Programmation Objet – Initiation à Java

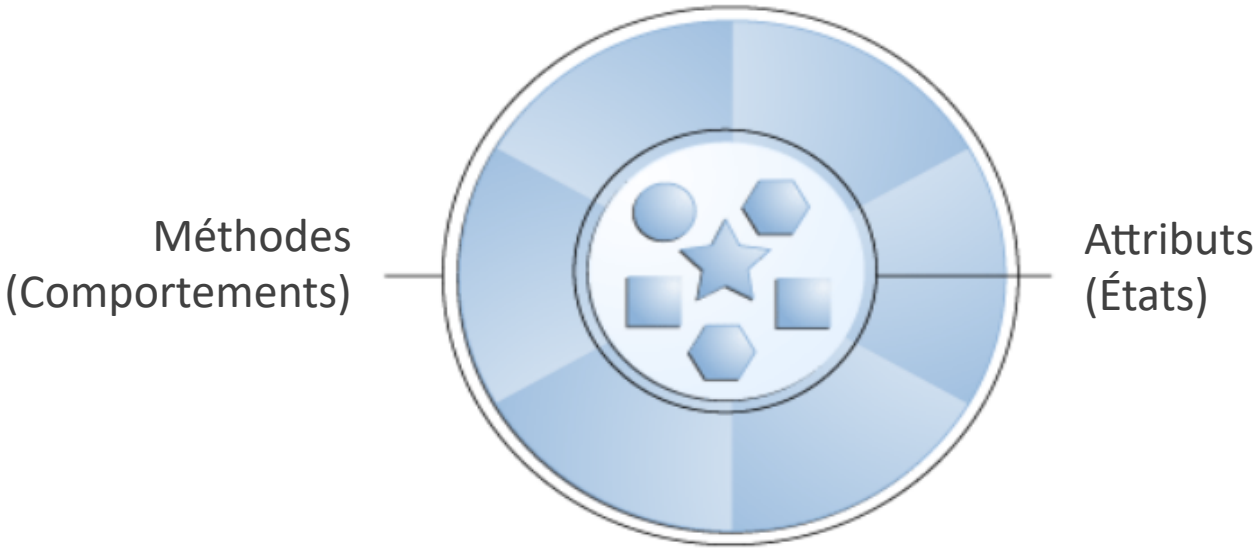
Letícia SEIXAS PEREIRA

Cours 05 – Concepts de base de l'OO

04/11/2019

La notion d'Objet

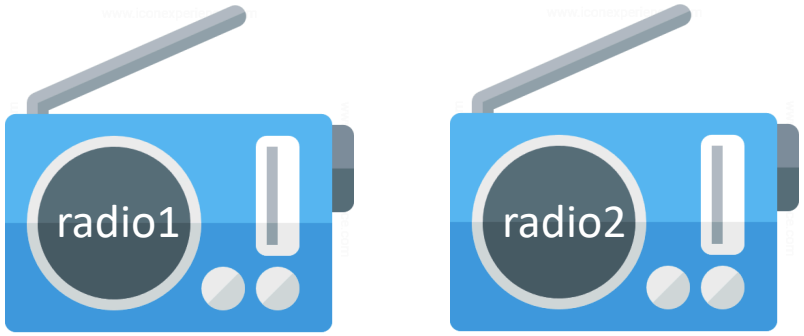
- L'idée de base de la programmation orientée objet est de rassembler dans une même entité appelée objet les ***données*** et les ***traitements*** qui s'y appliquent.



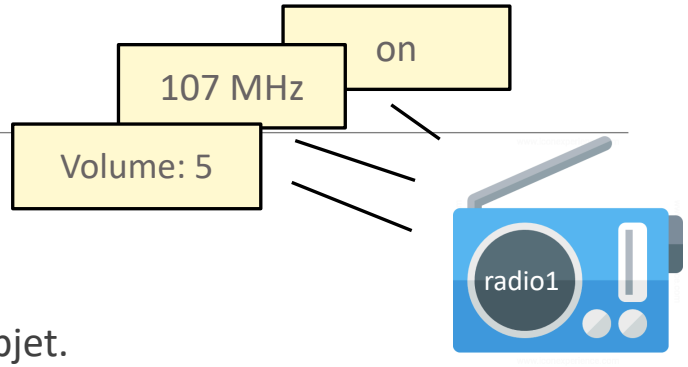
La notion d'Objet

- L'idée de base de la programmation orientée objet est de rassembler dans une même entité appelée objet les ***données*** et les ***traitements*** qui s'y appliquent.
- Un objet ou **instance** est une variable *presque* comme les autres. Il faut notamment qu'il soit déclaré avec son type.
- Le type d'un objet est un type complexe (par opposition aux types primitifs entier, caractère,…) qu'on appelle une classe.

```
class RadioDemo {

    public static void main(String[] args) {

        //Creer deux objets radios differents
        Radio radio1 = new Radio();
        Radio radio2 = new Radio();
    }
}
```



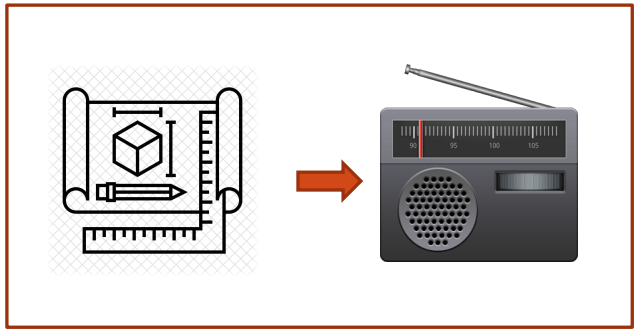
La notion d’Objet

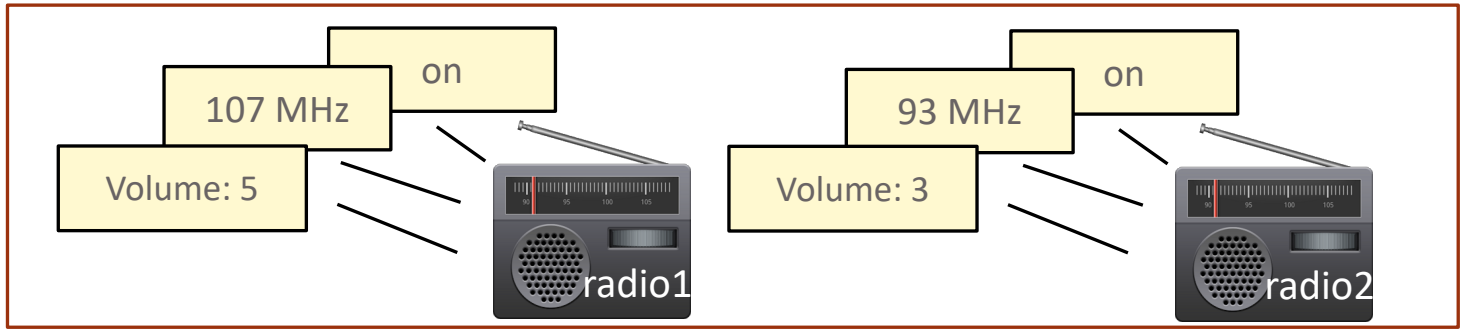


- Les objets contiennent des attributs (**variables d’instance**) et des méthodes.
- Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet.

La notion d'Objet

- Les objets contiennent des attributs (**variables d'instance**) et des méthodes.
- Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet.
- La classe est la description d'un objet.
- Un objet est une instance d'une classe.
- Pour chaque instance d'une classe, le code est le même, **seules les données sont différentes à chaque objet**.

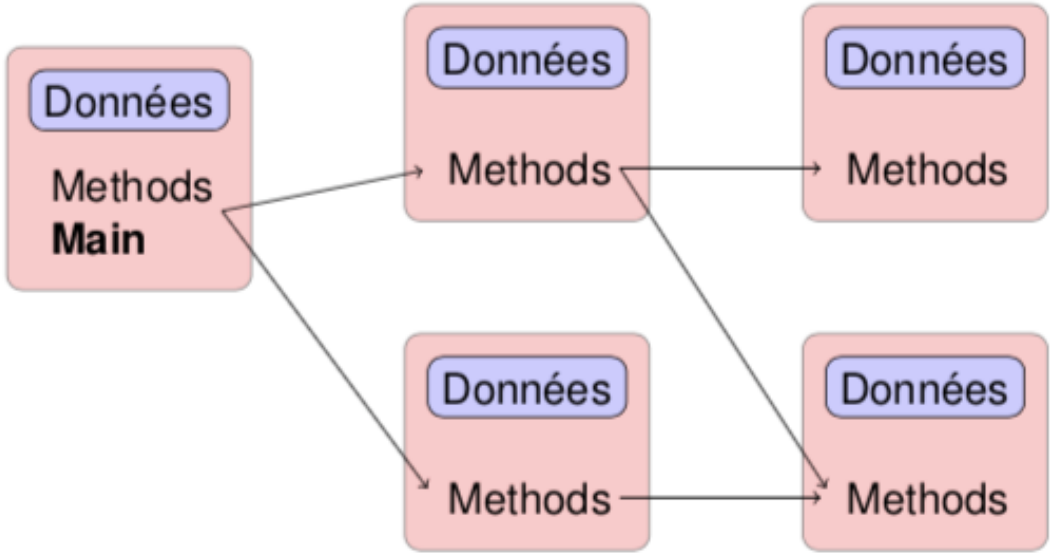




La notion d'Objet

Programmation Orientée Objet: Données et fonction de traitement de ces données sont réunies;

- Structurer le programme autour des objet manipulées;
- Associer les traitements à ces objets.



La notion d'Objet

- Attributs : les attributs décrivent les caractéristiques d'un objet

```
<type> <identificateur>
```

```
public class Rectangle {

     int longueur ;

     int largeur ;

     // ...

}
```

La notion d'Objet

- Attributs : les attributs décrivent les caractéristiques d'un objet

```
<type> <identificateur>
```

```
    int score,

    double pi;

    Rectangle r;
```

```
public class Rectangle {

     int longueur ;

     int largeur ;

     // ...

}
```

La notion d'Objet

- Méthodes : Les fonctions de traitement que l'on peut appliquer aux donnés des objets instances d'une classe :
  - Elles peuvent avoir des arguments:
    - de types primitifs;
    - objets.

```
<type> <nomMethode> (arguments...) {

        // Implémentation de la méthode

}
```

```
public class Rectangle {

     public Rectangle(int longueur, int largeur){

          this.longueur = longueur;

          this.largeur = largeur;

     }

     int surface () {

          return this.longueur * this.largeur ;

     }

}
```

La notion d'Objet

- Méthodes : Les fonctions de traitement que l'on peut appliquer aux donnés des objets instances d'une classe :
  - Elles peuvent avoir des arguments:
    - de types primitifs;
    - objets.

```
int fibo (int n) {
   if (n<2) return 1;
  return fibo(n-1) + fibo(n-2);
}
```

```
public class Rectangle {

     public Rectangle(int longueur, int largeur){

          this.longueur = longueur;

          this.largeur = largeur;

     }

     int surface () {

          return this.longueur * this.largeur ;

     }

}
```

La notion d'Objet

- Méthodes : Les fonctions de traitement que l'on peut appliquer aux donnés des objets instances d'une classe :
  - Elles peuvent avoir des arguments:
    - de types primitifs;
    - objets.

```
Matrice multiplie(Matrice m1, Matrice m2) {
        Matrice m = new Matrice();
        // ...
        return m;
}
```

```
public class Rectangle {
     public Rectangle(int longueur, int largeur){
          this.longueur = longueur;
          this.largeur = largeur;
     }
     int surface () {
          return this.longueur * this.largeur ;
     }
}
```

Concepts de base de l'OO

Concepts de base de l'OO

- Les principes fondamentaux de conception concernent tous les langages de programmation orientés objet et ne sont dons pas spécifiques à Java:
  - Java;
  - Python;
  - Ruby;
  - C++;
  - Objective C;
  - PHP…

Concepts de base de l'OO

1. Encapsulation
2. Héritage
3. Abstraction
4. Polymorphisme

Concepts de base de l'OO

Encapsulation

- Un objet porte en lui des données -> ces données sont *en principe* à l'usage exclusif de l'objet;
- Il est possible de *consulter* ou de *mettre à jour* ces données seulement si l'objet accepte de le faire:
  - Il faut *demander* à l'objet en lui envoyant un message de communiquer les données qu'il conserve, ou de les mettre à jour.

Concepts de base de l'OO

Encapsulation

- La visibilité des membres d'une classe:
  - Toutes les variables d'instance (attributs) d'une classe doivent être déclarés **private**

```
public class Rectangle {

    private int longueur ;

    private int largeur ;

    // ...

}
```

Visibilité	public	private
Dans la même classe	oui	oui
Dans une classe du même *package*	oui	**non**

Concepts de base de l'OO

```
public class Rectangle {

    public int longueur ;

    public int largeur ;

    public Rectangle(int longueur, int largeur){

        this.longueur = longueur;

        this.largeur = largeur;

    }

    int surface () {

        return this.longueur * this.largeur ;

    }

}
```

```
public class TestRectangle {

    public static void main (String[] args) {

        Rectangle rectangle1;

        rectangle1 = new Rectangle(3, 4);

        System.out.println(rectangle1.longueur);

    }

}
```

```
java TesteRectangle
3
```

Concepts de base de l'OO

```
public class Rectangle {

    private int longueur ;

    private int largeur ;

    public Rectangle(int longueur, int largeur){

        this.longueur = longueur;

        this.largeur = largeur;

    }

    int surface () {

        return this.

    }

}
```

```
public class TestRectangle {

    public static void main (String[] args) {

        Rectangle rectangle1;

        rectangle1 = new Rectangle(3, 4);

        System.out.println(rectangle1.longueur);

    }
```

```
javac TesteRectangle.java
TesteRectangle.java:9: error: longueur has private access in Rectangle
System.out.println(rectangle1.longueur);
                                  ^
1 error
```

Concepts de base de l'OO

Encapsulation

- Un objet porte en lui des données -> ces données sont *en principe* à l'usage exclusif de l'objet;
- Il est possible de *consulter* ou de *mettre à jour* ces données seulement si l'objet accepte de le faire:
  - Il faut *demander* à l'objet en lui envoyant un message de communiquer les données qu'il conserve, ou de les mettre à jour.

Il faut que l'objet soit conçu pour répondre à ces requêtes

Concepts de base de l'OO

Encapsulation : Accesseurs et mutateurs

Concepts de base de l'OO

Encapsulation : Accesseurs et mutateurs

- Un **accesseur** (*getter*) est une méthode qui va nous permettre d'accéder aux variables de nos objets en **lecture**;

- Les accesseurs ont toujours la forme:

```
public typeDeLAttribut getNomDeLAttribut () {
    return l'attributAAccéder;
}
```

Concepts de base de l'OO

Encapsulation : Accesseurs et mutateurs

- Un **accesseur** (*getter*) est une méthode qui va nous permettre d'accéder aux variables de nos objets en **lecture**;

- Les accesseurs ont toujours la forme:

```
public typeDeLAttribut getNomDeLAttribut () {
    return l'attributAAccéder;
}
```

```
private int longueur;
public int getLongueur(){
    return this.longueur;
}
```

Concepts de base de l'OO

Encapsulation : Accesseurs et mutateurs

- Un **accesseur** (*getter*) est une méthode qui va nous permettre d'accéder aux variables de nos objets en **lecture**;

- Les accesseurs ont toujours la forme:

```
public typeDeLAttribut getNomDeLAttribut () {
    return l'attributAAccéder;
}
```

```
private int longueur;
public int getLongueur(){
    return this.longueur;
}
```

Ne pas hésiter à utiliser this même lorsque ce n'est pas absolument nécessaire

Concepts de base de l'OO

Encapsulation : Accesseurs et mutateurs

- Un **mutateur** (*setter*) est une méthode qui va nous permettre d'accéder aux variables de nos objets en **écriture**;

```
public void setNomDeLAttribut (typeDeLAttribut identificateur) {
    attributAModifier = identificateur;
}
```

Concepts de base de l'OO

Encapsulation : Accesseurs et mutateurs

- Un **mutateur** (*setter*) est une méthode qui va nous permettre d'accéder aux variables de nos objets en **écriture**;

```
public void setNomDeLAttribut (typeDeLAttribut identificateur) {
    attributAModifier = identificateur;
}
```

```
private int longueur;
public void setLongueur(int l){
    this.longueur = l;
}
```

Ne pas hésiter à utiliser this même lorsque ce n'est pas absolument nécessaire

Concepts de base de l'OO

Encapsulation : Accesseurs et mutateurs : Exercice d'application

En utilisant la classe **Rectangle** comme base:

1. Créez les accesseurs et mutateurs pour les attributs de la classe Rectangle;
2. Dans la classe TesteRectangle, créez deux rectangles et comparez celui qui a la plus grande largeur.

```
public class Rectangle {

    private int longueur ;

    private int largeur ;

    public Rectangle(int longueur, int largeur){

        this.longueur = longueur;

        this.largeur = largeur;

    }

    int surface () {

        return this.longueur * this.largeur ;

    }

}
```

Concepts de base de l'OO

Encapsulation : La méthode toString()

```
public class TestRectangle {

    public static void main (String[] args) {

        Rectangle rectangle1;

        rectangle1 = new Rectangle(3, 4);

        System.out.println(rectangle1);

    }

}
```

```
java TesteRectangle
Rectangle@43556938
```

Concepts de base de l'OO

Encapsulation : La méthode toString()

- Celle-ci nous renvoie un objet de type String;
- Elle fait référence aux variables qui composent l'objet et nous renvoie une chaîne de caractères qui décrit l'objet.

```
public String toString (){

}
```

Concepts de base de l'OO

Encapsulation : La méthode toString()

- Celle-ci nous renvoie un objet de type St
- Elle fait référence aux variables qui comp

chaîne de caractères qui décrit l'objet.

```
public class TestRectangle {

    public static void main (String[] args) {

        Rectangle rectangle1;

        rectangle1 = new Rectangle(3, 4);

        System.out.println(rectangle1);

    }

}
```

```
java TesteRectangle
Rectangle 3x4
```

```
public class Rectangle {

    // ...

    public String toString(){

        return "Rectangle " + this.longueur + "x" + this.largeur;

    }

}
```

Concepts de base de l'OO

Encapsulation : La méthode toString()

En utilisant la classe **Rectangle** comme base:

1. Créez la méthode **toString()** pour afficher les attributs de la classe Rectangle;

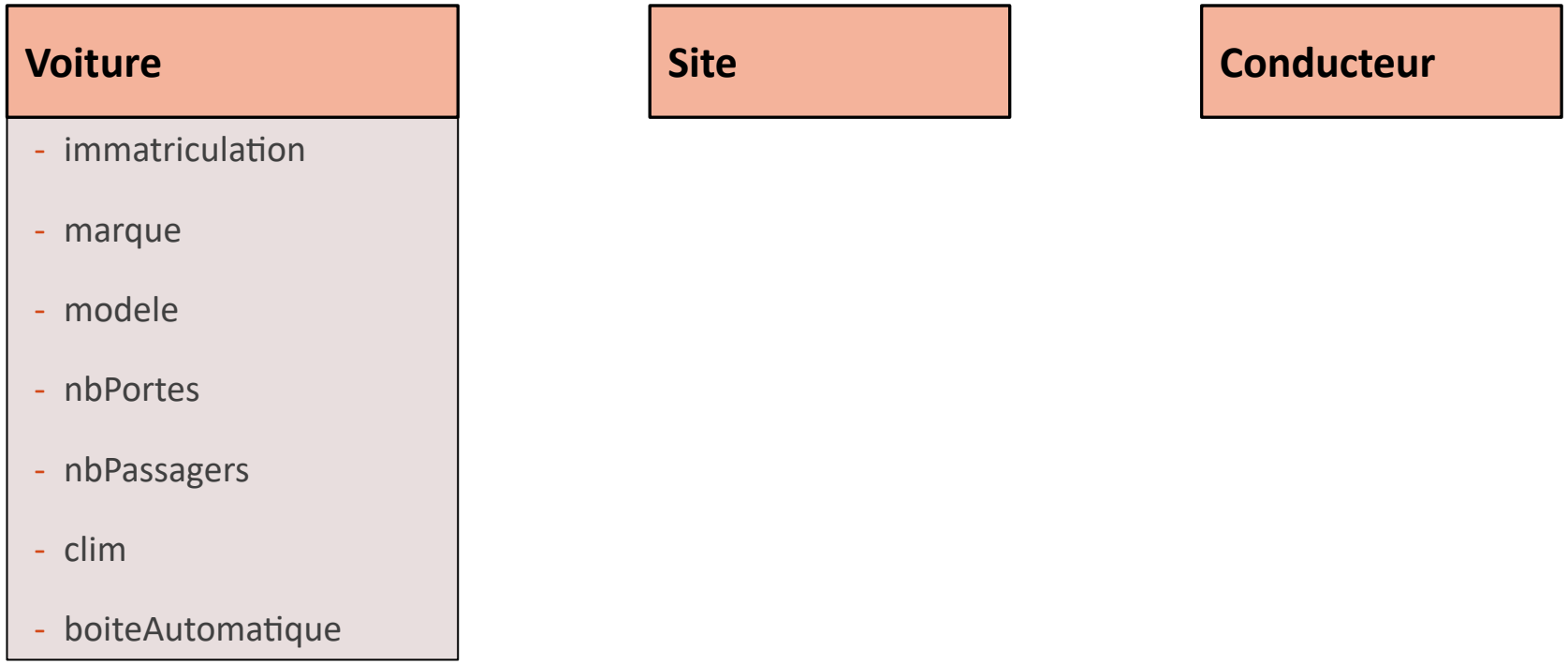
2. Dans la classe TesteRectangle, créez deux rectangles et affichez ses éléments.

```
public class Rectangle {

    // ...

    public String toString(){

        return "Rectangle " + this.longueur + "x" + this.largeur;

    }

}
```

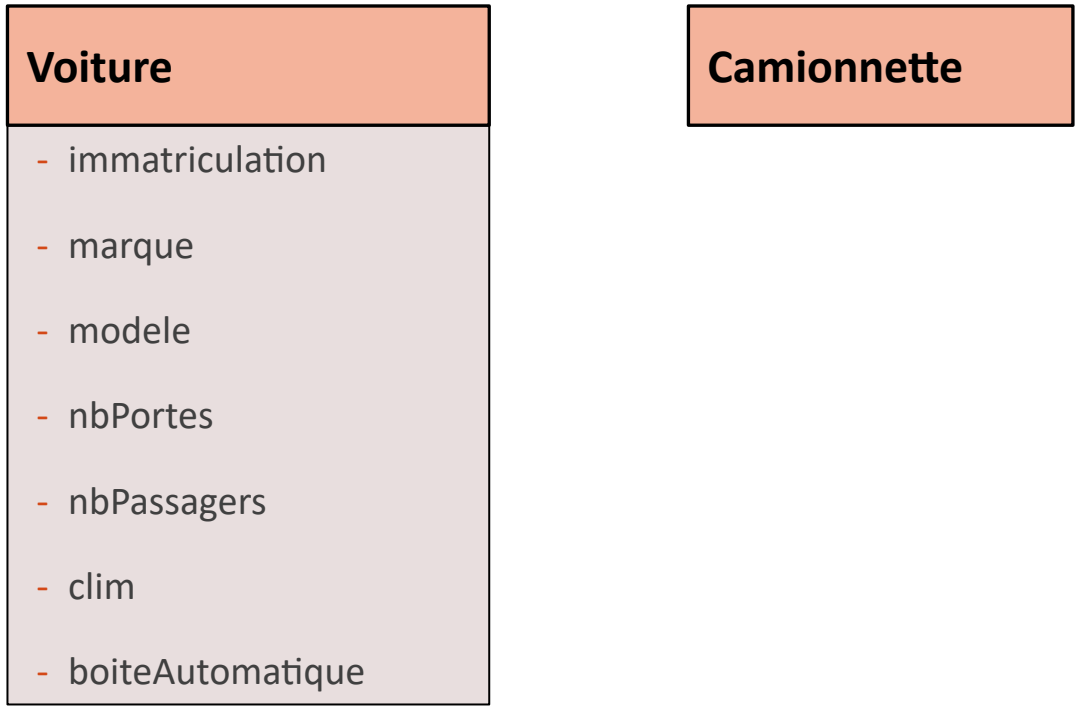
Concepts de base de l'OO

1. Encapsulation
2. **Héritage**
3. Abstraction
4. Polymorphisme

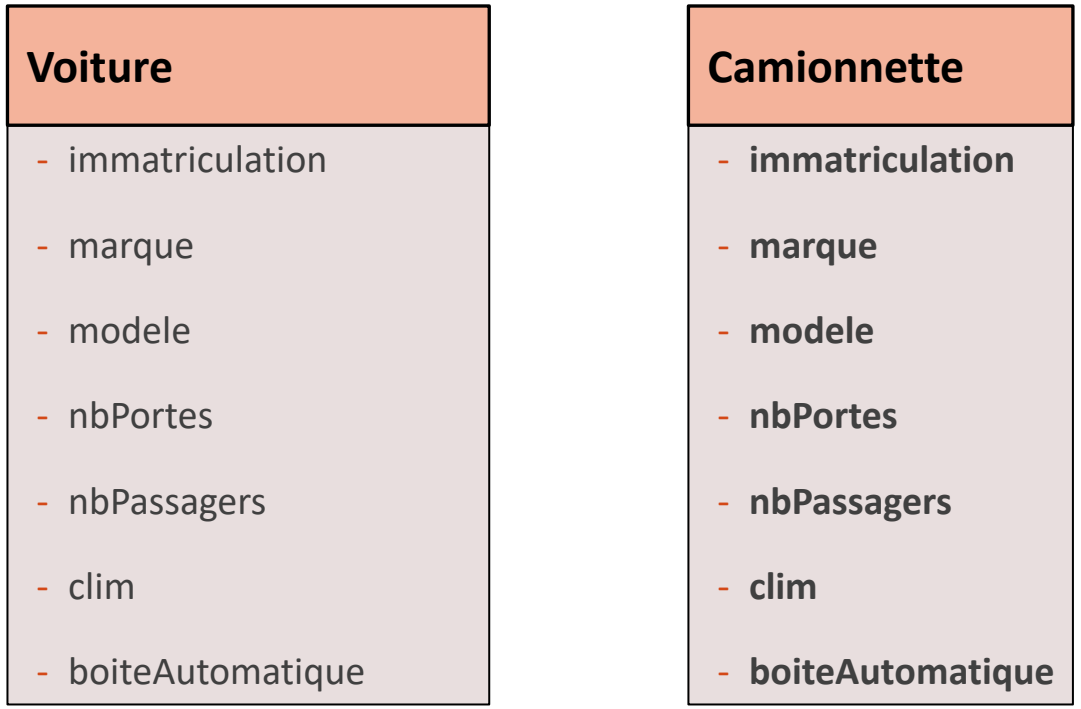
Héritage


Voiture
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
Site
Conducteur

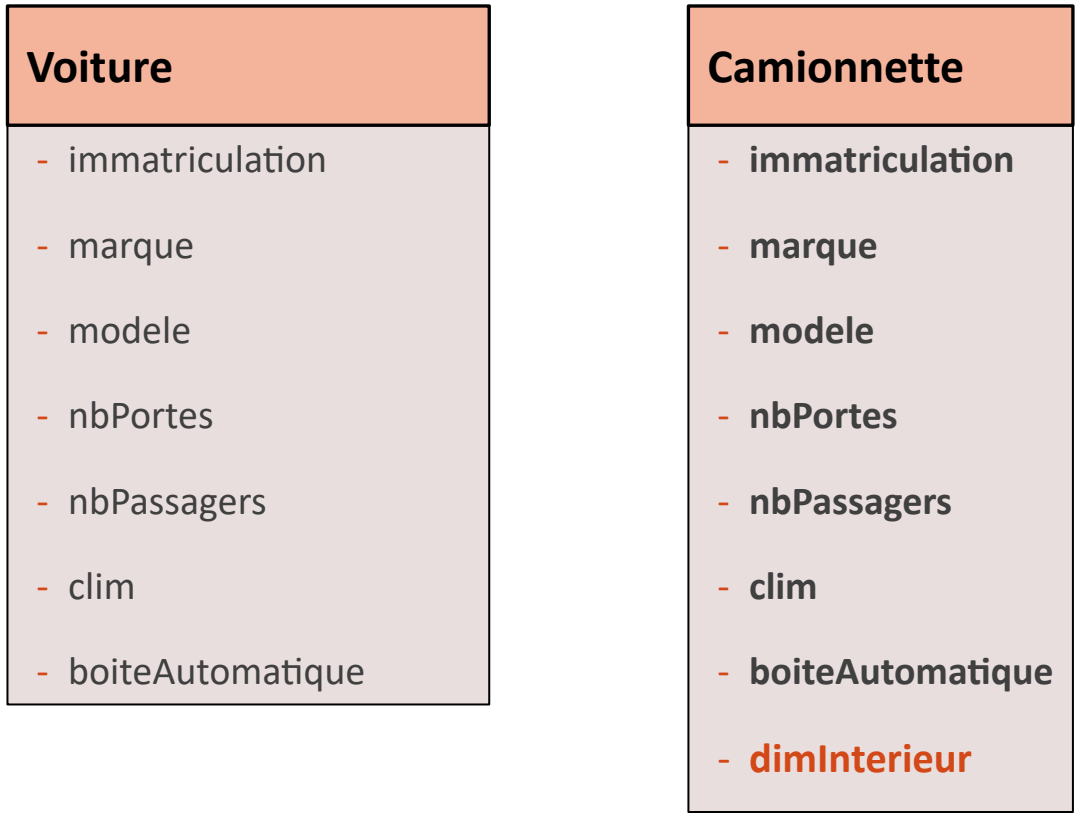
Héritage


Voiture
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
Camionnette

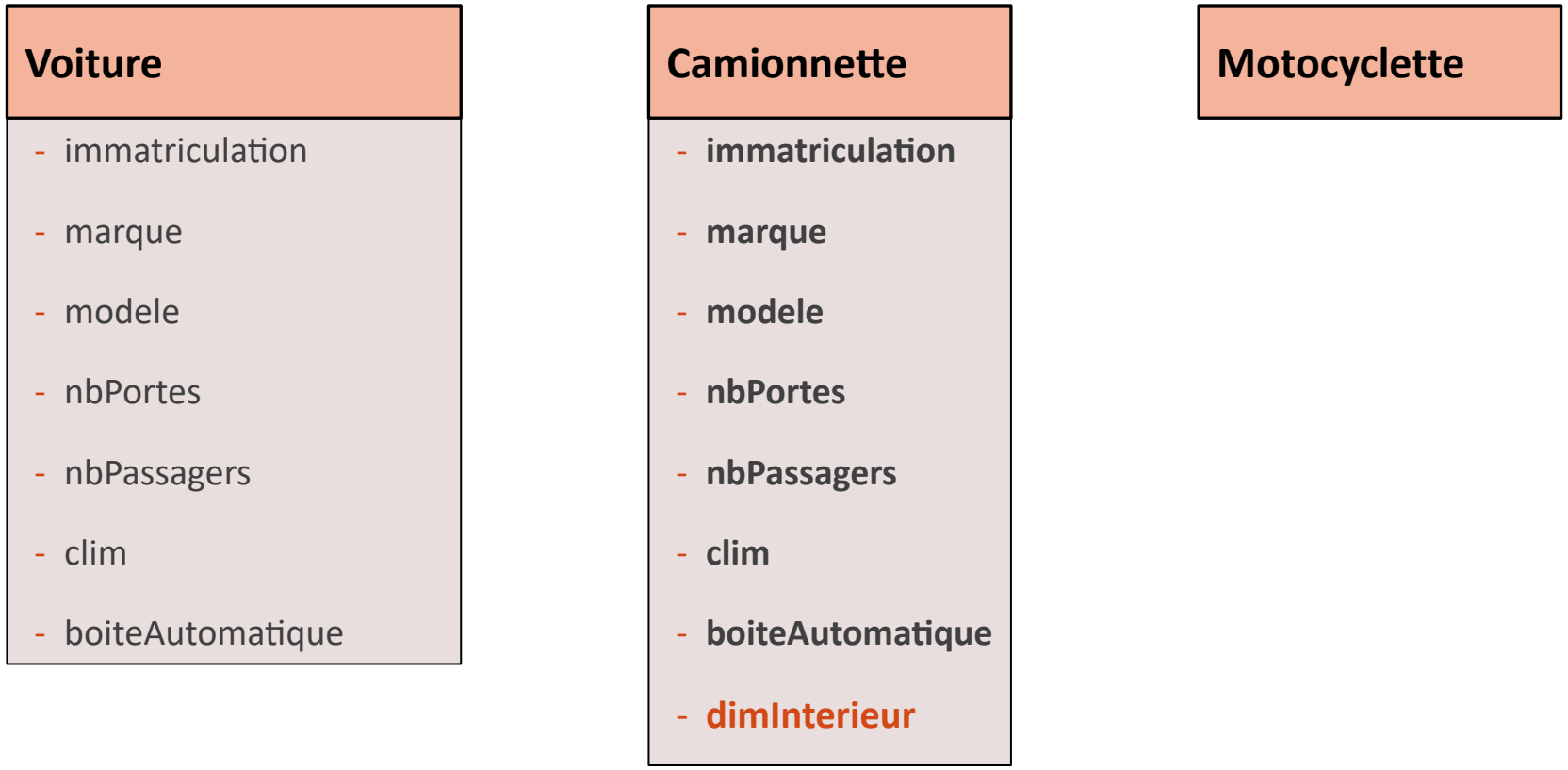
Héritage


Voiture
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
Camionnette
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique

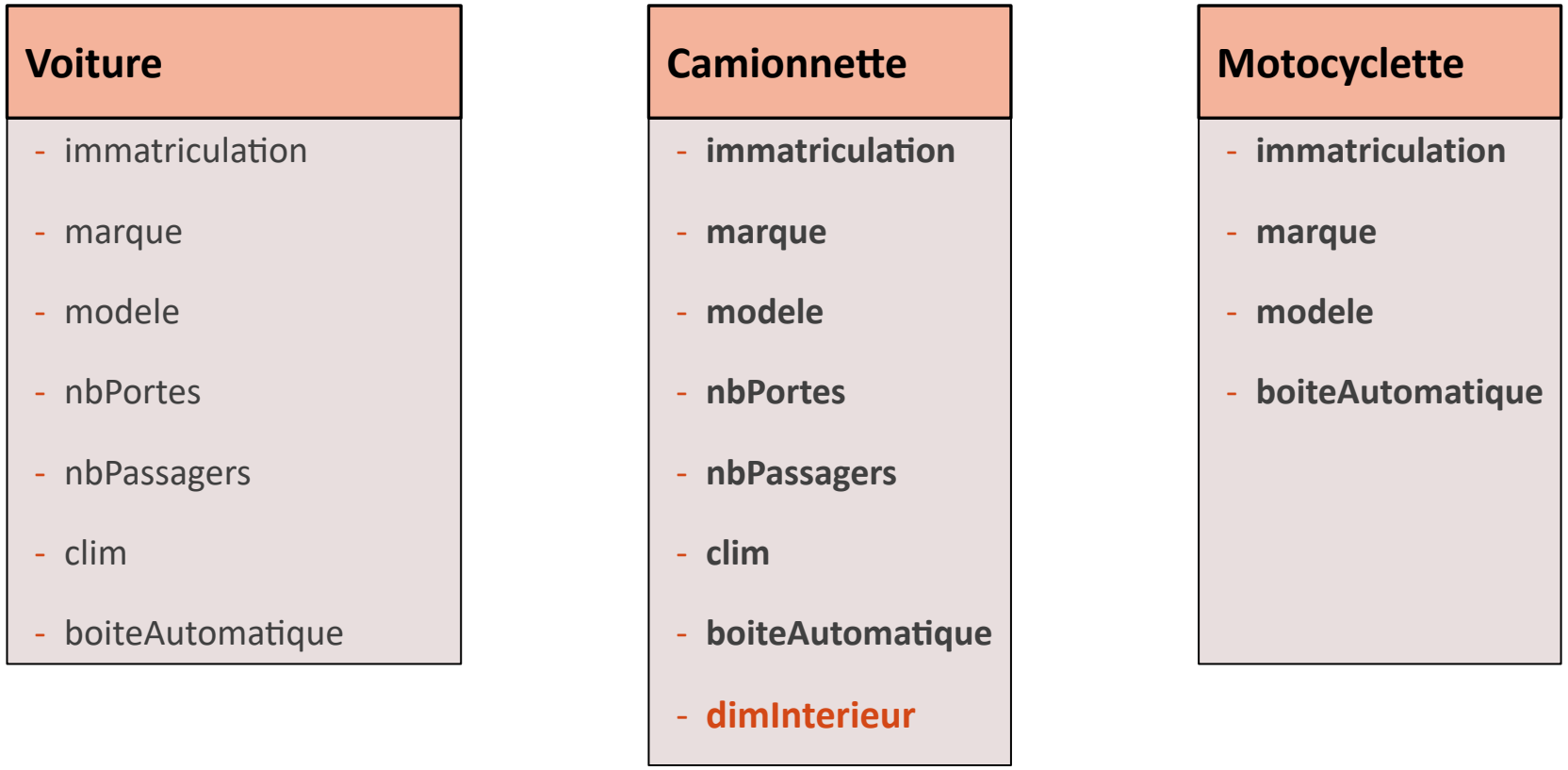
Héritage


Voiture
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
Camionnette
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
- dimInterieur

Héritage

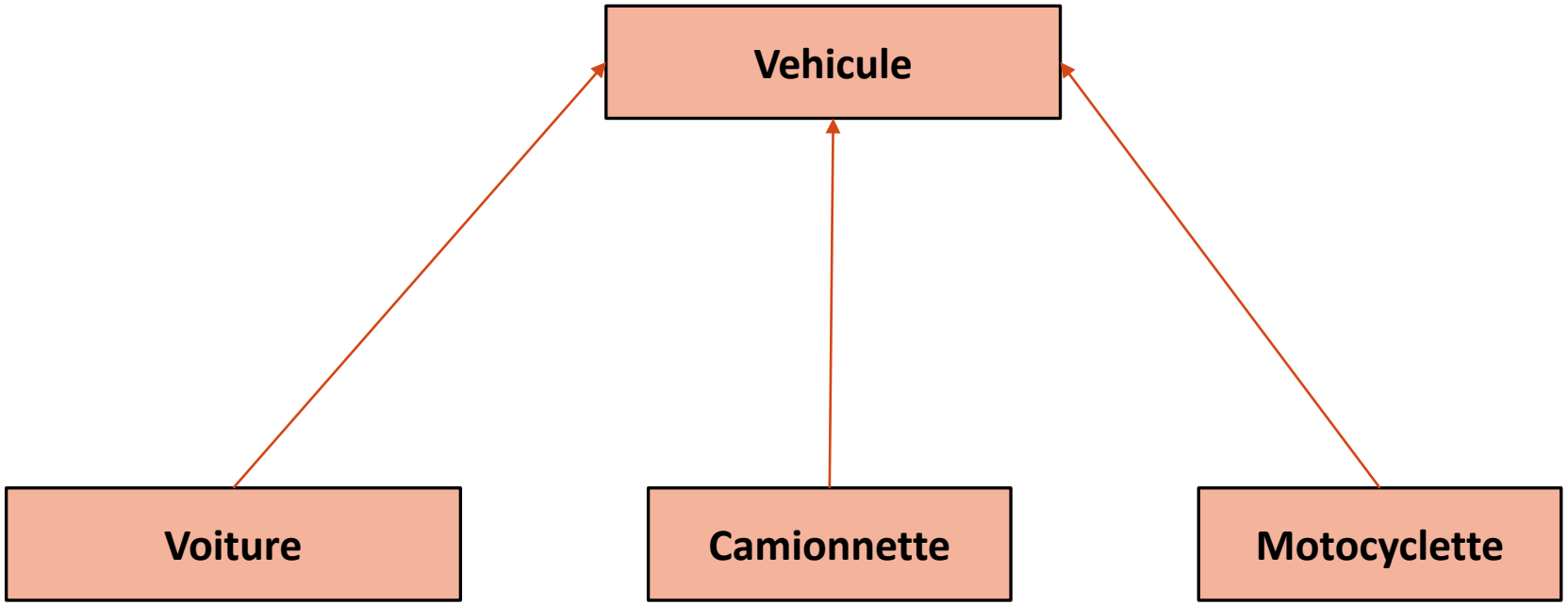

Voiture
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
Camionnette
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
- dimInterieur
Motocyclette

Héritage


Voiture
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
Camionnette
- immatriculation
- marque
- modele
- nbPortes
- nbPassagers
- clim
- boiteAutomatique
- dimInterieur
Motocyclette
- immatriculation
- marque
- modele
- boiteAutomatique

Héritage

Héritage



Héritage

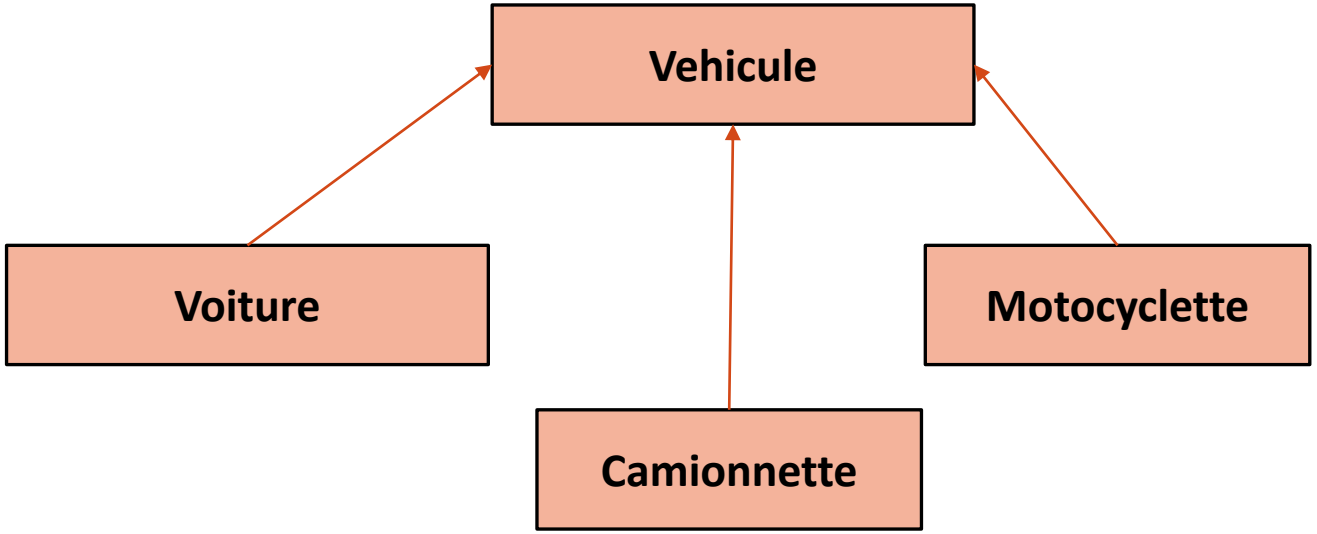
- La notion d'héritage est l'un des fondements de la programmation orientée objet;

Héritage

- La notion d'héritage est l'un des fondements de la programmation orientée objet;
- Une classe B **hérite** d'une classe A s'il existe une relation **« est un »** entre B et A.

Héritage

- La notion d'héritage est l'un des fondements de la programmation orientée objet;
- Une classe B **hérite** d'une classe A s'il existe une relation **« est un »** entre B et A.



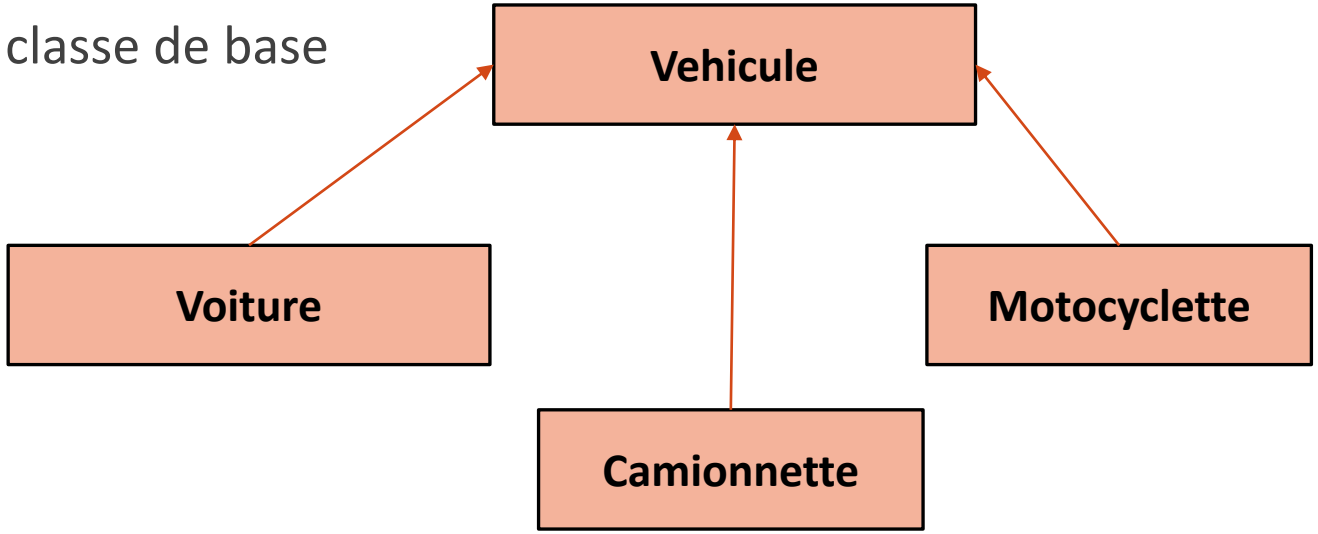
« Voiture **est un** véhicule »

« Camionnette **est un** véhicule »

« Motocyclette **est un** véhicule »

Héritage

- Voiture est une **sous-classe** de Vehicule;
  - sous-classe = classe fille = classe dérivée
- Vehicule est une **super-classe** de Voiture;
  - super-classe = classe mère = classe de base



Héritage

- C’est un mécanisme qui permet de définir une nouvelle classe à partir d’une classe existante;

Héritage

- C’est un mécanisme qui permet de définir une nouvelle classe à partir d’une classe existante;
- Une classe fille hérite de tous les composants de sa classe mère (attributs et méthodes);

Héritage

- C’est un mécanisme qui permet de définir une nouvelle classe à partir d’une classe existante;
- Une classe fille hérite de tous les composants de sa classe mère (attributs et méthodes);

  Cela permet :

  - d’étendre une classe en lui ajoutant des composants (attributs et/ou méthodes);

Héritage

- C’est un mécanisme qui permet de définir une nouvelle classe à partir d’une classe existante;
- Une classe fille hérite de tous les composants de sa classe mère (attributs et méthodes);

  Cela permet :
  - d’étendre une classe en lui ajoutant des composants (attributs et/ou méthodes);
  - de modifier le comportement d’une classe sans modifier la classe de base (= **redéfinir** les méthodes héritées);

Héritage

- C’est un mécanisme qui permet de définir une nouvelle classe à partir d’une classe existante;
- Une classe fille hérite de tous les composants de sa classe mère (attributs et méthodes);

  Cela permet :
  - d’étendre une classe en lui ajoutant des composants (attributs et/ou méthodes);
  - de modifier le comportement d’une classe sans modifier la classe de base (= **redéfinir** les méthodes héritées)
    - (!) les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.

Héritage

- Une classe peut avoir plusieurs sous-classes, par contre, une classe ne peut avoir qu'une seule classe mère;
  - (Il n'y a pas d'héritage multiple en Java)

Héritage

- Une classe peut avoir plusieurs sous-classes, par contre, une classe ne peut avoir qu'une seule classe mère;
  - (Il n'y a pas d'héritage multiple en Java)
- On dit qu'en Java tout est objet : Toute les classes héritent (implicitement) de la classe **Objet**, qui est la classe mère de toutes les classes.

Héritage

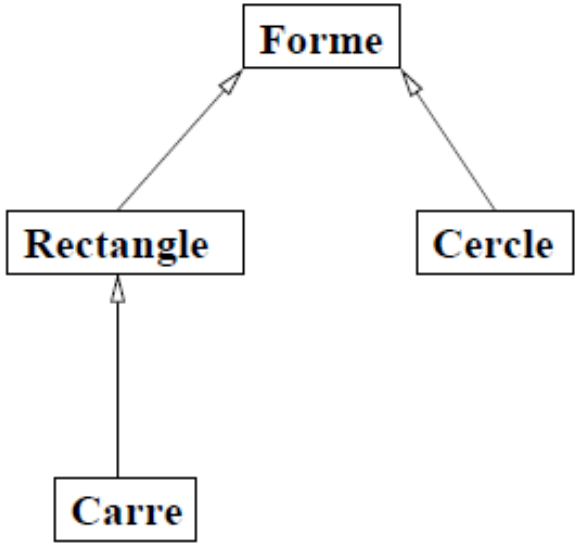
- **Déclaration:**

```
class SousClasse extends SuperClass
```

Héritage

- **Déclaration:**

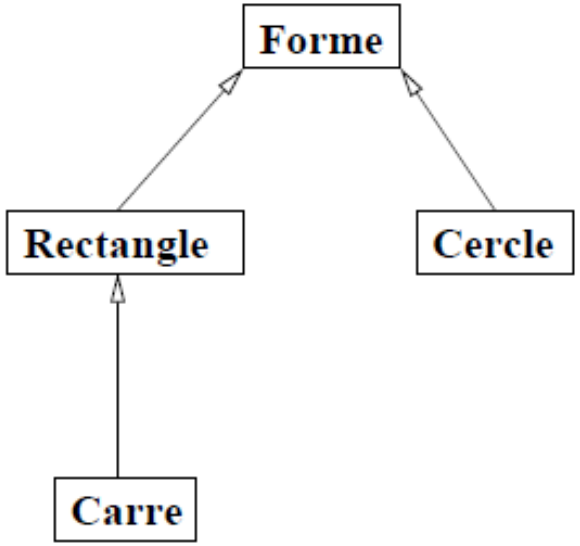
```
class SousClasse extends SuperClass
```



Héritage

- **Déclaration:**

```
class SousClasse extends SuperClass
```

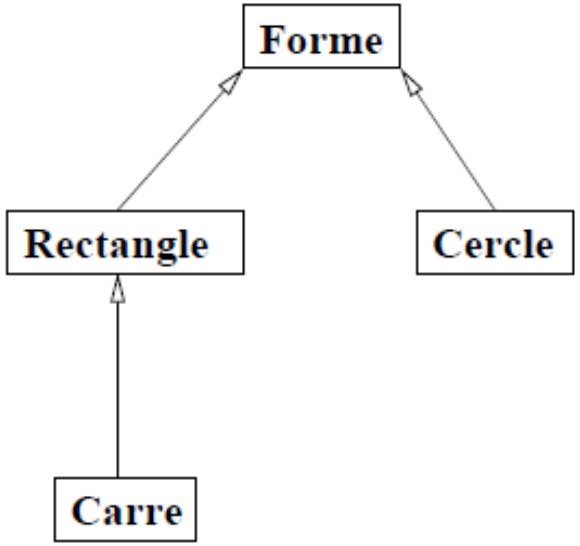



```
public class Cercle extends Forme{
        //...
}
public class Rectangle extends Forme{
        //...
}
```

Héritage

- **Déclaration:**

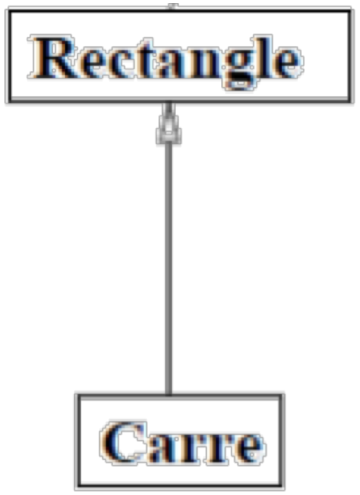
```
class SousClasse extends SuperClass
```




```
public class Carre extends Rectangle {
        //...
}
```

Héritage

- Exemple: Rectangle.java

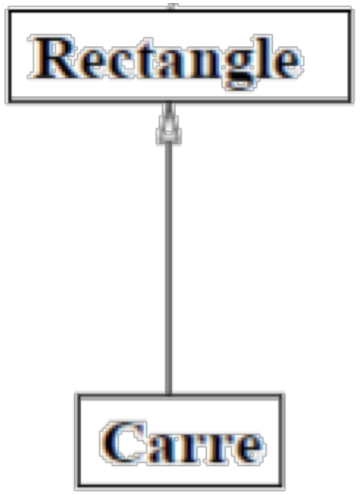


Héritage

- Pour appeler le constructeur de la classe mère, il suffit d'écrire **super (**<paramètres>**)**;
  - En Java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.
- Pour invoquer une méthode d'une classe mère, il suffit d'indiquer la méthode préfixée par **super**.

Héritage

- Exemple: Carre.java



Héritage

- L’appel au constructeur d’une classe supérieure doit toujours se situer dans un constructeur et toujours **en tant que première instruction** ;

Héritage

- L'appel au constructeur d'une classe supérieure doit toujours se situer dans un constructeur et toujours **en tant que première instruction** ;
- Si aucun appel à un constructeur d'une classe fille n'est fait, le constructeur fait appel implicitement à un **constructeur vide** de la classe supérieure (comme si la ligne super() était présente);

Héritage

- L'appel au constructeur d'une classe supérieure doit toujours se situer dans un constructeur et toujours **en tant que première instruction** ;
- Si aucun appel à un constructeur d'une classe fille n'est fait, le constructeur fait appel implicitement à un **constructeur vide** de la classe supérieure (comme si la ligne super() était présente);
  - Si aucun constructeur vide n'est accessible dans la classe supérieure, une erreur se produit lors de la compilation.

Héritage- Accès

Héritage- Accès

```
public class Rectangle {
        private int longueur ;
        private int largeur ;
         public Rectangle (int longueur , int largeur )
        {
        //...
        }
}
```

Héritage- Accès

- **Modificateurs d'accès:**
  - public
  - private
  - protected

Héritage- Accès

- **Modificateurs d'accès:**
  - **public:** Les variables et méthodes restent publiques à traves

    l'héritage et toutes les autres classes peuvent y accéder;

Héritage- Accès

- **Modificateurs d'accès:**
  - **public:** Les variables et méthodes restent publiques à traves l'héritage et toutes les autres classes peuvent y accéder;
  - **private:** Les variables et méthodes ne sont pas accessible directement;

Héritage- Accès

- **Modificateurs d'accès:**
  - **public:** Les variables et méthodes restent publiques à traves l'héritage et toutes les autres classes peuvent y accéder;
  - **private:** Les variables et méthodes ne sont pas accessible directement;
  - **protected:** Les variables et méthodes sont accessible que pour les classes filles.

Héritage- Accès

- Pour invoquer une méthode d'une classe mère, il suffit d'indiquer la méthode préfixé par **<super>**:

```
public class Carre extends Rectangle {

        //...

        System.out.println(super.surface ());

}
```

Héritage- Accès

- Ordre d'appel des constructeurs

```
public class MainFormes{

   public static void main(String[] args){

            Carre carre = new Carre(2);

             carre.afficher();

   }

}
```

Héritage- Accès

- Ordre d'appel des constructeurs

```
public class MainFormes{

    public static void main(String[] args){

                Carre carre = new Carre(2);

                 carre.afficher();

    }

}
```

```
Appel du constructeur de la classe Rectangle
Appel du constructeur de la classe Carre
4
Rectangle 2x2
```

- La méthode afficher écrit le mot “Rectangle” en début de chaîne. Il serait souhaitable que ce soit “carré” qui s’affiche;
- Le problème peut être résolu par une **redéfinition de la méthode** afficher dans la classe Carre.

Héritage- Accès

- Redéfinition des méthodes héritées
  - Une méthode d'une sous-classe redéfinit une méthode de sa classe supérieure, si elles ont la **même signature** mais que le **traitement effectué est réécrit** dans la sous-classe:

Héritage- Accès

- Redéfinition des méthodes héritées
  - Une méthode d'une sous-classe redéfinit une méthode de sa classe supérieure, si elles ont la **même signature** mais que le **traitement effectué est réécrit** dans la sous-classe.

```
public class Carre extends Rectangle {
                //...
    public void afficher () {
                System.out.println("Carre " + this.getLongueur());
    }
}
```

Héritage- Accès

- Redéfinition des méthodes héritées
  - Il est encore possible d'accéder à la méthode redéfinie dans la classe supérieure lors de la redéfinition d'une méthode:
    - En utilisant le mot-clé **<super>** comme préfixe: **super.afficher();**

Héritage- Accès

- Redéfinition des méthodes héritées
  - Il est encore possible d'accéder à la méthode redéfinie dans la classe supérieure lors de la redéfinition d'une méthode:
    - En utilisant le mot-clé **<super>** comme préfixe: **super.afficher();**
  - Il est possible d'interdite la redéfinition d'une méthode ou d'une variable:
    - En utilisant le mot-clé **<final>** au début d'une signature de méthode: **final public int surface();**

Héritage- Accès

- Redéfinition des méthodes héritées
  - Il est encore possible d'accéder à la méthode redéfinie dans la classe supérieure lors de la redéfinition d'une méthode:
    - En utilisant le mot-clé **<super>** comme préfixe: **super.afficher();**
  - Il est possible d'interdite la redéfinition d'une méthode ou d'une variable:
    - En utilisant le mot-clé **<final>** au début d'une signature de méthode: **final public int surface();**
  - Il est possible d'interdite l'héritage d'une classe:
    - En utilisant le mot-clé **<final>** au début de la déclaration d'une classe: **final classe Carre extends Rectangle;**

Héritage- Accès

- Redéfinition des méthodes héritées

```
public final class Rectangle {
    private int longueur ;
    private int largeur ;
    public Rectangle (int longueur , int largeur ) {
        this.longueur = longueur;
        this.largeur = largeur ;
    }
    public int surface () {
        return this.longueur * this.largeur ;
    }
}
```

Héritage - Accès

- Redéfinition des méthodes héritées

```
public final class Rectangle {
    private int longueur ;
    private int largeur ;
    public Rectangle (int longueur , int largeur ) {
        this.longueur = longueur;
        this.largeur = largeur ;
    }
    public int surface () {
        return this.longueur * this.largeur ;
    }
}
```

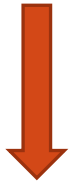
```
public class Carre extends Rectangle{
    //…
}
```

Héritage- Accès

- Redéfinition des méthodes héritées

```
public final class Rectangle {
    private int longueur ;
    private int largeur ;
    public Rectangle (int longueur , int largeur ) {
        this.longueur = longueur;
        this.largeur = largeur ;
    }
    public int surface () {
        return this.longueur * this.largeur ;
    }
}
```

```
public class Carre extends Rectangle{
    //…
}
```



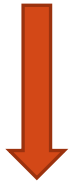
« error : cannot inherit from final Rectangle »

Héritage- Accès

- Redéfinition des méthodes héritées

```
public class Rectangle {
    private int longueur ;
    private int largeur ;
    public Rectangle (int longueur , int largeur ) {
        this.longueur = longueur;
        this.largeur = largeur ;
    }
    public final int surface () {
        return this.longueur * this.largeur ;
    }
}
```

```
public class Carre extends Rectangle{
    //…
    public int surface(){
        //…
    }
}
```



« error : surface() in Carre cannot override surface() in Rectangle … overridden method is final »