Rappel :
les classes et les objets

Pourquoi la Programmation Orientée-Objets (POO) ?

- Organiser les programmes de façon plus efficace grâce à :
  - Notions d'encapsulation et d'abstraction
  - Notions d'héritage et de polymorphisme
- Modéliser un système réel
- Réutiliser le code : coder deux fois, c’est mal coder !
- Améliorer la lisibilité et la pérennité de votre code

Notion d’encapsulation

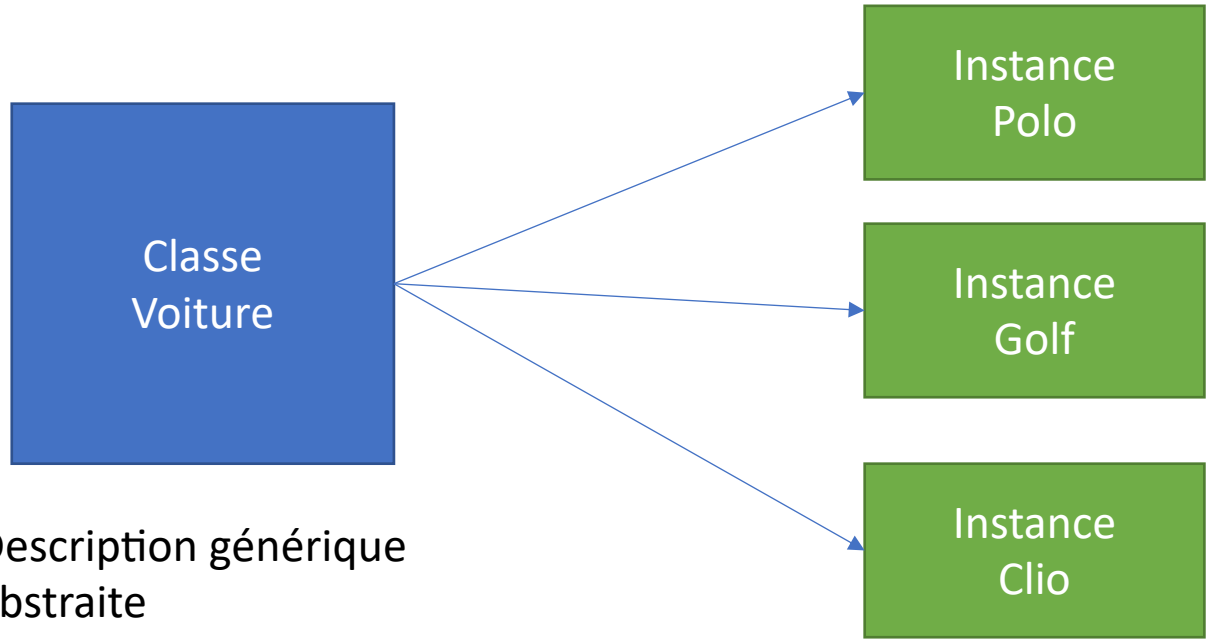
- Regrouper dans un même élément les données avec les traitements (les fonctions) qui les utilisent
- Cet élément informatique est appelé objet
  - Les données sont appelées attributs
  - Les fonctions de manipulation sont appelées méthodes
  - **Objet = attribut + méthodes**
- Une classe regroupe par exemple toutes les opérations et données nécessaires à la création à la représentation et à la manipulation de voitures, vélos, livres, factures...
- Permet de définir différents niveaux de perception
  - Niveau externe
    - Perception de l'objet depuis l'extérieur, partie visible, partie publique
    - Interface de l'objet avec l'extérieur
  - Niveau interne
    - Perception de l'objet depuis l'intérieur, partie privée
    - Correspond à l'implémentation de l'objet

Notion d’abstraction

- Identifier pour un ensemble donné d'éléments
  - Des caractéristiques valides pour la totalité de ces éléments
  - Des mécanismes commun à la totalité de ces éléments
- Concept abstrait de "voiture", toutes les voitures ont à un moment donné :
  - une marque, une couleur, une vitesse, une direction, un rapport...
  - on peut freiner, accélérer, tourner, faire le plein...

Qu’est-ce qu’une classe ?

- Résultat du processus d'abstraction
- Factorise les caractérisations communes
- C'est un moule... et un créateur d'instances



Description générique abstraite

Classe = attributs + méthodes + mécanisme d’instanciation

Différences entre classes et objets

- Classe : Modèle, prototype d'objet plutôt qu'un objet lui même...
- Objet, instance de classe : objet réel, utilisable, manipulable
- Même différence qu'entre une recette de clafoutis et un clafoutis...
  - A partir d'une recette de clafoutis vous pouvez faire autant de clafoutis que vous voulez ! Par contre vous ne pouvez pas manger une recette de clafoutis !
- Qu'est ce qu'un objet ?
  - Une instance de classe
  - Un objet réagit à certains message qu'on lui envoie et sa façon de réagir, son comportement dépend de l'état dans lequel il est

instance = valeurs d'attributs + accès aux méthodes

La méthode

- Une méthode est un moyen d’interaction et de communication entre les objets (simples ou paramétrés)

Pour résumer…

Attributs

Méthodes

Voiture
couleur puissance modele
accelerer() freiner() tourner(direction) klaxonner()

Classe Voiture

Instanciation

MaClio:Voiture
couleur = rouge puissance = 110 modele = Clio
accelerer() freiner() tourner(direction) klaxonner()

Objet Clio

Exemple de classe : Voiture

```java
public class Voiture {
        // Variables d'instance
        private String modele;
        private String couleur;
        private int puissance;
        // Constructeur
        public Voiture(String modele, String couleur, int puissance) {
                this.modele = modele;
                this.couleur = couleur;
                this.puissance = puissance;
        }
        /// Méthodes de classes ///
        // Accesseur ou Getter
        public String getModele() {
                return modele;
        }
        // Modification ou Setter
        public void setModele(String modele) {
                this.modele = modele;
        }
}
```

Définition d’une classe

- Modificateur de classe
  - abstract : classe abstraite, non instanciable
  - final : classe non dérivable
  - public : visible et accessible par tous
  - default ou rien : visible et accessible aux classes du package et aux classes filles

```
[modificateur]class NomClasse
                    [extends ClasseMère]
                    [implements Interface]{
}
```

Les constructeurs

- Ils sont appelés à la création de l'objet
- Permettent de créer de nouvelles instances et leur donner un état initial
- Ils portent le même nom que la classe
- On peut définir (surcharger) plusieurs constructeurs :
  - `public Voiture(){…}`
  - `public Voiture(int puissance){…}`
  - `public Voiture(String modele){…}`

Le mot clé **this**

- Lever une ambiguïté de nommage entre une variable locale et une variable de classe :

```
public Voiture(String modele, String couleur, int puissance) {
                this.modele = modele;
}
```

- Passer une référence à l'instance courante dans un appel de méthode :

```
public void trace(){System.out.println(this);}
```

- Invoquer un autre constructeur de la classe dans la définition du constructeur :

```
public Voiture(String modele, String couleur, int puissance) {
                this.modele = modele;
}

public Voiture() {
                this(modele, couleur, puissance);
}
```

La signature d'une méthode

- Signature : nom, nombre et type des paramètres
  - Il se peut qu'il n'y ait aucun paramètre, on a alors `methode()`
- Type de retour : primitif ou objet, simple ou complexe, ou rien (void)
  - Pour renvoyer quelque chose (ou rien) on utilise le mot clé `return`
- Si vous avez spécifié un type de retour différent de void pour votre méthode, elle doit retourner ce type dans tous les cas. :

```
public boolean superieur(int n1, n2){
        if (n1 > n2) return true;
} // Erreur de compilation, pas de valeur de retour dans tous les
     cas
```

Surcharge des méthodes

- Une méthode peut-être définie avec des signatures différentes (nombres et types d'arguments) :
  - `public int addition(int a, int b) { };`
  - `public int addition(int tab[]) { };`
- Les constructeurs peuvent aussi être surchargés
  - `public maClasse(int i){ };`
  - `public maClasse(String s){ };`

La portée des variables

- Variable d’instances
  - Déclarées en dehors de toute méthode
  - Modélisent l’état d’un objet
  - Elles sont accessibles par toutes les méthodes de la classe
  - On parle aussi de membres ou d’attributs d’une classe
  - Leur initialisation n’est pas obligatoire
- Variables locales (à une méthode)
  - Déclarées à l’intérieur d’une méthode
  - Accessible dans le bloc ou elles ont été déclarées
  - Doit obligatoirement être initialisées

Le mot-clé **static**

- Variables de classes statiques :
  - Variables qui sont partagées par toutes les instances de la classe
  - Déclaration : `public static double PI = 3.14;`
  - Invocation : `double pi = MonMath.PI`
- Méthodes de classes statiques :
  - Méthodes qui exécutent une action indépendante d'une instance particulière
  - Déclaration : `public static double getPI();`
  - Invocation : `double pi = MonMath.getPI();`
- Les blocs de code statiques :
  - Permet d'initialiser des variables statiques complexes. Ils sont exécutés une seule fois au chargement de la classe en mémoire :
  - `static { int i = 10000; }`

Critère d’accès des variables et méthodes

- Sécuriser le code en maîtrisant l’accès aux champs d’un objet.
- Faciliter la réutilisation du code
- Les modificateurs :
  - private : accès réduit, seulement depuis la classe
    - Pour partager la variable, employer les getters et setters
  - protected : accès depuis la classe, les classes filles et les classes du package
  - public : accès libre depuis partout
  - package ou rien : accès depuis la classe et les classes du package

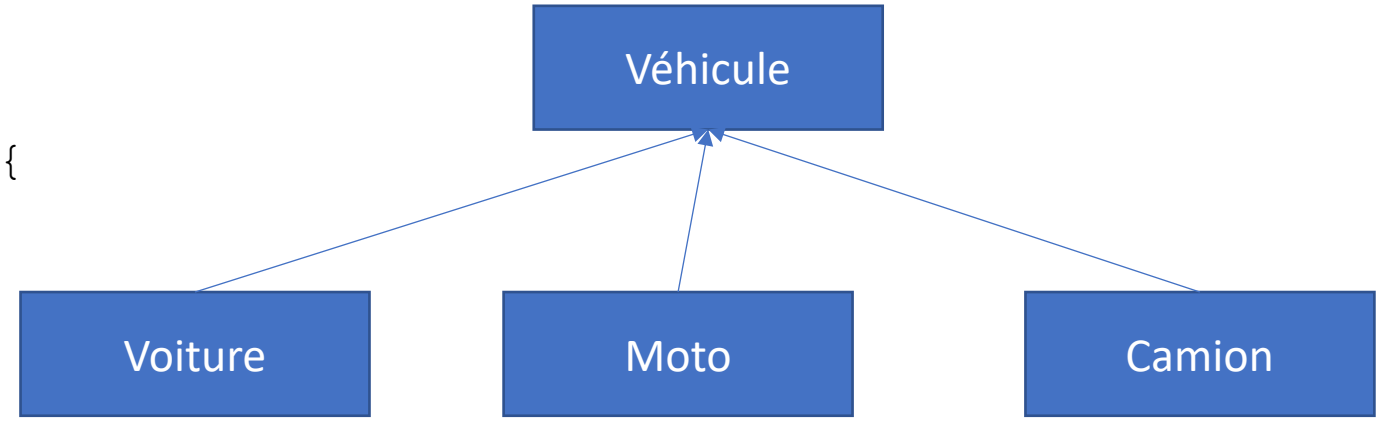
Le mot clé **final**

- Indique que la valeur de la variable (d'instance, de classe, ou locale) ne peut être modifiée
- Elle ne peut recevoir de valeur qu'une seule fois : à la déclaration ou plus tard
- Une variable déclarée finale et statique doit être initialisée à la déclaration et ne peut plus être modifiée ensuite
- Exemple : `static final double PI = 3.1459;`
- On peut utiliser le mot clé final pour bloquer la valeur d'un paramètre : `int methodeTest(final int i){...} // i est inchangé`

L’héritage

- L’héritage consiste à créer une nouvelle classe « fille » héritant des caractéristiques (attributs et méthodes) de sa classe « mère »
- Nous pouvons définir de nouveaux attributs et méthode sur cette classe fille, en plus de celles héritées par la classe mère.
- Nous pouvons redéfinir (spécialiser) les méthodes dont elle hérite
- Avantage : rationnaliser le temps de codage en réutilisant des méthodes déjà développées et testée

```
public class Vehicule{
}
public class Voiture extends Vehicule{
}
```



L’héritage : redéfinition des méthodes

- La sous-classe peut redéfinir les méthodes héritées en réécrivant la méthode en conservant la même signature (nom + paramètres) et le même type de retour mais en modifiant le code, en utilisant le mot clé **super**

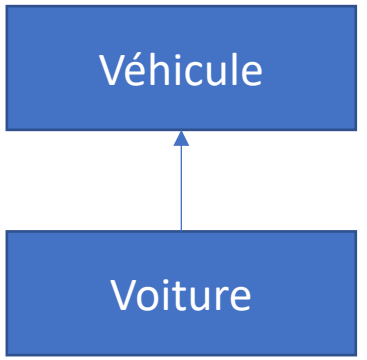
```
public class Vehicule{
	public void rouler (){ }
}
public class Voiture extends Vehicule{
	// L’action de rouler d’une voiture est différente d’un véhicule
	public void rouler() {
		super.rouler(); // Appel à la méthode la classe mère
		// Faire autre chose….
	}
}
```

L’héritage : redéfinition du constructeur

- Pour construire un objet d’une sous classe (un objet spécialisé) il faut partir d’un objet de la classe Mère, la super classe (l’objet général)...
  - La première instruction du constructeur doit être un appel à un constructeur de la classe Mère ou à un autre constructeur de la classe (`this(...)`)
  - Invocation via le mot clé `super` :
- `super(); // invocation du constructeur sans arguments`
- Ou
- `super(par1, par2, ..., parn) // ; constucteur à plusieurs arguments`

Polymorphisme

- Un objet peut être vu comme une instance de n'importe laquelle des classe héritées de la classe dont il est l'instance
- Le polymorphisme permet d'écrire :
  - `Voiture clio = new Vehicule();`
- L'objet `clio` utilisera les méthodes et attributs de `Voiture` s'ils ont été redéfinis dans la classe Voiture, sinon les méthodes et attributs de la classe `Vehicule`, sinon les méthode et attributs la classe mère, et ainsi de suite



Le classe Object

- C’est la racine de l’arbre d’héritage : java.lang.Object
- Toutes vos classe sont des descendantes de Object
- Les méthodes de la classe Object :
  - String toString()
    - A redéfinir pour personnaliser l’affichage de la valeur d’un objet
  - boolean equals(Object o)
    - Vérifie l’égalité entre 2 objets (la valeur et non la référence à l’objet)
  - public Class getClass()
    - Retourne un objet représentation la classe réelle d’un objet