à partir d'une classe**).

Une classe doit donc nécessairement avoir un constructeur.

```
public class Personne {  
    // Attributs  
    private String nom;  
    private int age;  
  
    // Constructeur  
    public Personne(String nom, int age){  
        this.nom = nom;  
        this.age = age;  
    }  
}
```

Nom du constructeur identique au nom de la classe

Valeur des attributs en paramètres

Attribution des valeurs

# Surcharge de méthodes

Une méthode est identifiée par le compilateur grâce à :

- Sa classe
- Son nom
- Sa liste de paramètres (noms, types, ordre)

Le doublet (nom, listeDeParametres) qui constitue la signature d'une méthode est donc unique au sein d'une classe.

Néanmoins, il est possible de définir deux méthodes ayant le même nom mais dont la signature diffère. On parle alors de surcharge de méthode, qui permet d'appeler le même service avec une liste de paramètres qui varie.

La surcharge de méthode est par exemple très utilisée pour définir différents constructeurs, comme dans l'exemple ci-contre.

```
public class Personne {  
    private String nom;  
    private int age;  
    // Constructeur 1  
    public void Personne(String nom, int age) {  
        this.nom = nom;  
        this.age = age;  
    }  
    // Constructeur 2  
    public void Personne(String nom) {  
        this.nom = nom;  
        this.age = -1;  
    }  
}
```

```
Personne leto = new Personne("Leto Atréides");  
leto.setAge(3500);
```

```
Personne paul = new Personne("Paul Atréides", 15);
```

# Surcharge de méthodes

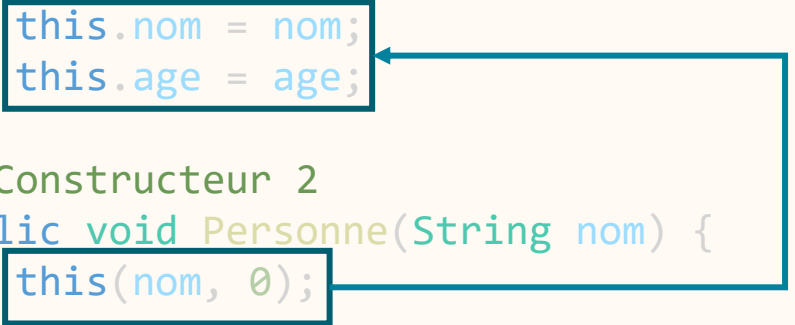
Il arrive que les constructeurs s'appellent entre eux. Pour appeler un constructeur dans un autre constructeur de la classe, on utilise la syntaxe :

`this(listeDeParametres);`

Si la liste de paramètres est cohérente et existe pour un autre constructeur de la classe, alors ce constructeur est appelé. Dans le cas contraire, cela résulte en une erreur de compilation.

Dans l'exemple ci-contre, `this(nom, 0);` présente une liste de paramètres qui correspond à celle attendue par le constructeur 1. Ainsi, l'instruction provoque l'exécution du constructeur 1.

```
public class Personne {  
    private String nom;  
    private int age;  
    // Constructeur 1  
    public void Personne(String nom, int age) {  
        this.nom = nom;  
        this.age = age;  
    }  
    // Constructeur 2  
    public void Personne(String nom) {  
        this(nom, 0);  
    }  
}
```





# Définition d'une classe (Constructeurs par défaut)

Pour obtenir une instance d'une classe, il faut « construire » l'instance. Pour construire une instance, il faut un **constructeur**.

Si aucun constructeur n'est défini, un constructeur par défaut est appelé et attribue des valeurs par défaut aux attributs de la classe.

Liste des valeurs attribuées par défaut, en fonction du type :

- int, byte, short, long => 0
- float, double => 0.0
- char => \u0000 (caractère null)
- boolean => false
- types objet => null

```
// Classe sans constructeur surchargé
public class Personne {

    // Attributs
    private String nom;
    private int age;
}
```

```
Personne personne_default = new Personne();

System.out.println(personne_default.getId());
// >> 0

System.out.println(personne_default.getNom());
// >> " "

System.out.println(personne_default.getAge());
// >> 0
```

# Définition d'une classe (Méthode toString())

```
public class Personne {  
  
    // Attributs  
    private String nom;  
    private int age;  
  
    public String toString() {  
        return "Personne[nom=" + nom + ", age=" + age + ", id=" + id + "]";  
    }  
}
```

Contrairement aux types primitifs, l'affichage des types objet n'est pas défini par défaut.

Pour afficher un objet de façon explicite, on définit alors une méthode spécifique : la méthode toString qui construit la chaîne de caractère décrivant l'objet.

Lorsque l'on affiche un objet qui ne possède pas de méthode toString(), seule la référence est affichée.

```
// Sans la méthode toString() surchargée  
System.out.print(paul);  
>> @1b28cdfa  
  
// Avec la méthode toString() surchargée  
System.out.print(paul);  
>> Personne[nom=Paul Atréides, age=15, id=1]
```



# 03

## Relations entre les classes

# Relations entre les classes

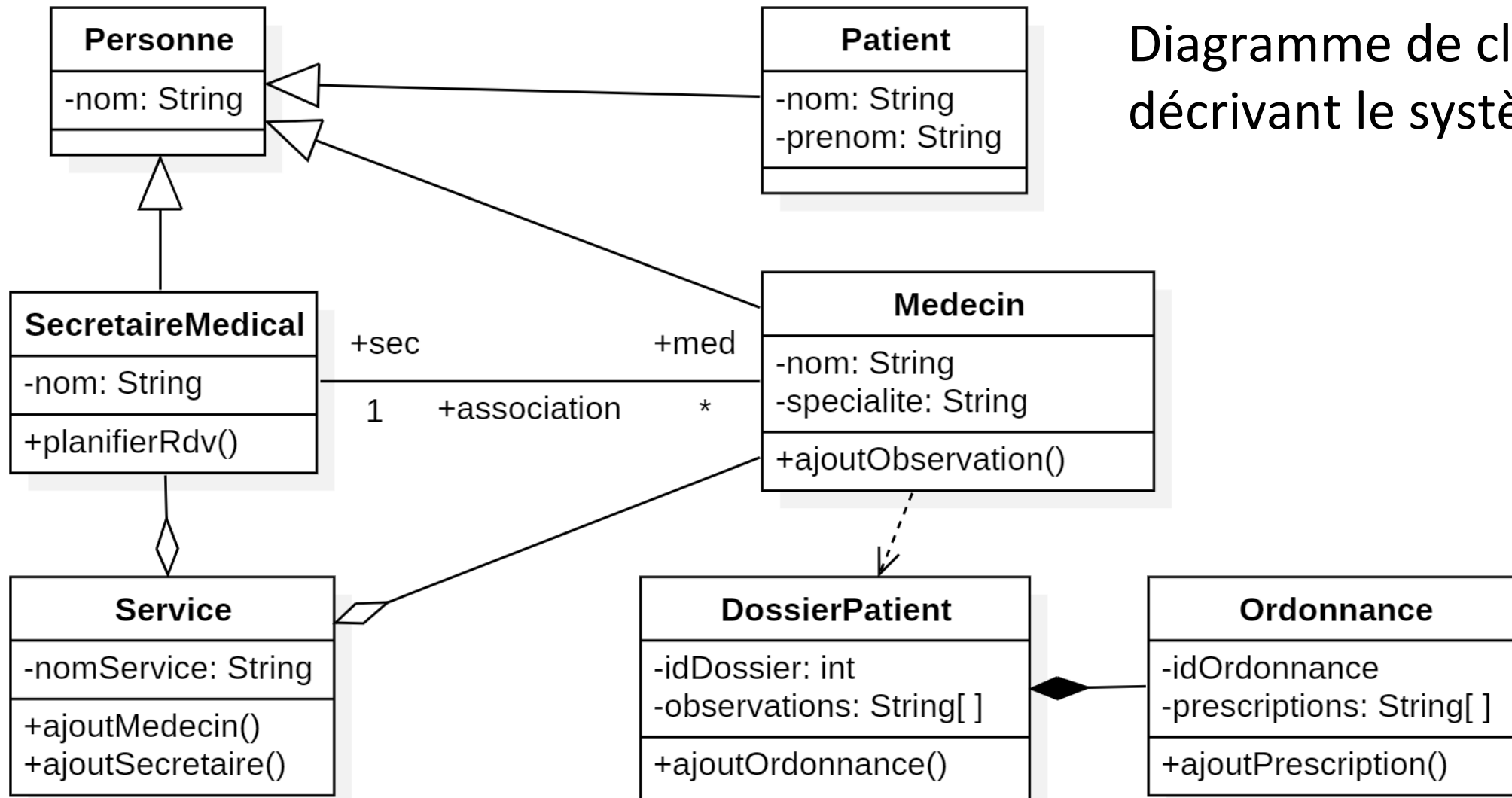
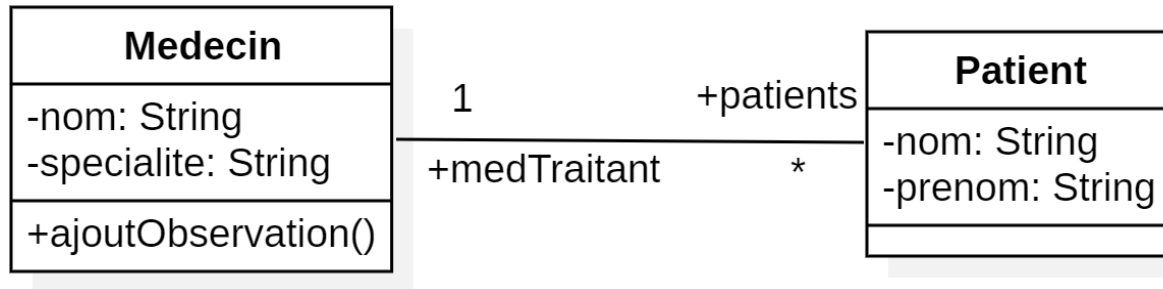


Diagramme de classes UML  
décrivant le système Hôpital



# Relation d'association



La création d'une **relation d'association** entre deux classes donne lieu à la création d'attributs dans les classes impliquées dans la relation.

La multiplicité de la relation définit le type de l'attribut ajouté à chaque classe :

- Multiplicité **1** => **Objet**
- Multiplicité **\*** => **Collection<Objet>**

Dans l'exemple ci-contre, la relation entre la classe **Patient** et la classe **Medecin** entraîne la création des attributs :

**medTraitant** (de type **Medecin**) dans la classe **Patient**  
**Patients** de type **HashSet<Patient>** dans la classe **Medecin**.

```

public class Patient {

    private String nom;
    private String prenom;
    private Medecin medTraitant;

    // Suite de la classe
}

```

```

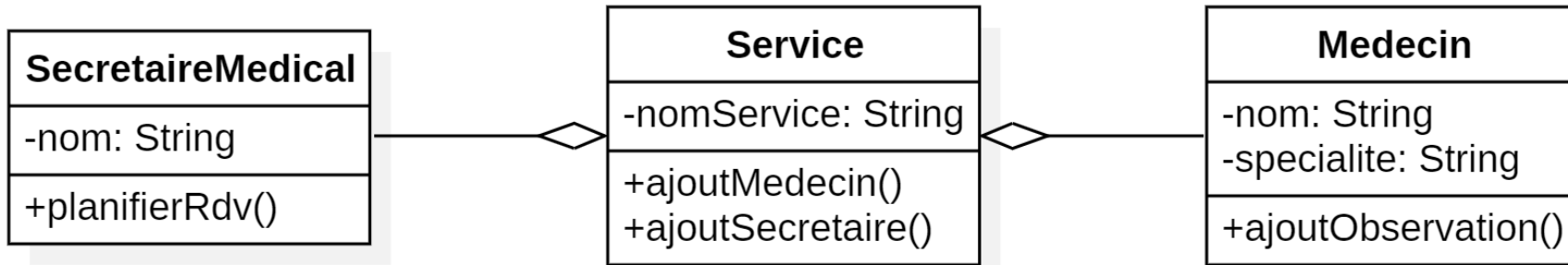
public class Medecin {

    private String nom;
    private String specialite;
    private ArrayList<Patient> patients;

    // Suite de la classe
}

```

# Relation d'agrégation



Dans une **relation d'agrégation**, les objets qui s'agrègent (**Médecin**, **SecrétaireMedical**) sont construits à l'extérieur de l'objet agrégeant (Service) car indépendants de celui-ci.

Dans l'exemple ci-contre, le constructeur de la classe **Service** prend en argument une liste de médecins et une liste de secrétaires médicaux.

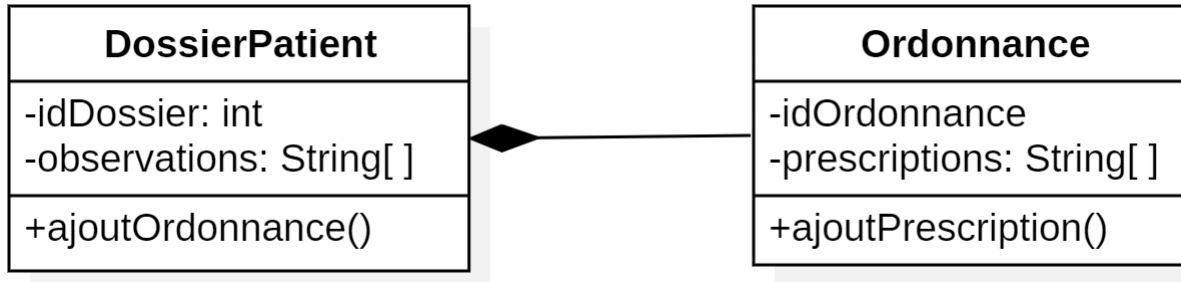
C'est également le cas à l'ajout, on peut créer un **médecin** ou un **secrétaire** et le passer en argument des méthodes assurant l'ajout.

C'est également le cas à la destruction d'un **Service**, mais invisible car déjà géré par le garbage collector.

```

public class Service {
    private String nomService;
    private ArrayList<Medecin> medecins;
    private ArrayList<SecretaireMedical> secretaires;
    public Service(String nom, ArrayList<Medecin> medecins,
                  ArrayList<SecretaireMedical> secretaires) {
        this.medecins = medecins;
        this.secretaires = secretaires;
        this.nomService = nom;
    }
    public void ajoutMedecin(Medecin medecin) {
        this.medecins.add(medecin);
    }
    public void ajoutSecretaire(SecretaireMedical secretaire) {
        this.secretaires.add(secretaire);
    }
}
  
```

# Relation de composition



Dans une relation de composition, la construction des objets composants (**Ordonnances**) est de la responsabilité de l'objet composé (**DossierPatient**).

```

public class DossierPatient {
    private int idDossier;
    private HashMap<String, String> observations;
    private ArrayList<Ordonnance> ordonnances;

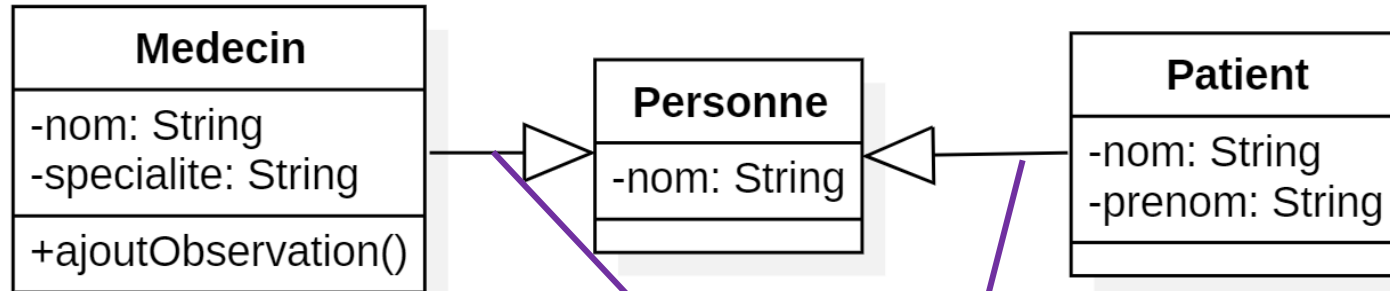
    public DossierPatient() {
        this.idDossier = 0;
        this.observations = new HashMap<>();
        this.ordonnances = new ArrayList<>();
    }

    public void ajoutOrdonnance(int id, HashMap<String, String> prescriptions) {
        this.ordonnances.add(new Ordonnance(id, prescriptions));
    }
}
  
```

Dans l'exemple ci-contre, c'est le constructeur de la classe **Dossier** qui est en charge de la création des **Ordonnances**.

De cette manière, l'ordonnance n'existe pas en dehors du dossier.

# Relation d'héritage



Une **relation d'héritage** entre deux classes se traduit par le mot clé **extends** lors de la déclaration de la classe **filles**, suivi du nom de la classe **mère**.

```
public class Medecin extends Personne {
    // Détails de la classe Medecin
}
```

L'héritage permet de réutiliser ce qui a été implémenté dans une classe sans avoir à le réécrire.

Une classe fille hérite donc de tous les attributs et méthodes (public et protected) de la classe mère.

```
public class Patient extends Personne {
    // Détails de la classe SecretaireMedical
}
```

```
public class Medecin extends Personne, SecretaireMedical {
    // Détails de la classe SecretaireMedical
}
```

Une classe mère peut avoir plusieurs classes filles, mais l'inverse n'est pas possible.

En Java, l'héritage multiple est géré par les interfaces.





# Relation d'héritage

```

Personne personne = new Personne();
Patient patient = new Patient();
Medecin medecin = new Medecin();

personne.parler();
// >> Bonjour !
medecin.parler();
// >> Ça vous chatouille ou ça vous gratouille ?
patient.parler();
// >> Ça me gratouille...
patient.crier();
// >> Aïe !
medecin.crier();
// >> error : cannot find symbol
// >> symbol: method crier

```

```

public class Patient extends Personne {

    @Override
    public void parler() {
        System.out.println("Ça me gratouille...");
    }

    public void crier() {
        System.out.println("Aïe !");
    }

}

```

```

public class Personne {

    public void parler() {
        System.out.println("Bonjour !");
    }

}

```

```

public class Medecin extends Personne {

    @Override
    public void parler() {
        System.out.println("Ça vous chatouille ou ça vous gratouille ?");
    }

}

```

Une Classe ???

Un Objet ???

A quoi sert la programmation orientée objet ?

# JAVA OOP CHEAT SHEET

Learn JAVA from experts at <https://www.edureka.co>

## Object Oriented Programming in Java

Java is an Object Oriented Programming language that produces software for multiple platforms. An object-based application in Java is concerned with declaring classes, creating objects from them and interacting between these objects.



### Java Class

```
class Test {  
    // class body  
    member variables  
    methods  
}
```

### Java Object

```
//Declaring and Initializing an object  
Test t = new Test();
```

## Constructors

### Default Constructor

```
class Test{  
    /* Added by the Java Compiler at the Run Time  
    public Test(){  
    }  
    */  
    public static void main(String args[]) {  
        Test testObj = new Test();  
    }  
}
```

### Parameterized Constructor

```
public class Test {  
    int appId;  
    String appName;  
    //parameterized constructor with two parameters  
    Test(int id, String name){  
        this.appId = id;  
        this.appName = name;  
    }  
    void info(){  
        System.out.println("Id: "+appId+" Name: "+appName);  
    }  
    public static void main(String args[]){  
        Test obj1 = new Test(11001,"Facebook");  
        Test obj2 = new Test(23003,"Instagram");  
        obj1.info();  
        obj2.info();  
    }  
}
```



**JAVA CERTIFICATION  
TRAINING**

## Modifiers in Java

### Access Modifiers

Scope	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

## Inheritance

### Single Inheritance

```
Class A {  
    //your parent class code  
}  
Class B extends A {  
    //your child class code  
}
```

### Multi Level Inheritance

```
Class A {  
    //your parent class code  
}  
Class B extends A {  
    //your code  
}  
Class C extends B {  
    //your code  
}
```

### Hybrid Inheritance



## Polymorphism

### Compile Time Polymorphism

```
class Calculator {  
    static int add(int a, int b){  
        return a+b;  
    }  
    static double add( double a, double b){  
        return a+b;  
    }  
    public static void main(String args[]){  
        System.out.println(Calculator.add(123,17));  
        System.out.println(Calculator.add(18.3,1.9));  
    }  
}
```

### Run Time Polymorphism

```
public class Mobile{  
    void sms(){System.out.println("Mobile class");}  
}  
//Extending the Mobile class  
public class OnePlus extends Mobile{  
    //Overriding sms() of Mobile class  
    void sms(){  
        System.out.println("OnePlus class");  
    }  
    public static void main(String[] args) {  
        OnePlus smsObj= new OnePlus();  
        smsObj.sms();  
    }  
}
```

### Hierarchical Inheritance

```
Class A {  
    //your parent class code  
}  
Class B extends A {  
    //your child class code  
}  
Class C extends A {  
    //your child class code  
}
```

### Multiple Inheritance

```
Class A {  
    //your parent class code  
}  
Class B {  
    //your parent class code  
}  
Class C extends A,B {  
    //your child class code  
}
```

## Abstraction

### Abstract Class

```
public abstract class MyAbstractClass  
{  
    public abstract void abstractMethod();  
    public void display(){  
        System.out.println("Concrete method");  
    }  
}
```

### Interface

```
//Creating an Interface  
public interface Bike { public void start(); }  
//Creating classes to implement Bike interface  
class Honda implements Bike{  
    public void start(){  
        System.out.println("Honda Bike");  
    }  
}  
class Apache implements Bike{  
    public void start(){  
        System.out.println("Apache Bike");  
    }  
}  
class Rider{  
    public static void main(String args[]){  
        Bike b1=new Honda();  
        b1.start();  
        Bike b2=new Apache();  
        b2.start();  
    }  
}
```

## Encapsulation

```
public class Artist {  
    private String name;  
    //getter method  
    public String getName() { return name; }  
    //setter method  
    public void setName(String name) { this.name = name; }  
}  
public class Show{  
    public static void main(String[] args){  
        //creating instance of the encapsulated class  
        Artist s=new Artist();  
        //setting value in the name member  
        s.setName("BTS");  
        //getting value of the name member  
        System.out.println(s.getName());  
    }  
}
```

# JAVA CHEATSHEET

Learn JAVA from experts at [www.edureka.co](http://www.edureka.co)

## Iterative Statements

```
// for loop  
for (condition) {expression}  
  
// for each loop  
for (int i: someArray) {}  
  
// while loop  
while (condition) {expression}  
  
// do while loop  
do {expression} while(condition)
```

### Fibonacci series

```
for (i = 1; i <= n; ++i)  
{  
    System.out.print(t1 + " + ");  
    int sum = t1 + t2; t1 = t2;  
    t2 = sum;  
}
```

### Pyramid Pattern

```
k = 2*n - 2;  
for(i=0; i<n; i++)  
{  
    for(j=0; j<k; j++){System.out.print(" ");}  
    k = k - 1;  
    for(j=0; j<=i; j++) {System.out.print("* ");}  
    System.out.println();  
}
```

## Decisive Statements

```
//if statement  
if (condition) {expression}  
  
//if-else statement  
if (condition) {expression} else {expression}  
  
//switch statement  
switch (var) { case 1: expression; break;  
default: expression; break; }
```

### Prime Number

```
if (n < 2)  
{  
    return false;  
}  
for (int i=2; i <= n/i; i++)  
{  
    if (n%i == 0) return false;  
}  
return true;
```

### Factorial of a Number

```
int factorial(int n)  
{  
    if (n == 0)  
        return 1;  
    else  
    {  
        return(n * factorial(n-1));  
    }  
}
```

## Basic Java Program



Save className.java

Compile javac className

Execute java className

## Arrays In Java

### 1 - Dimensional

```
// Initializing  
type[] varName= new type[size];  
  
// Declaring  
type[] varName= new type[]{values1, value2,...};
```

### Array with Random Variables

```
double[] arr = new double[n];  
for (int i=0; i<n; i++)  
{a[i] = Math.random();}
```

### Maximum value in an Array

```
double max = 0;  
for (int i=0; i<arr.length(); i++)  
{ if(a[i] > max) max = a[i]; }
```

### Reversing an Array

```
for(int i=0; i<(arr.length())/2; i++)  
{ double temp = a[i];  
a[i] = a[n-1-i];  
a[n-1-i] = temp; }
```

### Multi – Dimensional Arrays

```
// Initializing  
datatype[][] varName = new datatype[row][col];  
// Declaring  
datatype[][] varName = {{value1, value2...},{value1,  
value2...}...};
```

### Transposing A Matrix

```
for(i = 0; i < row; i++)  
{ for(j = 0; j < column; j++)  
{ System.out.print(array[i][j]+" "); }  
System.out.println(" ");  
}
```

### Multiplying two Matrices

```
for (i = 0; i < row1; i++)  
{ for (j = 0; j < col2; j++)  
{ for (k = 0; k < row2; k++)  
{ sum = sum + first[i][k]*second[k][j]; }  
multiply[i][j] = sum;  
sum = 0; } } }
```

## Java Strings

```
// Creating String using literal  
String str1 = "Welcome";
```

```
// Creating String using new keyword  
String str2 = new String("Eduureka");
```

### String Methods

```
str1==str2 //compare the address;  
String newStr = str1.equals(str2); //compares the values  
String newStr = str1.equalsIgnoreCase() //  
newStr = str1.length() //calculates length  
newStr = str1.charAt(i) //extract i'th character  
newStr = str1.toUpperCase() //returns string in ALL CAPS  
newStr = str1.toLowerCase() //returns string in ALL LOWERCASE  
newStr = str1.replace(oldVal, newVal) //search and replace  
newStr = str1.trim() //trims surrounding whitespace  
newStr = str1.contains("value"); //Check for the values  
newStr = str1.toCharArray(); //Convert into character array  
newStr = str1.isEmpty(); //Check for empty String  
newStr = str1.endsWith(); //Checks if string ends with the given suffix
```

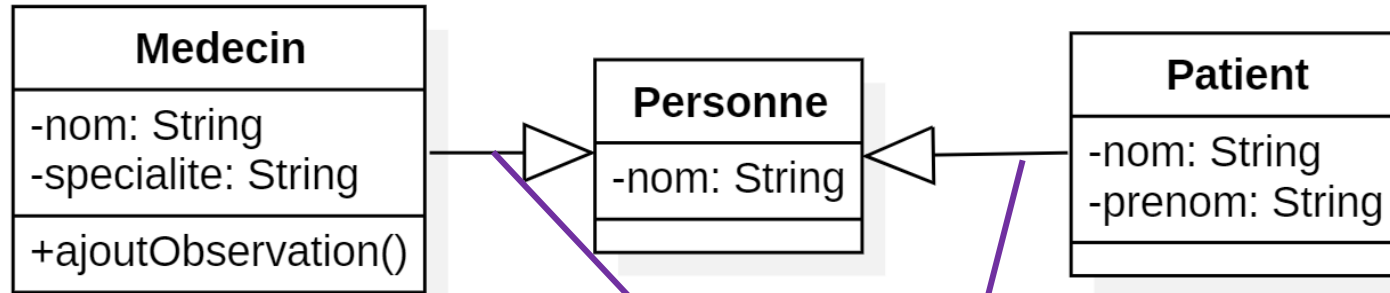


04

# Interfaces, exceptions, énumération, Principes SOLID



# Relation d'héritage



Une **relation d'héritage** entre deux classes se traduit par le mot clé **extends** lors de la déclaration de la classe **filles**, suivi du nom de la classe **mère**.

```
public class Medecin extends Personne {
    // Détails de la classe Medecin
}
```

L'héritage permet de réutiliser ce qui a été implémenté dans une classe sans avoir à le réécrire.

Une classe fille hérite donc de tous les attributs et méthodes (public et protected) de la classe mère.

```
public class Patient extends Personne {
    // Détails de la classe SecretaireMedical
}
```

```
public class Medecin extends Personne, SecretaireMedical {
    // Détails de la classe SecretaireMedical
}
```

Une classe mère peut avoir plusieurs classes filles, mais l'inverse n'est pas possible.

En Java, l'héritage multiple est géré par les interfaces.



# Relation d'héritage

```

Personne personne = new Personne();
Patient patient = new Patient();
Medecin medecin = new Medecin();

personne.parler();
// >> Bonjour !
medecin.parler();
// >> Ça vous chatouille ou ça vous gratouille ?
patient.parler();
// >> Ça me gratouille...
patient.crier();
// >> Aïe !
medecin.crier();
// >> error : cannot find symbol
// >> symbol: method crier

```

```

public class Patient extends Personne {

    @Override
    public void parler() {
        System.out.println("Ça me gratouille...");
    }

    public void crier() {
        System.out.println("Aïe !");
    }

}

```

```

public class Personne {

    public void parler() {
        System.out.println("Bonjour !");
    }

}

```

```

public class Medecin extends Personne {

    @Override
    public void parler() {
        System.out.println("Ça vous chatouille ou ça vous gratouille ?");
    }

}

```

# Sur-typage et sous-typage

```
Personne personne = new Personne(); // Typage simple
```

Main.java

```
Personne patient_sur_typee = new Patient(); // Sur-typage
```

Main.java

**Sur-typage** : On parle de sur-typage lorsqu'une classe mère (Personne) est utilisée pour instancier un objet de la classe fille (Patient).

**Sous-typage** : Le sous-typage en revanche, est l'opération inverse, c'est-à-dire la définition de l'instance d'une classe mère à l'aide de la classe fille.

Java permet le sur-typage mais ne supporte pas le sous-typage de façon automatique.

Il est toujours possible de faire un sous-typage de façon explicite, mais il faut s'assurer que l'instance remplit les exigences de la classe de sur-typage. Cette pratique est toutefois déconseillée.

```
Patient patient = new Personne(); // Sous-typage
```



Main.java

```
Personne personne_sous_typee
// Opérations intermédiaires
if (personne_sous_typee instanceof Patient){
    personne_sous_typee = (Patient) personne_sous_typee; // Sous-typage explicite
}
```

Main.java

A éviter

# Opérateur instanceof

```
obj instanceof Class
```

Main.java

L'opérateur instanceof permet de s'assurer qu'une instance répond aux exigences d'une classe ou d'une interface en terme d'attributs et de méthodes.

```
CustomCollection collection = new Collection()

if (collection instanceof List) {
    System.out.println("L'élément est une liste avec " + collection.size() + "
        éléments.");
} else if (collection instanceof Map) {
    System.out.println("L'élément est une map avec " + collection.size() + "
        paires clé-valeur.");
}
}
```

Main.java

On l'utilise notamment avant de forcer le sous-typage (caster) d'une instance (voir slide précédente).



# Classes abstraites et méthodes abstraites

```
public class Personne {
    protected String nom;

    public void parler() {
        System.out.println("Bonjour !");
    }
}
```

Personne.java

```
public class Patient extends Personne {

    public void parler() {
        System.out.println("Ça me gratouille...");
    }

    public void crier() {
        System.out.println("Aïe !");
    }
}
```

Personne.java

```
public abstract class Personne {
    protected String nom;

    public abstract void parler();
}
```

Personne.java

On peut également déclarer des classes de façon abstraite, dans lesquelles certaines méthodes (abstraites également) ne sont pas implémentées.

Cela oblige les classes héritées à implémenter les méthodes abstraites, sans quoi une erreur est levée par le compilateur.

Attention, une classe abstraite, comme une interface ne peut pas être instanciée directement.

Déclarer des classes abstraites est une bonne pratique, permettant d'assurer la robustesse du code.

# Interfaces

```
public class Medecin extends Personne, SecretaireMedical {

    // Détails de la classe SecretaireMedical
}
```

Rappel : L'héritage multiple n'est pas géré par java.

À la place, on passe par des interfaces.

Une interface se déclare comme une classe dans laquelle les méthodes ne sont pas implémentées, et dit ce que peut faire une classe (services).

```
public interface Electronique {
    void powerOn();
    void powerOff();
}
```

Electronique.java

```
public interface Vehicule {
    void start();
    void stop();
}
```

Vehicule.java

Une même interface peut être implémentée par plusieurs classes.

Une même classe peut implémenter plusieurs interfaces (ce qui règle le problème de l'héritage multiple).

L'implémentation d'une interface se fait au niveau de la classe qui l'implémente (via le mot clé **implements**).

```
public class Voiture implements Vehicule, Electronique {
    public void start() { }

    public void stop() { }

    public void powerOn() { }
}
```

Voiture.java

```
public class Velo implements Vehicule {

    public void start() { }

    public void stop() { }
}
```

Velo.java

# Invariants de classe et exceptions

Lorsqu'on définit une classe (méthodes et attributs), on ne définit pas nécessairement toutes les contraintes implicites qui garantissent la bonne exécution des méthodes. Il faut tout de même les assurer. Pour cela, on peut définir des invariants de classe qui doivent être toujours vérifiés.

```

public class CompteBancaire {
    public void retirer(double montant) {
        if (montant <= 0) {
            throw new IllegalArgumentException("Le montant à
            retirer doit être positif.");
        }
        if (montant > solde) {
            throw new IllegalArgumentException("Fonds
            insuffisants.");
        }
        solde -= montant;
    }
    private void verifierInvariant() {
        if (solde < 0) {
            throw new IllegalStateException("L'invariant de
            classe est violé : le solde ne peut pas être négatif.");
        }
    }
}

```

CompteBancaire.java

Lorsqu'un invariant de classe ou une condition n'est pas vérifié, pour éviter des erreurs à l'exécution, on peut lever des exceptions.

De nombreuses exceptions existent en Java, parmi lesquelles :

- RuntimeException
- IOException
- IllegalArgumentException

Un invariant doit être vérifié à chaque appel de méthode. Il peut donc être judicieux de définir une méthode dédiée à la vérification de ces invariants.

# Principes SOLID

Ensemble de principes de programmation orientée objet qui permettent de créer du code robuste.

Single  
responsibility



Open close  
principle

Liskov  
substitution



Interface  
segregation

Dependency  
inversion



Martin, R. C. (2000). Design principles and design patterns. Object Mentor, 1(34), 597.



# Single responsibility principle



« Une classe ne doit changer que pour une seule raison »

Autrement dit, chaque classe ne doit être responsable que d'une seule fonctionnalité du système.

```
public class Livre { Livre.java
    private String titre;
    private String auteur;

    public Livre(String titre, String auteur) {
        this.titre = titre;
        this.auteur = auteur;
    }

    public void enregistrerEnBaseDeDonnees() {
        // Code
    }

    public void envoyerParEmail() {
        // Code
    }
}
```

Ici, la classe Livre gère à la fois l'enregistrement du livre et l'envoi par mail des informations.

Ces fonctionnalités doivent être déléguées à d'autres classes.

```
public class Main { Main.java
    public static void main(String[] args) {
        Livre livre = new Livre("Le Petit Prince",
                                "Antoine de Saint-Exupéry");
        livre.enregistrerEnBaseDeDonnees();
        livre.envoyerParEmail();
    }
}
```

# Single responsibility principle



« Une classe ne doit changer que pour une seule raison »

```
public class Livre {
    private String titre;
    private String auteur;

    public Livre(String titre, String auteur) {
        this.titre = titre;
        this.auteur = auteur;
    }
}
```

Livre.java

```
public class LivreRepository {
    public void enregistrer(Livre livre) {
    }
}
```

LivreRepository.java

```
public class NotificationService {
    public void envoyerEmail(Livre livre) {
    }
}
```

NotificationService.java

```
public class MainSRP {
    public static void main(String[] args) {
        Livre livre = new Livre("Le Petit Prince", "Antoine de Saint-Exupéry");
        LivreRepository repository = new LivreRepository();
        NotificationService notificationService = new NotificationService();

        repository.enregistrer(livre);
        notificationService.envoyerEmail(livre);
    }
}
```

MainSRP.java

# Open close principle



« Une classe doit être à la fois ouverte à l'extension et fermée à la modification »

```
class Forme {
    private String type;
    private double rayon, largeur, hauteur;

    Forme(String type, double dimension1, double dimension2) {
        this.type = type;
        if (type.equals("Cercle")) {
            this.rayon = dimension1;
        } else if (type.equals("Rectangle")) {
            this.largeur = dimension1;
            this.hauteur = dimension2;
        }
    }

    double calculerAire() {
        if (type.equals("Cercle")) {
            return Math.PI * rayon * rayon;
        } else if (type.equals("Rectangle")) {
            return largeur * hauteur;
        }
        return 0;
    }
}
```

Forme.java

Une fois codée et validée par les tests unitaires, une classe ne doit plus être modifiée.

En revanche il est possible de l'étendre, c'est-à-dire de créer des classes héritées.

Ici, la classe Forme permet de créer un cercle ou un rectangle. Mais si l'on souhaite créer un triangle, il faut modifier toutes les méthodes de la classe Forme.

Pour éviter cela, on préférera créer une classe Forme générique, et étendre cette classe en créant les classes Cercle, Rectangle et Triangle.

# Open close principle



« Une classe doit être à la fois ouverte à l'extension et fermée à la modification »

```
abstract class Forme {  
    private String color;  
    abstract double calculerAire();  
}
```

Forme.java

```
class Rectangle extends Forme {  
    private double largeur;  
    private double hauteur;  
  
    Rectangle(double largeur, double hauteur) {  
        this.largeur = largeur;  
        this.hauteur = hauteur;  
    }  
  
    double calculerAire() {  
        return largeur * hauteur;  
    }  
}
```

Rectangle.java

```
class Cercle extends Forme {  
    private double rayon;  
  
    Cercle(double rayon) {  
        this.rayon = rayon;  
    }  
  
    double calculerAire() {  
        return Math.PI * rayon * rayon;  
    }  
}
```

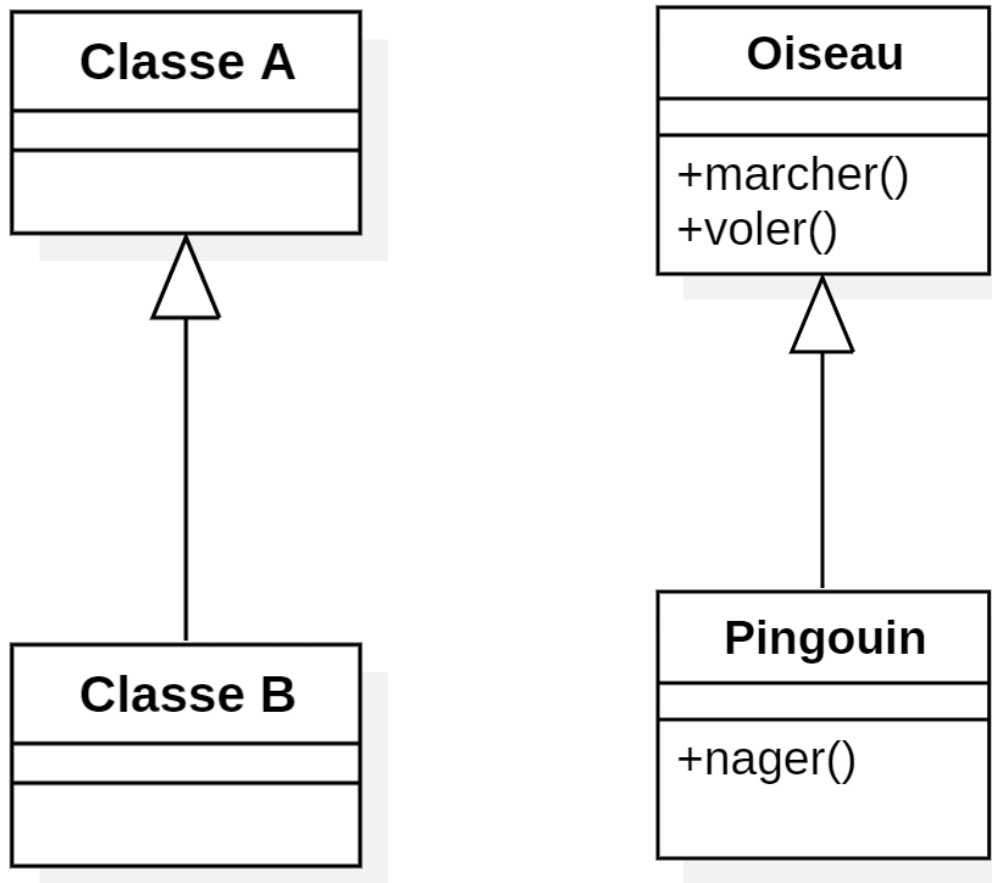
Cercle.java



# Liskov substitution principle



« Si B est une sous classe de A, alors tout objet de type A peut être remplacé par un objet de type B sans altérer les propriétés désirables du programme concerné »



Ici le principe de Liskov n'est pas respecté :

La classe Oiseau implémente la méthode voler()

La classe Pingouin est héritée de la classe Oiseau

Hors, un pingouin ne sait pas voler...

On ne pourrait pas remplacer tous les oiseaux par des pingouins

Ceci est donc contraire au principe de Liskov.

# Liskov substitution principle



« Si B est une sous classe de A, alors tout objet de type A peut être remplacé par un objet de type B sans altérer les propriétés désirables du programme concerné »

```
class Oiseau {
    void voler() {
        System.out.println("L'oiseau vole !");
    }
}
```

Oiseau.java

```
class Pingouin extends Oiseau {
    void voler() {
        throw new UnsupportedOperationException(
            "Le pingouin ne peut pas voler !");
    }
}
```

Pingouin.java

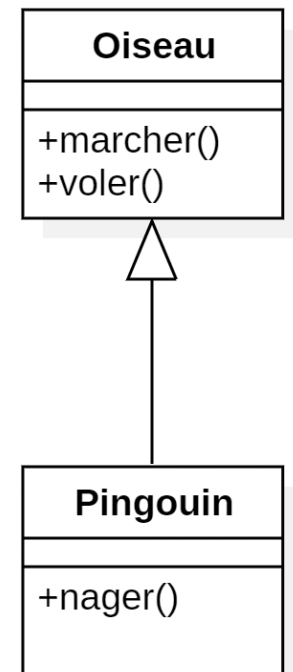
```
public class Main {
    public static void main(String[] args) {
        Oiseau oiseau = new Oiseau();
        oiseau.voler();

        Oiseau pingouin = new Pingouin();
        try {
            pingouin.voler(); // Ceci lancera une exception
        } catch (UnsupportedOperationException e) {
            System.out.println(e.getMessage());
            // Affiche : Le pingouin ne peut pas voler !
        }
    }
}
```

Main.java

Que se passe-t-il si l'on implémente quand même les méthodes ?

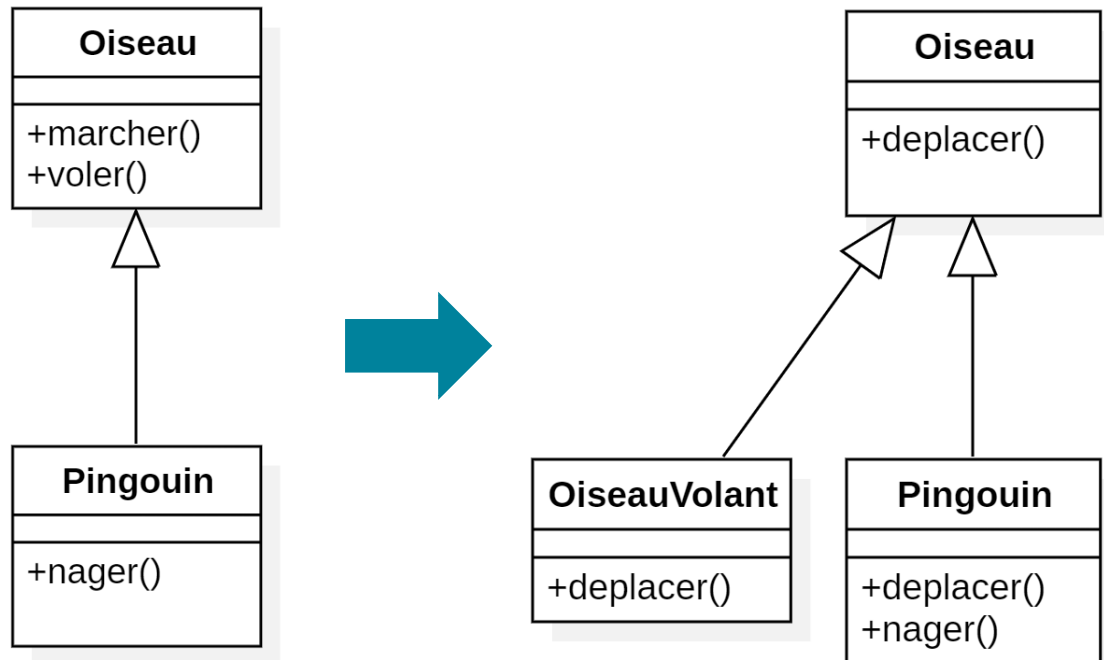
=> Si un oiseau doit voler quelque part dans le programme, cela ne sera pas possible dès qu'il sera remplacé par un pingouin, il faudra donc retoucher au programme.



# Liskov substitution principle



« Si B est une sous classe de A, alors tout objet de type A peut être remplacé par un objet de type B sans altérer les propriétés désirables du programme concerné »



```
class Oiseau {
    void deplacer() {
        System.out.println("L'oiseau se déplace !");
    }
}
```

Oiseau.java

```
class OiseauVolant extends Oiseau {
    void deplacer() {
        System.out.println("L'oiseau vole !");
    }
}
```

OiseauVolant.java

```
class Pingouin extends Oiseau {
    void deplacer() {
        System.out.println("Le pingouin marche !");
    }
    void nager() {
        System.out.println("Le pingouin nage !");
    }
}
```

Pingouin.java

# Interface segregation principle



« Une classe qui implémente une interface donnée ne doit pas implémenter des méthodes dont elle n'a pas l'utilité »

On cherchera donc à rendre les interfaces le plus atomiques possible.

```
public interface Distribuer {  
    void retirer(double montant);  
    void déposer(double montant);  
}
```

Distribuer.java

Dans cet exemple, on utilise une interface Distribuer qui déclare les méthodes retirer et déposer.

Cette Interface est utilisée pour implémenter la classe DistributeurRetrait qui ne permet pas de faire un dépôt. L'implémentation de la méthode déposer n'est donc pas nécessaire. Le principe de ségrégation des interfaces n'est pas respectée.

```
public class DistributeurRetrait implements Distribuer {  
    public void retirer(double montant) {  
        System.out.println("Retrait de " + montant + " euros.");  
    }  
  
    public void déposer(double montant) {  
        throw new UnsupportedOperationException("Cette opération n'est pas supportée.");  
    }  
}
```

DistributeurRetrait.java



# Interface segregation principle



« Une classe qui implémente un interface donnée ne doit pas implémenter des méthodes dont elle n'a pas l'utilité »

```
public interface Deposer {
    void deposer(double montant);
}
```

Deposer.java

```
public interface Retirer {
    void retirerArgent(double montant);
}
```

Retirer.java

La solution est donc de diviser l'interface Distribuer en deux interfaces : Retirer et Déposer.

```
public class DistributeurRetrait implements Retirer {
    public void retirer(double montant) {
        System.out.println("Retrait de " + montant + " euros.");
    }
}
```

DistributeurRetrait.java

Cela permet de définir la classe DistributeurRetrait qui n'implémente que la méthode retirer de l'interface Retirer.

```
public class DistributeurComplet implements Deposer, Retirer {
    public void retirer(double montant) {
        System.out.println("Retrait de " + montant + " euros.");
    }

    public void deposer(double montant) {
        System.out.println("Depot de " + montant + " euros.");
    }
}
```

DistributeurComplet.java

Il reste par ailleurs possible de réaliser l'implémentation de la classe DistributeurComplet en implémentant à la fois l'interface Deposer et l'interface Retirer.

# Dependency inversion principle



« Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. »

« Les abstractions ne doivent pas dépendre des détails »

Autrement dit, il ne faut pas mettre les classes en dépendance directe, mais utiliser le plus possible les abstractions (classes abstraites, interfaces)

```
class MoteurEssence {  
    void demarrer() {  
        System.out.println("Moteur démarre");  
    }  
}
```

Moteur.java

```
class Voiture {  
    private MoteurEssence moteur;  
  
    public Voiture() {  
        this.moteur = new MoteurEssence();  
    }  
  
    void demarrer() {  
        moteur.demarrer();  
    }  
}
```

Voiture.java

# Dependency inversion principle



« Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. »

« Les abstractions ne doivent pas dépendre des détails »

```
interface IMoteur {
    void demarrer();
}
```

IMoteur.java

```
interface IMoteur {
    void demarrer();
}
```

Pour casser la dépendance directe entre les deux classes, on passe par une interface

```
class MoteurEssence implements IMoteur {
    public void demarrer() {
        System.out.println("Moteur à essence démarre");
    }
}
```

MoteurEssence.java

La classe MoteurEssence implémente l'interface IMoteur. Il y a une dépendance entre IMoteur et Moteur.

```
class Voiture {
    private IMoteur moteur;

    public Voiture(IMoteur moteur) {
        this.moteur = moteur;
    }

    void demarrer() {
        moteur.demarrer();
    }
}
```

Voiture.java

La dépendance initiale entre la classe MoteurEssence et la classe Voiture est remplacée par une dépendance entre l'interface IMoteur et la classe Voiture.

Les dépendances directes entre classes ont disparu.

# Dependency inversion principle



« Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. »

« Les abstractions ne doivent pas dépendre des détails »

```
abstract class Moteur {  
    void demarrer();  
}
```

Moteur.java

On pourrait, de façon tout à fait analogue,  
passer par une classe abstraite

```
class MoteurEssence extends Moteur {  
    public void demarrer() {  
        System.out.println("Moteur à essence démarre");  
    }  
}
```

MoteurEssence.java

```
class Voiture {  
    private Moteur moteur;  
  
    public Voiture(Moteur moteur) {  
        this.moteur = moteur;  
    }  
    void demarrer() {  
        moteur.demarrer();  
    }  
}
```

Voiture.java

# Enumerations

```
public enum PointCardinal {
    NORD, SUD, EST, OUEST
}
```

PointCardinal.java

Une énumération, est un type de données spécial qui permet de définir un ensemble de constantes nommées.

Ces constantes sont considérées comme des valeurs possibles. Il s'agit d'un objet pratique pour définir un ensemble de données fini comme les jours de la semaine, des rôles pour les droits d'accès.

On déclare une énumération avec le mot clé **enum**.

Exemple d'utilisation :

```
Direction direction = Direction.NORD;
System.out.println("Direction actuelle : " + direction);
System.out.println("Direction opposée : " + direction.getOpposee());
```

Main.java

Même si c'est rarement utilisé, les énumérations peuvent également contenir des attribut et des méthodes

```
public enum Direction {
    NORD, EST, SUD, OUEST;

    public Direction getOpposee() {
        switch (this) {
            case NORD:
                return SUD;
            case SUD:
                return NORD;
            case EST:
                return OUEST;
            case OUEST:
                return EST;
        }
    }
}
```

Direction.java



# Classe Object et méthode equals

Toute classe Java hérite de la classe Object et notamment des méthodes qui sont définies dans cette classe.

En particulier, la classe Object définit les méthodes toString (vue précédemment) et la méthode equals qui permet de vérifier l'égalité entre deux objets.

Par défaut, la méthode equals compare les références de deux objets, mais il peut être intéressant de la redéfinir pour que deux objets aux attributs identiques soient considérés égaux malgré leurs références distinctes.

Dans l'exemple ci-dessous, on considère deux personnes égales si elles ont le même nom et le même âge.

```
public class Personne {
    private String nom;
    private int age;

    public boolean equals(Personne personne) {
        if (this == personne) return true; // Egalite des references
        return this.age == personne.age && this.name.equals(personne.name); // Comparaison des
attributs
    }
}
```

Personne.java

# Programmation Orientée Objet (POO)

---

