Java : programmation objet

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Plan

Java

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Java

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)

Java

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe

Java

Qu'est ce qu'une classe en POO ?

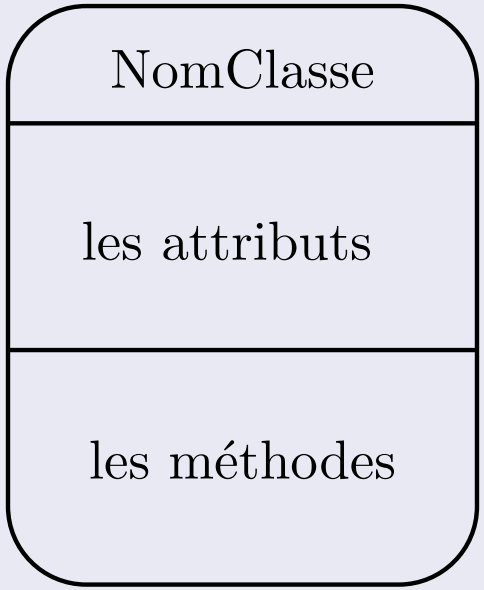
- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe
- instance $\equiv$ objet

Java

De quoi est composé une classe ?



- Attribut : [visibilité] + type + nom
- Méthode : [visibilité] + valeur de retour + nom + arguments ≡ signature : exactement comme les fonctions en procédurale

Java

Particularité du **Java**

- Toutes les classes héritent implicitement (pas besoin d'ajouter `extends`) d'une classe mère `Object`.
- La classe `Object` contient plusieurs méthodes telles que `toString()` (pour transformer un objet en chaîne de caractère), `clone()` (pour cloner)...
- Le mot-clé `this` permet de désigner l'objet courant.
- Contrairement à certains LOO, le `this` n'est pas obligatoire si aucune ambiguité ne se présente.

Java

Commençons par créer un nouveau projet **Java**

- Aller dans `File > New > Java Project`
- Remplir le champ `Project name:` avec `cours-poo` puis cliquer sur `Next`
- Décocher la case `Create module-info.java file` puis cliquer sur `Finish`

Java

Commençons par créer un nouveau projet **Java**

- Aller dans `File > New > Java Project`
- Remplir le champ `Project name:` avec `cours-poo` puis cliquer sur `Next`
- Décocher la case `Create module-info.java file` puis cliquer sur `Finish`

Créons deux packages `org.eclipse.test` et `org.eclipse.model`

- Aller dans `File > New > Package`
- Saisir le nom du premier package et valider
- Refaire la même chose pour le second

Java

Créons deux classes

- Une classe `Personne` dans `org.eclipse.model` contenant trois attributs : `num`, `nom` et `prénom`
- Une classe `Main` dans `org.eclipse.test` contenant le `public static void main` dans lequel on instanciera la classe `Personne`

Java

Créons deux classes

- Une classe `Personne` dans `org.eclipse.model` contenant trois attributs : `num`, `nom` et `prénom`
- Une classe `Main` dans `org.eclipse.test` contenant le `public static void main` dans lequel on instanciera la classe `Personne`

Créons les classes

- Aller dans `File > New > Class`
- Saisir le nom du package et celui de la classe
- Pour la classe `Main`, cocher la case `public static void main (String[] args)`
- Cliquer sur `Finish`

Contenu de la classe `Personne`

```
package org.eclipse.model;

public class Personne {
    int num;
    String nom;
    String prenom;
}
```

Contenu de la classe `Personne`

```
package org.eclipse.model;

public class Personne {
    int num;
    String nom;
    String prenom;
}
```

Remarques

- En Java, toute classe a un constructeur par défaut sans paramètres.
- Par défaut, la visibilité des attributs, en **Java**, est `package`.
- Donc, les attributs ne seront pas accessibles depuis la classe `Main` qui se situe dans un package différent (`org.eclipse.test`)
- Donc, changeons la visibilité des trois attributs de la classe `Personne`

Java

Nouveau contenu de la classe `Personne`

```
package org.eclipse.model;

public class Personne {
    public int num;
    public String nom;
    public String prenom;
}
```

Java

Contenu de la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
    // TODO Auto-generated method stub

    }
}
```

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

Étape 1 : déclaration d'un objet (objet non créé)

```
Personne personne;
```

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

Étape 1 : déclaration d'un objet (objet non créé)

```
Personne personne;
```

Étape 2 : instanciation de la classe `Personne` (objet créé)

```
personne = new Personne();
```

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

Étape 1 : déclaration d'un objet (objet non créé)

```
Personne personne;
```

Étape 2 : instanciation de la classe `Personne` (objet créé)

```
personne = new Personne();
```

On peut faire déclaration + instanciation

```
Personne personne = new Personne();
```

Java

Affectons les valeurs aux différents attributs

```
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";
```

Java

Affectons les valeurs aux différents attributs

```
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";
```


Pour être sûr que les valeurs ont bien été affectées aux attributs, on affiche

```
System.out.println(personne)
```

Java

Contenu de la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.num = 1;
        personne.nom = "wick";
        personne.prenom = "john";
        System.out.println(personne);
    }
}
```

Java

Contenu de la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.num = 1;
        personne.nom = "wick";
        personne.prenom = "john";
        System.out.println(personne);
    }
}
```

En exécutant, le résultat est

```
org.eclipse.model.Personne@7852e922
```

Java

Explication

- Pour afficher les détails d'un objet, il faut que la méthode `toString()` soit implémentée
- Pour la générer, clic droit sur la classe `Personne`, aller dans `source > Generate toString()...`, sélectionner les champs à afficher puis valider.

Java

Explication

- Pour afficher les détails d'un objet, il faut que la méthode `toString()` soit implémentée
- Pour la générer, clic droit sur la classe `Personne`, aller dans `source > Generate toString()...`, sélectionner les champs à afficher puis valider.

Le code de la méthode `toString()`

```
@Override
public String toString() {
    return "Personne [num=" + num + ", nom=" + nom + ", prenom=
       " + prenom + "]";
}
```

Java

@Override

- Une annotation (appelé aussi décorateur par **MicroSoft**)
- Pour nous rappeler qu'on redéfinit une méthode qui appartient à la classe mère (ici `Object`)

Java

@Override

- Une annotation (appelé aussi décorateur par **MicroSoft**)
- Pour nous rappeler qu'on redéfinit une méthode qui appartient à la classe mère (ici `Object`)

En exécutant, le résultat est :

```
Personne [num=1, nom=wick, prenom=john]
```

Java

Type de donnée	Valeur par défaut
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`boolean`	`false`
`char`	`'\u0000'` (caractère nul)
`String` (et autres objets)	`null`

Java

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

Java

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

Démarche

1. Bloquer l'accès directe aux attributs (mettre la visibilité à `private`)
2. Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (les `setter`)

Java

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

Démarche

1. Bloquer l'accès directe aux attributs (mettre la visibilité à `private`)
2. Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (les `setter`)

Convention

- Mettre la visibilité `private` ou `protected` pour tous les attributs
- Mettre la visibilité `public` pour toutes les méthodes

Mettons la visibilité `private` pour tous les attributs de la classe `Personne`

```
package org.eclipse.model;

public class Personne {
    private int num;
    private String nom;
    private String prenom;

    @Override
    public String toString() {
        return "Personne [num=" + num + ", nom=" + nom + ", prenom=" +
               prenom + "]";
    }
}
```

Mettons la visibilité `private` pour tous les attributs de la classe `Personne`

```
package org.eclipse.model;

public class Personne {
    private int num;
    private String nom;
    private String prenom;

    @Override
    public String toString() {
        return "Personne [num=" + num + ", nom=" + nom + ", prenom=" +
                prenom + "]";
    }
}
```


Dans la classe `Main`, les trois lignes suivantes sont soulignées en rouge

```
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";
```

Java

Explication

Les attributs sont privés, donc aucun accès direct ne sera autorisé

Java

Explication

Les attributs sont privés, donc aucun accès direct ne sera autorisé

Solution : générer les setters

- Faire clic droit sur la classe `Personne`
- Aller dans `Source > Generate Getters and Setters...`
- Cliquer sur chaque attribut et cocher la case `setNomAttribut(...)`
- Valider

Les trois méthodes générées

```
public void setNum(int num) {
    this.num = num;
}
public void setNom(String nom) {
    this.nom = nom;
}
public void setPrenom(String prenom) {
    this.prenom = prenom;
}
```

Les trois méthodes générées

```
public void setNum(int num) {
    this.num = num;
}
public void setNom(String nom) {
    this.nom = nom;
}
public void setPrenom(String prenom) {
    this.prenom = prenom;
}
```

Modifions `setNum()` pour ne plus accepter de valeurs inférieures à 1

```
public void setNum(int num) {
    if (num >= 1){
        this.num = num;
    }
}
public void setNom(String nom) {
    this.nom = nom;
}
public void setPrenom(String prenom) {
    this.prenom = prenom;
}
```

Java

Mettons à jour la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.setNum(1);
        personne.setNom("wick");
        personne.setPrenom("john");
        System.out.println(personne);
    }
}
```

Java

Mettons à jour la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.setNum(1);
        personne.setNom("wick");
        personne.setPrenom("john");
        System.out.println(personne);
    }
}
```

En exécutant, le résultat est

```
Personne [num=1, nom=wick, prenom=john]
```

Java

Mettons à jour la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.setNum(-1);
        personne.setNom("wick");
        personne.setPrenom("john");
        System.out.println(personne);
    }
}
```

Java

Mettons à jour la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.setNum(-1);
        personne.setNom("wick");
        personne.setPrenom("john");
        System.out.println(personne);
    }
}
```

En exécutant, le résultat est

```
Personne [num=0, nom=wick, prenom=john]
```

Java

Hypothèse

Si on voulait afficher les attributs (privés) de la classe `Personne`, un par un, dans la classe `Main` sans passer par le `toString()`

Java

Hypothèse

Si on voulait afficher les attributs (privés) de la classe `Personne`, un par un, dans la classe `Main` sans passer par le `toString()`

Démarche

Définir des méthodes qui retournent les valeurs des attributs (les `getter`)

Java

Pour générer les getters

- Faire clic droit sur la classe `Personne`
- Aller dans `Source > Generate Getters and Setters...`
- Cliquer sur chaque attribut et cocher la case `getNomAttribut()`
- Valider

Java

Mettons à jour la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.setNum(1);
        personne.setNom("wick");
        personne.setPrenom("john");
        System.out.println(personne.getNum() + " " + personne.getNom()
          + " " + personne.getPrenom());
    }
}
```

Java

Mettons à jour la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.setNum(1);
        personne.setNom("wick");
        personne.setPrenom("john");
        System.out.println(personne.getNum() + " " + personne.getNom()
          + " " + personne.getPrenom());
    }
}
```

En exécutant, le résultat est :

```
1 wick john
```

Java

Remarques

- Par défaut, toute classe en **Java** a un constructeur par défaut sans paramètre
- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe

Java

Remarques

- Par défaut, toute classe en **Java** a un constructeur par défaut sans paramètre
- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe

Pour générer un constructeur avec paramètre

1. Faire clic droit sur la classe `Personne`
2. Aller dans `Source > Generate Constructor using Fields...`
3. Vérifier que toutes les cases sont cochées et valider

Java

Le constructeur généré

```
public Personne(int num, String nom, String prenom) {
    super();
    this.num = num;
    this.nom = nom;
    this.prenom = prenom;
}
```

Java

Le constructeur généré

```
public Personne(int num, String nom, String prenom) {
    super();
    this.num = num;
    this.nom = nom;
    this.prenom = prenom;
}
```

Pour préserver la cohérence, il faut que le constructeur contrôle la valeur de l'attribut `num`

```
public Personne(int num, String nom, String prenom) {
    super();
    if (num >= 1){
        this.num = num;
    }
    this.nom = nom;
    this.prenom = prenom;
}
```

Java

On peut aussi appelé le `setter` dans le constructeur

```
public Personne(int num, String nom, String prenom)
   {
     super();
     this.setNum(num);
     this.nom = nom;
     this.prenom = prenom;
}
```

Java

Dans la classe `Main`, la ligne suivante est soulignée en rouge

```
Personne personne = new Personne();
```

Java

Dans la classe `Main`, la ligne suivante est soulignée en rouge

```
Personne personne = new Personne();
```

Explication

Le constructeur par défaut a été écrasé (il n'existe plus).

Java

Dans la classe `Main`, la ligne suivante est soulignée en rouge

```
Personne personne = new Personne();
```

Explication

Le constructeur par défaut a été écrasé (il n'existe plus).

Solution

Générer un constructeur sans paramètre.

Mettons à jour la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.setNum(1);
        personne.setNom("wick");
        personne.setPrenom("john");
        System.out.println(personne.getNum() + " " + personne.getNom()
          + " " + personne.getPrenom());
        Personne personne2 = new Personne(2, "bob", "mike");
        System.out.println(personne2);
    }
}
```

Mettons à jour la classe `Main`

```
package org.eclipse.test;

public class Main {

    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.setNum(1);
        personne.setNom("wick");
        personne.setPrenom("john");
        System.out.println(personne.getNum() + " " + personne.getNom()
          + " " + personne.getPrenom());
        Personne personne2 = new Personne(2, "bob", "mike");
        System.out.println(personne2);
    }
}
```

En exécutant, le résultat est :

```
1 wick john
Personne [num=2, nom=bob, prenom=mike]
```

Java

En testant l'égalité entre les deux objets suivants

```
Personne personne2 = new Personne(2, "bob", "mike");
Personne personne3 = new Personne(2, "bob", "mike");
System.out.println(personne2.equals(personne3));
```

Java

En testant l'égalité entre les deux objets suivants

```
Personne personne2 = new Personne(2, "bob", "mike");
Personne personne3 = new Personne(2, "bob", "mike");
System.out.println(personne2.equals(personne3));
```


Le résultat est

```
false
```

Java

Explication

- Par défaut, deux objets d'une même classe sont égaux s'ils pointent sur le même espace mémoire (identiques).
- Pour comparer les valeurs, il faut redéfinir les méthodes `equals()` et `hashCode()`.
  - `equals()` permet de définir comment tester sémantiquement l'égalité de deux objets.
  - `hashCode()` permet de retourner la valeur de hachage d'un objet (elle ne retourne pas un identifiant unique).
- Si une classe redéfinit `equals()`, alors elle doit aussi redéfinir `hashCode()`.
- `equals()` et `hashCode()` doivent utiliser les mêmes attributs.

Java

equals() et hashCode() doivent respecter les contraintes suivantes

- **Symétrie** : si `a.equals(b)` retourne `true` alors `b.equals(a)` aussi.
- **Réflexivité** : `a.equals(a)` retourne `true` si `a` est un objet non `null`.
- **Transitivité** : si `a.equals(b)` et `b.equals(c)` alors `a.equals(c)`.
- **Consistance** : si `a.equals(b)` alors `a.hashCode() == b.hashCode()`.
- `a.equals(null)` retourne `false`.

Java

Code par défaut de `equals()` dans `Object`

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

Java

Génération sous **Eclipse**

- Faire clic droit sur la classe `Personne`,
- Aller dans `source > Generate hashCode() and equals()`,
- Sélectionner les champs à utiliser puis valider.

Code généré

```
@Override
public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((nom == null) ? 0 : nom.hashCode());
        result = prime * result + num;
        result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
        return result;
}
@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        Personne other = (Personne) obj;
        if (nom == null) {
                if (other.nom != null)
                        return false;
        } else if (!nom.equals(other.nom))
                return false;
        if (num != other.num)
                return false;
        if (prenom == null) {
                if (other.prenom != null)
                        return false;
        } else if (!prenom.equals(other.prenom))
                return false;
        return true;
}
```

Java

En retestant le code précédent

```
Personne personne2 = new Personne(2, "bob", "mike");
Personne personne3 = new Personne(2, "bob", "mike");
System.out.println(personne2.equals(personne3));
```

Java

En retestant le code précédent

```
Personne personne2 = new Personne(2, "bob", "mike");
Personne personne3 = new Personne(2, "bob", "mike");
System.out.println(personne2.equals(personne3));
```


Le résultat est

```
true
```

Java

Exercice 1

1. Définir une classe `Adresse` contenant trois attributs privés : `rue`, `codePostal` et `ville`, tous de type chaîne de caractères.
2. Ajouter un constructeur avec trois paramètres, les getters, les setters, ainsi que la méthode `toString()`.
3. Dans la classe `Personne`, ajouter un attribut `adresse` de type `Adresse`, définir un nouveau constructeur à quatre paramètres, ainsi que le getter et le setter pour cet attribut.
4. Dans la classe `Main`, créer un objet `adresse` de type `Adresse`.
5. Créer également un objet `personne` de type `Personne` en lui associant l'objet `adresse` créé précédemment.
6. Afficher tous les attributs de l'objet `personne`.

Java

Exercice 2

1. Dans la classe `Personne`, remplacer l'attribut `adresse` de type `Adresse`, par `adresses` de type `Adresse[]` et régénérer le constructeur à quatre paramètres, ainsi que le getter et le setter.
2. Dans la classe `Main`, créer quelques objets de type `Adresse`.
3. Créer également un objet `personne` de type `Personne` en lui associant les objets de type `Adresse` créés précédemment.
4. Afficher tous les attributs de l'objet `personne`.

Java

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs.

Java

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs.

Hypothèse

Et si nous voudrions qu'un attribut ait une valeur partagée par toutes les instances (le nombre d'objets instanciés de la classe `Personne`).

Java

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs.

Hypothèse

Et si nous voudrions qu'un attribut ait une valeur partagée par toutes les instances (le nombre d'objets instanciés de la classe `Personne`).

Définition

Un attribut dont la valeur est partagée par toutes les instances de la classe est appelée : attribut statique ou attribut de classe.

Java

Exemple

- Pour créer un attribut contenant le nombre des objets créés à partir de la classe `Personne`.
- Cet attribut doit être `static`, sinon chaque objet pourrait avoir sa propre valeur pour cet attribut.

Java

Ajoutons un attribut statique `nbrPersonnes` à la liste d'attributs de la classe `Personne`

```
private  static int nbrPersonnes;
```

Java

Ajoutons un attribut statique `nbrPersonnes` à la liste d'attributs de la classe `Personne`

```
private  static int nbrPersonnes;
```

Incrémentons notre compteur de personnes dans les constructeurs

```
public Personne() {
    super();
    nbrPersonnes++;
}
public Personne(int num, String nom, String prenom) {
    super();
    this.setNum(num);
    this.nom = nom;
    this.prenom = prenom;
    nbrPersonnes++;
}
```

Générons un `getter` pour l'attribut static `nbrPersonnes`

```
public static int getNbrPersonnes() {
    return nbrPersonnes;
}
```

Générons un `getter` pour l'attribut static `nbrPersonnes`

```
public static int getNbrPersonnes() {
    return nbrPersonnes;
}
```

Testons cela dans la classe `Main`

```
Personne personne = new Personne();
personne.setNum(1);
personne.setNom("wick");
personne.setPrenom("john");
System.out.println(personne.getNum() + " " +personne.getNom() + " " + personne.
   getPrenom());
System.out.println(Personne.getNbrPersonnes());
Personne personne2 = new Personne(2, "bob", "mike");
System.out.println(personne2);
System.out.println(Personne.getNbrPersonnes());
```

Générons un `getter` pour l'attribut static `nbrPersonnes`

```
public static int getNbrPersonnes() {
    return nbrPersonnes;
}
```

Testons cela dans la classe `Main`

```
Personne personne = new Personne();
personne.setNum(1);
personne.setNom("wick");
personne.setPrenom("john");
System.out.println(personne.getNum() + " " +personne.getNom() + " " + personne.
   getPrenom());
System.out.println(Personne.getNbrPersonnes());
Personne personne2 = new Personne(2, "bob", "mike");
System.out.println(personne2);
System.out.println(Personne.getNbrPersonnes());
```

Le résultat est

```
1 wick john
1
Personne [num=2, nom=bob, prenom=mike]
2
```

Java

Block statique ou initialiseur statique (**Static Initializer**)

- Exécuté une seule fois lorsque le fichier `.class` est chargé par **ClassLoader**
- Exécuté même avant `main`
- N'est pas exécuté à chaque instanciation de la classe
- N'a pas accès à `this`

Java

Block statique ou initialiseur statique (**Static Initializer**)

- Exécuté une seule fois lorsque le fichier `.class` est chargé par **ClassLoader**
- Exécuté même avant `main`
- N'est pas exécuté à chaque instanciation de la classe
- N'a pas accès à `this`

Initialiseur d'instance (**Instance Initializer**)

- Exécuté chaque fois qu'un constructeur de la classe est appelé
- A accès à `this`
- A accès aux attributs statiques et non-statiques

Exemple

```
package org.eclipse.model;

public class Personne {

    // attributs

    static {
        System.out.println("Static initializer");
    }
    {
        System.out.println("Instance initializer");
    }

    // méthodes

}
```

Exemple

```
package org.eclipse.model;

public class Personne {

    // attributs

    static {
        System.out.println("Static initializer");
    }
    {
        System.out.println("Instance initializer");
    }

    // méthodes

}
```

Lancez le projet et vérifiez que le block static s'exécute une seule fois alors que l'initialiseur d'instance s'exécute chaque fois qu'on instance la classe `Personne`.

Java

Quelle est la différence entre le constructeur et l'initialiseur d'instance

- L'intialiseur est souvent utilisé pour contenir le code qui se répète dans tous les constructeurs de la classe.
- L'initialiseur est toujours exécuté avant le constructeur.
- L'initialiseur est aussi utilisé par les classes internes anonymes qui ne peuvent avoir de constructeur.

Java

Classe statique

- Seules les classes imbriquées peuvent être static
- Pas de lien implicite avec l'instance de la classe externe
- Pouvant être instanciée sans objet de la classe externe
- Pouvant contenir des attributs/méthodes static et non-static

Java

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est **une sorte de** `Classe2`

Java

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire

Java

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau

Java

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne

Java

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que numéro, nom et prénom

Java

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que numéro, nom et prénom
- Donc, on peut mettre en commun les attributs numéro, nom et prénom dans une classe `Personne`

Java

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que numéro, nom et prénom
- Donc, on peut mettre en commun les attributs numéro, nom et prénom dans une classe `Personne`
- Les classes `Étudiant` et `Enseignant` hériteront (`extends`) de la classe `Personne`

Java

Particularité du **Java**

- L'héritage multiple n'est pas permis par le langage Java
- C'est-à-dire, une classe ne peut hériter de plusieurs classes différentes

Java

Créons les classes `Etudiant` et `Enseignant`

- Aller dans `File > New > Class`
- Choisir le package `org.eclipse.model`
- Saisir le nom de la classe
- Dans la section `Superclass`, cliquer sur `Browse` et chercher puis sélectionner `Personne`
- Cliquer sur `Ok` et enfin `Finish`
- Refaire la même chose pour la seconde classe

Java

Contenu de la classe `Enseignant`

```
package org.eclipse.model;

public class Enseignant extends Personne {

}
```

Java

Contenu de la classe `Enseignant`

```
package org.eclipse.model;

public class Enseignant extends Personne {

}
```

Contenu de la classe `Etudiant`

```
package org.eclipse.model;

public class Etudiant extends Personne {

}
```

Java

Contenu de la classe `Enseignant`

```
package org.eclipse.model;

public class Enseignant extends Personne {

}
```

Contenu de la classe `Etudiant`

```
package org.eclipse.model;

public class Etudiant extends Personne {

}
```

`extends` est le mot-clé à utiliser pour définir une relation d'héritage entre deux classes

Java

Ensuite

- Créer un attribut `niveau` dans la classe `Etudiant` puis générer les getter et setter
- Créer un attribut `salaire` dans la classe `Enseignant` puis générer les getter et setter

Java

Pour créer un objet de type `Enseignant`

```
Enseignant enseignant = new Enseignant();
enseignant.setNum(3);
enseignant.setNom("green");
enseignant.setPrenom("jonas");
enseignant.setSalaire(1700);
System.out.println(enseignant);
```

Java

Pour créer un objet de type `Enseignant`

```
Enseignant enseignant = new Enseignant();
enseignant.setNum(3);
enseignant.setNom("green");
enseignant.setPrenom("jonas");
enseignant.setSalaire(1700);
System.out.println(enseignant);
```

En exécutant, le résultat est :

```
Personne [num=3, nom=green, prenom=jonas]
```

Java

Pour créer un objet de type `Enseignant`

```
Enseignant enseignant = new Enseignant();
enseignant.setNum(3);
enseignant.setNom("green");
enseignant.setPrenom("jonas");
enseignant.setSalaire(1700);
System.out.println(enseignant);
```

En exécutant, le résultat est :

```
Personne [num=3, nom=green, prenom=jonas]
```

Mais on ne voit pas le salaire, pourquoi ?

car on a pas redéfini la méthode `toString()`, on a utilisé celle de la classe mère

Java

Et si on génère le `toString()` dans la classe `Enseignant`

```
@Override
public String toString() {
    return "Enseignant [salaire=" + salaire + "]";
}
```

Java

Et si on génère le `toString()` dans la classe `Enseignant`

```
@Override
public String toString() {
    return "Enseignant [salaire=" + salaire + "]";
}
```

Pour appeler le `toString()` de la classe mère à partir de classe fille (`Enseignant`)

```
@Override
public String toString() {
    return super.toString() +  " Enseignant [salaire=" + salaire + "]";
}
```

Java

Et si on génère le `toString()` dans la classe `Enseignant`

```
@Override
public String toString() {
    return "Enseignant [salaire=" + salaire + "]";
}
```

Pour appeler le `toString()` de la classe mère à partir de classe fille (`Enseignant`)

```
@Override
public String toString() {
    return super.toString() +  " Enseignant [salaire=" + salaire + "]";
}
```

Le mot-clé `super` permet d'appeler une méthode de la classe mère

Comment générer un constructeur à plusieurs paramètres et utiliser celui de la classe mère

- Faire clic droit sur la classe `Enseignant`
- Aller dans `Source > Generate Constructor using Fields...`
- Dans `Select super constructor to invoke`, sélectionner le constructeur à trois paramètres
- Dans `Select fields to initialize`, vérifier que la case `salaire` est sélectionnée et valider

Comment générer un constructeur à plusieurs paramètres et utiliser celui de la classe mère

- Faire clic droit sur la classe `Enseignant`
- Aller dans `Source > Generate Constructor using Fields...`
- Dans `Select super constructor to invoke`, sélectionner le constructeur à trois paramètres
- Dans `Select fields to initialize`, vérifier que la case `salaire` est sélectionnée et valider

Le constructeur généré

```
public Enseignant(int num, String nom, String prenom, int salaire) {
    super(num, nom, prenom);
    this.salaire = salaire;
}
```

Comment générer un constructeur à plusieurs paramètres et utiliser celui de la classe mère

- Faire clic droit sur la classe `Enseignant`
- Aller dans `Source > Generate Constructor using Fields...`
- Dans `Select super constructor to invoke`, sélectionner le constructeur à trois paramètres
- Dans `Select fields to initialize`, vérifier que la case `salaire` est sélectionnée et valider

Le constructeur généré

```
public Enseignant(int num, String nom, String prenom, int salaire) {
    super(num, nom, prenom);
    this.salaire = salaire;
}
```

Maintenant, on peut créer un enseignant ainsi

```
Enseignant enseignant = new Enseignant(3, "green", "jonas", 1700);
```

Java

À partir de la classe `Enseignant`

- On ne peut avoir accès direct à un attribut de la classe mère
- C'est-à-dire, on ne peut faire `this.num` car les attributs ont une visibilité `private`
- Pour modifier la valeur d'un attribut privé de la classe mère, il faut
  - soit utiliser les getters/setters
  - soit mettre la visibilité des attributs de la classe mère à `protected`

Java

	Class	**Package**	**Subclass** same package	**Subclass** diff package	**Everywhere**
`public`	√	√	√	√	√
`protected`	√	√	√	√	
no-visibility	√	√	√		
`private`	√				

Java

On peut créer un objet de la classe `Personne` ainsi

```
Enseignant enseignant = new Enseignant(3, "green", "jonas", 1700);
```

Java

On peut créer un objet de la classe `Personne` ainsi

```
Enseignant enseignant = new Enseignant(3, "green", "jonas", 1700);
```

Ou ainsi

```
Personne enseignant = new Enseignant(3, "green", "jonas", 1700);
```

Java

On peut créer un objet de la classe `Personne` ainsi

```
Enseignant enseignant = new Enseignant(3, "green", "jonas", 1700);
```

Ou ainsi

```
Personne enseignant = new Enseignant(3, "green", "jonas", 1700);
```

Ceci est faux

```
Enseignant enseignant = new Personne(3, "green", "jonas");
```

Java

Remarque

Pour connaître la classe d'un objet, on peut utiliser le mot-clé `instanceof`

Java

Remarque

Pour connaître la classe d'un objet, on peut utiliser le mot-clé `instanceof`

Exemple

```
System.out.println(enseignant instanceof Enseignant);
// affiche true

System.out.println(enseignant instanceof Personne);
// affiche true

System.out.println(personne instanceof Enseignant);
// affiche false
```

Java

Exercice

1. Créer un objet de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne` et stocker les tous dans un même tableau.
2. Parcourir le tableau et afficher pour chacun soit le `numero` s'il est personne, soit le `salaire` s'il est enseignant ou soit le `niveau` s'il est étudiant.

Java

Exercice

1. Créer un objet de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne` et stocker les tous dans un même tableau.
2. Parcourir le tableau et afficher pour chacun soit le `numero` s'il est personne, soit le `salaire` s'il est enseignant ou soit le `niveau` s'il est étudiant.

Pour parcourir un tableau, on peut utiliser le raccourci de `for`

```
Personne personnes [] = { personne, etudiant, enseignant };

for(Personne perso : personnes) {

}
```

Java

Solution

```
Personne personnes[] = { personne, etudiant, enseignant };

for (Personne perso : personnes) {
    if (perso instanceof Etudiant) {
        System.out.println(((Etudiant) perso).getNiveau());
    } else if (perso instanceof Enseignant) {
        System.out.println(((Enseignant) perso).getSalaire());
    } else {
        System.out.println(perso.getNum());
    }
}
```

Java

Solution sans cast [disponible depuis Java 14]

```
Personne personnes[] = { personne, etudiant, enseignant };

for (Personne perso : personnes) {
    if (perso instanceof Etudiant e) {
        System.out.println(e.getNiveau());
    } else if (perso instanceof Enseignant e) {
        System.out.println(e.getSalaire());
    } else {
        System.out.println(perso.getNum());
    }
}
```

Java

L'agrégation

- C'est une association non-symétrique
- Elle représente une relation de type ensemble/élément

Java

L'agrégation

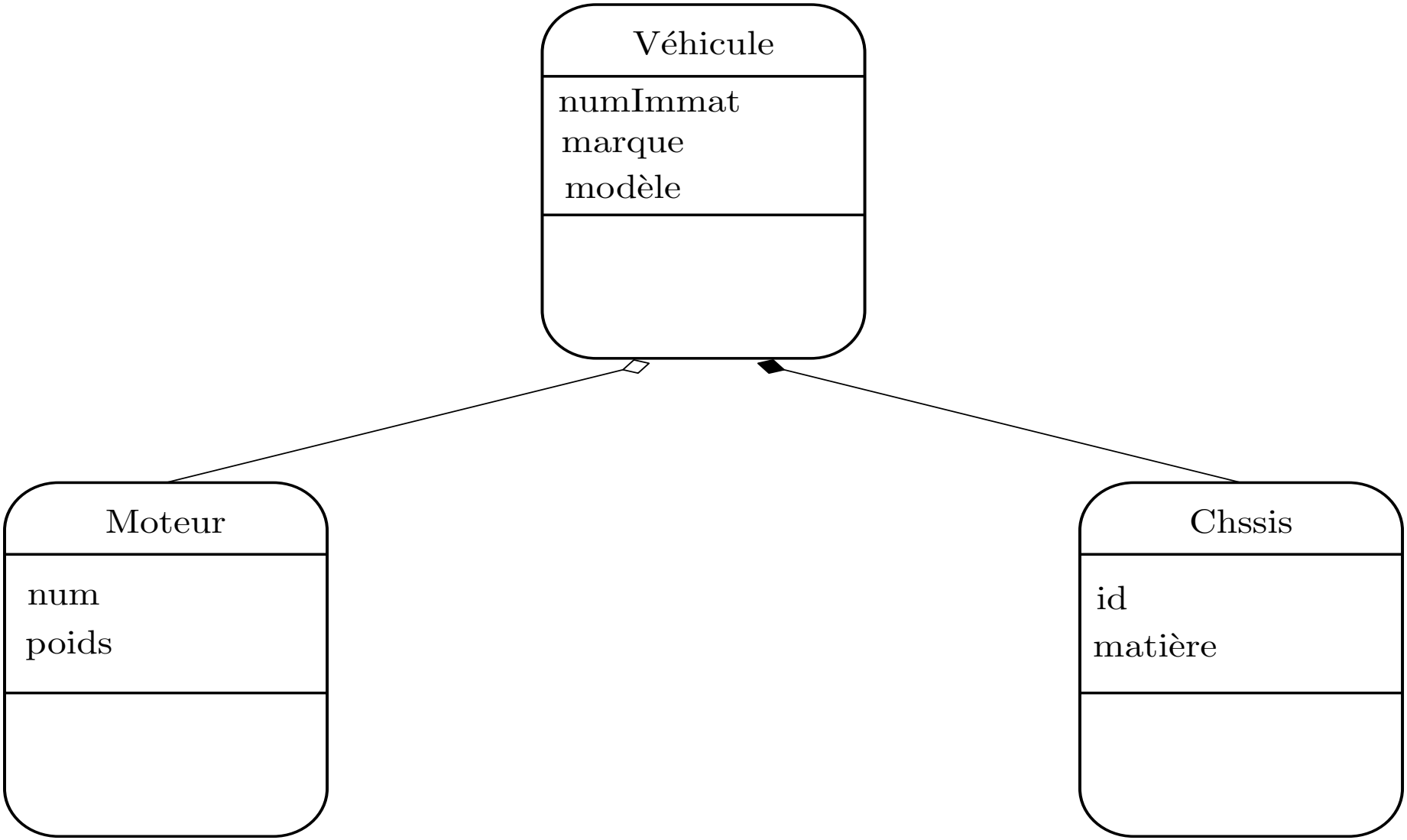
- C'est une association non-symétrique
- Elle représente une relation de type ensemble/élément

La composition

- C'est une agrégation forte
- L'élément n'existe pas sans l'agrégat (l'élément est détruit lorsque l'agrégat n'existe plus)

Java

Comment coder des relations d'agrégation et de composition en Java ?



Java

La classe `Châssis`

```
public class Chassis {
        private int id;
        private String matiere;
        public int getId() {
                return id;
        }
        public void setId(int id) {
                this.id = id;
        }
        public String getMatiere() {
                return matiere;
        }
        public void setMatiere(String matiere) {
                this.matiere = matiere;
        }
}
```

Java

La classe `Moteur`

```
public class Moteur {
        private int num;
        private float poids;
        public int getNum() {
                return num;
        }
        public void setNum(int num) {
                this.num = num;
        }
        public float getPoids() {
                return poids;
        }
        public void setPoids(float poids) {
                this.poids = poids;
        }
}
```

Java

La classe `Véhicule`

```
public class Vehivule {
        private Moteur moteur;
        private final Chassis chassis;
        public Vehivule(Chassis chassis) {
                this.chassis = chassis;
        }
        public Moteur getMoteur() {
                return moteur;
        }
        public void setMoteur(Moteur moteur) {
                this.moteur = moteur;
        }
}
```

Java

La classe `Véhicule`

```
public class Vehivule {
        private Moteur moteur;
        private final Chassis chassis;
        public Vehivule(Chassis chassis) {
                this.chassis = chassis;
        }
        public Moteur getMoteur() {
                return moteur;
        }
        public void setMoteur(Moteur moteur) {
                this.moteur = moteur;
        }
}
```

Explication

- un composant est déclaré `final` et instanciable une seule fois au moment de l'instanciation de l'objet composite
- un composant n'a ni getter ni setter
- lorsque le véhiculé est détruit, le châssis le sera aussi

Java

Pour tester

```
public class Main {
  public static void main(String[] args) {
    Chassis chassis = new Chassis();
    chassis.setId(100);
    chassis.setMatiere("fer");
    Vehivule vehivule = new Vehivule(chassis);
    Moteur moteur = new Moteur();
    moteur.setNum(100);
    moteur.setPoids(500f);
    vehivule.setMoteur(moteur);
    // pas de méthode permettant de modifier le châ
       ssis d'un véhicule
  }
}
```

Java

Classe abstraite

- C'est une classe qu'on ne peut instancier
- Les constructeurs peuvent donc être supprimés

Java

Classe abstraite

- C'est une classe qu'on ne peut instancier
- Les constructeurs peuvent donc être supprimés

Si on déclare la classe `Personne` abstraite

```
public abstract class Personne {
    ...
}
```

Java

Classe abstraite

- C'est une classe qu'on ne peut instancier
- Les constructeurs peuvent donc être supprimés

Si on déclare la classe `Personne` abstraite

```
public abstract class Personne {
    ...
}
```

Tout ce code sera souligné en rouge

```
Personne personne = new Personne();
...
Personne personne2 = new Personne(2, "bob", "mike");
```

Java

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

Java

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

Déclarons une méthode abstraite `afficherDetails()` dans `Personne`

```
public abstract void afficherDetails();
```

Java

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

Déclarons une méthode abstraite `afficherDetails()` dans `Personne`

```
public abstract void afficherDetails();
```

`Etudiant` et `Enseignant` sont soulignées en rouge.

Java

Pour implémenter la méthode abstraite

- Placer le curseur sur le nom de la classe
- Dans le menu afficher, choisir `Add unimplemented methods`

Java

Pour implémenter la méthode abstraite

- Placer le curseur sur le nom de la classe
- Dans le menu afficher, choisir `Add unimplemented methods`

Code généré

```
@Override
public void afficherDetails() {
    // TODO Auto-generated method stub
}
```

Java

Dans `Enseignant`

```
@Override
public void afficherDetails() {
    System.out.println(getPrenom() + " " + getNom() + " " + salaire);
}
```

Java

Dans `Enseignant`

```
@Override
public void afficherDetails() {
    System.out.println(getPrenom() + " " + getNom() + " " + salaire);
}
```

Dans `Etudiant`

```
@Override
public void afficherDetails() {
    System.out.println(getPrenom() + " " + getNom() + " " + niveau);
}
```

Java

Dans `Enseignant`

```
@Override
public void afficherDetails() {
    System.out.println(getPrenom() + " " + getNom() + " " + salaire);
}
```

Dans `Etudiant`

```
@Override
public void afficherDetails() {
    System.out.println(getPrenom() + " " + getNom() + " " + niveau);
}
```

Pour tester

```
Enseignant enseignant =  new Enseignant(3, "green", "jonas", 1700);
enseignant.afficherDetails();
// affiche jonas green 1700
```

Java

Classe finale

C'est une classe qui ne peut avoir de classes filles.

Java

Classe finale

C'est une classe qui ne peut avoir de classes filles.

Pour tester

Commençons par mettre en commentaire `afficherNomComplet()` dans `Personne`, `Enseignant` et `Etudiant`.

Java

Classe finale

C'est une classe qui ne peut avoir de classes filles.

Pour tester

Commençons par mettre en commentaire `afficherNomComplet()` dans `Personne`, `Enseignant` et `Etudiant`.

Déclarons la classe `Personne` finale

```
public final class Personne {
    ...
}
```

Java

Classe finale

C'est une classe qui ne peut avoir de classes filles.

Pour tester

Commençons par mettre en commentaire `afficherNomComplet()` dans `Personne`, `Enseignant` et `Etudiant`.

Déclarons la classe `Personne` finale

```
public final class Personne {
    ...
}
```

Les deux classes filles sont en rouge

```
The type Enseignant cannot subclass the final class Personne
```

Java

Méthode finale

C'est une méthode qu'on ne peut redéfinir.

Java

Méthode finale

C'est une méthode qu'on ne peut redéfinir.

Pour tester

Commençons par supprimer le mot-clé `final` dans `Personne`.

Java

Méthode finale

C'est une méthode qu'on ne peut redéfinir.

Pour tester

Commençons par supprimer le mot-clé `final` dans `Personne`.

Ajoutons une méthode finale `afficherNomComplet()` dans `Personne`

```
public final void afficherNomComplet() {
    System.out.println(this.getPrenom() + " " + this.getNom());
}
```

Java

Méthode finale

C'est une méthode qu'on ne peut redéfinir.

Pour tester

Commençons par supprimer le mot-clé `final` dans `Personne`.

Ajoutons une méthode finale `afficherNomComplet()` dans `Personne`

```
public final void afficherNomComplet() {
    System.out.println(this.getPrenom() + " " + this.getNom());
}
```

Si on essaye de redéfinir cette méthode dans `Enseignant`

```
Cannot override the final method from Personne
```

Java

Remarques

- Une classe abstraite ne doit pas forcément contenir une méthode abstraite
- Une classe finale ne doit pas forcément contenir une méthode finale
- Une méthode finale ne doit pas forcément être dans une classe finale

Java

En **Java**, une classe

- ne peut hériter (**étendre**) que d'une seule classe
- peut hériter (**implémenter**) de plusieurs interfaces

Java

Une interface

- déclarée avec le mot-clé `interface`
- comme une classe abstraite (impossible de l'instancier) dont :
  - toutes les méthodes sont abstraites
  - tous les attributs sont constants
- un protocole, un contrat : toute classe qui hérite d'une interface doit implémenter toutes ses méthodes
- pouvant proposée une implémentation par défaut pour les méthodes en utilisant le mot-clé `default`

Java

Pour créer une interface sous **Eclipse**

- Aller dans `File > New > Interface`
- Saisir `org.eclipse.interfaces` dans `Package name`
- Saisir `IMiseEnForme` dans `Name`
- Valider

Java

Contenu généré de l'interface `IMiseEnForme`

```
package org.eclipse.interfaces;

public interface IMiseEnForme {

}
```

Java

Contenu généré de l'interface `IMiseEnForme`

```
package org.eclipse.interfaces;

public interface IMiseEnForme {

}
```

Définissons la signature de deux méthodes dans l'interface `IMiseEnForme`

```
package org.eclipse.interfaces;

public interface IMiseEnForme {
    public void afficherNomMajuscule();
    public void afficherPrenomMajuscule();
}
```

Java

Pour hériter d'une interface, on utilise le mot-clé `implements`

```
public class Personne implements IMiseEnForme {
    ...
}
```

Java

Pour hériter d'une interface, on utilise le mot-clé `implements`

```
public class Personne implements IMiseEnForme {
    ...
}
```

La classe `Personne` est soulignée en rouge

- Placer le curseur sur la classe `Personne`
- Dans le menu affiché, sélectionner `Add unimplemented methods`

Java

Code généré

```
@Override
public void afficherNomMajuscule() {
    // TODO Auto-generated method stub

}

@Override
public void afficherPrenomMajuscule() {
    // TODO Auto-generated method stub

}
```

Java

Modifions le code de deux méthodes générées

```
@Override
public void afficherNomMajuscule() {
        System.out.println(nom.toUpperCase());
}

@Override
public void afficherPrenomMajuscule() {
        System.out.println(prenom.toUpperCase());

}
```

Java

Pour tester

```
Enseignant enseignant =  new Enseignant(3, "green", "jonas", 1700);
enseignant.afficherNomMajuscule();
enseignant.afficherPrenomMajuscule();
```

Java

Pour tester

```
Enseignant enseignant =  new Enseignant(3, "green", "jonas", 1700);
enseignant.afficherNomMajuscule();
enseignant.afficherPrenomMajuscule();
```

En exécutant, le résultat est

```
GREEN
JONAS
```

Java

Définissons un attribut `i` dans l'interface `IMiseEnForme`

```
package org.eclipse.interfaces;

public interface IMiseEnForme {

    int i = 5;

    public void afficherNomMajuscule();
    public void afficherPrenomMajuscule();

}
```

Java

Définissons un attribut `i` dans l'interface `IMiseEnForme`

```
package org.eclipse.interfaces;

public interface IMiseEnForme {

    int i = 5;

    public void afficherNomMajuscule();
    public void afficherPrenomMajuscule();

}
```

L'attribut `i` est, par définition `static` et `final`

Java

Il est possible de définir une implémentation par défaut pour une méthode d'interface (depuis Java 8)

```
package org.eclipse.interfaces;

public interface IMiseEnForme {
        int i = 5;

        default public void afficherNomMajuscule() {
                System.out.println("DOE");
        }
        public void afficherPrenomMajuscule();
}
```

Java

Il est possible de définir une implémentation par défaut pour une méthode d'interface (depuis Java 8)

```
package org.eclipse.interfaces;

public interface IMiseEnForme {
        int i = 5;

        default public void afficherNomMajuscule() {
                System.out.println("DOE");
        }
        public void afficherPrenomMajuscule();
}
```

Remarque

Les classes filles ne sont pas dans l'obligation d'implémenter les méthodes d'une interface ayant une implémentation par défaut.

Java

Remarques

- Une interface peut hériter de plusieurs autres interfaces (mais pas d'une classe)
- Pour cela, il faut utiliser le mot-clé `extends` et pas `implements` car une interface n'implémente jamais de méthodes.

Java

Remarques

- Une interface peut hériter de plusieurs autres interfaces (mais pas d'une classe)
- Pour cela, il faut utiliser le mot-clé `extends` et pas `implements` car une interface n'implémente jamais de méthodes.

Question : une interface est-elle vraiment une classe abstraite ?

Non, car toute classe abstraite hérite de la classe `Object` mais une interface **non**.

Java

Définissons une deuxième interface `IAffichage`

```
package org.eclipse.interfaces;

public interface IAffichage {
        public void afficherNomPrenomEnMajuscule();
}
```

Java

Définissons une deuxième interface `IAffichage`

```
package org.eclipse.interfaces;

public interface IAffichage {
        public void afficherNomPrenomEnMajuscule();
}
```

Une interface peut étendre une ou plusieurs autres interfaces

```
package org.eclipse.interfaces;

public interface IMiseEnForme extends IAffichage{
        int i = 5;

        default public void afficherNomMajuscule() {
                System.out.println("DOE");
        }
        public void afficherPrenomMajuscule();
}
```

Java

Remarque

Les classes qui implémentent `IMiseEnForme` doivent

- soit implémenter la méthode `afficherNomPrenomEnMajuscule()` de `IAffichage`,
- soit se déclarer abstraite et ne pas implémenter la méthode `afficherNomPrenomEnMajuscule()`. Dans ce cas, ses sous-classes qui doivent l'implémenter.

Java

Considérons le tableau d'étudiant suivant

```
Etudiant etudiant1 = new Etudiant(203, "El Mouelhi", "Achref", "licence");
Etudiant etudiant2 = new Etudiant(201, "El Mouelhi", "Achref", "master");
Etudiant etudiant3 = new Etudiant(202, "El Mouelhi", "Achref", "doctorat");
Etudiant[] etudiants = new Etudiant[3];
etudiants[0] = etudiant1;
etudiants[1] = etudiant2;
etudiants[2] = etudiant3;
```

Java

Considérons le tableau d'étudiant suivant

```
Etudiant etudiant1 = new Etudiant(203, "El Mouelhi", "Achref", "licence");
Etudiant etudiant2 = new Etudiant(201, "El Mouelhi", "Achref", "master");
Etudiant etudiant3 = new Etudiant(202, "El Mouelhi", "Achref", "doctorat");
Etudiant[] etudiants = new Etudiant[3];
etudiants[0] = etudiant1;
etudiants[1] = etudiant2;
etudiants[2] = etudiant3;
```

Trier le tableau $\Rightarrow$ erreur

```
Arrays.sort(etudiants);
```

Java

Considérons le tableau d'étudiant suivant

```
Etudiant etudiant1 = new Etudiant(203, "El Mouelhi", "Achref", "licence");
Etudiant etudiant2 = new Etudiant(201, "El Mouelhi", "Achref", "master");
Etudiant etudiant3 = new Etudiant(202, "El Mouelhi", "Achref", "doctorat");
Etudiant[] etudiants = new Etudiant[3];
etudiants[0] = etudiant1;
etudiants[1] = etudiant2;
etudiants[2] = etudiant3;
```

Trier le tableau $\Rightarrow$ erreur

```
Arrays.sort(etudiants);
```

Quelle solution pour trier le tableau ?

implémenter l'interface `Comparable`.

Java

Commençons par faire implémenter l'interface `Comparable`

```
public class Etudiant extends Personne implements MiseEnForme, Comparable<Etudiant> {

    // contenu précédent

}
```

Java

Commençons par faire implémenter l'interface `Comparable`

```
public class Etudiant extends Personne implements MiseEnForme, Comparable<Etudiant> {

    // contenu précédent

}
```

Implémentons la méthode `compare` de cette interface

```
@Override
public int compareTo(Etudiant o) {
    return Integer.compare(num, o.num);
}
```

Java

Commençons par faire implémenter l'interface `Comparable`

```
public class Etudiant extends Personne implements MiseEnForme, Comparable<Etudiant> {

    // contenu précédent

}
```

Implémentons la méthode `compare` de cette interface

```
@Override
public int compareTo(Etudiant o) {
    return Integer.compare(num, o.num);
}
```

Testons et vérifions que le tableau a été trié selon l'identifiant

```
System.out.println("Avant : " + Arrays.toString(etudiants));
Arrays.sort(etudiants);
System.out.println("Après : " + Arrays.toString(etudiants));
```

Nous pourrons aussi définir une classe qui implémente l'interface `Comparator`

```
package org.eclipse.comparator;

import java.util.Comparator;

import org.eclipse.model.Etudiant;

public class EtudiantNumComparator implements Comparator<Etudiant> {

    @Override
    public int compare(Etudiant o1, Etudiant o2) {
        return Integer.compare(o1.getNum(), o2.getNum());
    }

}
```

Pour tester

```
System.out.println("Avant : " + Arrays.toString(etudiants));
Arrays.sort(etudiants, new EtudiantNumComparator());
System.out.println("Après : " + Arrays.toString(etudiants));
```

Java

	Comparable	**Comparator**
Autorise les multiples comparateurs		√
Doit être implémentée par la classe concernée	√	
Définie dans `java.lang`	√	
Définie dans `java.util`		√

Java

En copiant un objet, modifier un $\Rightarrow$ modifier l'autre

```
var etudiant1 = new Etudiant(203, "El Mouelhi", "Achref", "licence");
var etudiant2 = (Etudiant) etudiant1.clone();
etudiant2.setNom("Doe");
System.out.println(etudiant1.getNom());
// affiche Doe
```

Java

En copiant un objet, modifier un $\Rightarrow$ modifier l'autre

```
var etudiant1 = new Etudiant(203, "El Mouelhi", "Achref", "licence");
var etudiant2 = (Etudiant) etudiant1.clone();
etudiant2.setNom("Doe");
System.out.println(etudiant1.getNom());
// affiche Doe
```

Quelle solution pour avoir deux copies indépendantes ?

implémenter l'interface `Cloneable`.

Java

Implémentons l'interface `Cloneable` et sa méthode `clone()`

```
public class Etudiant extends Personne implements Cloneable {

    //contenu précédent

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}
```

Java

Implémentons l'interface `Cloneable` et sa méthode `clone()`

```
public class Etudiant extends Personne implements Cloneable {

    //contenu précédent

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}
```

Appelant la méthode `clone` pour créer une copie indépendante

```
var etudiant1 = new Etudiant(203, "El Mouelhi", "Achref", "licence");
var etudiant2 = (Etudiant) etudiant1.clone();
etudiant2.setNom("Doe");
System.out.println(etudiant1.getNom());
// affiche El Mouelhi
```

Java

Classe scellée [**Java 15**]

- Une classe qui précise les classes qui peuvent l'étendre.
- Elle utilise des nouveaux mots-clés : `sealed`, `non-sealed` et `permits`.

Java

Classe scellée [**Java 15**]

- Une classe qui précise les classes qui peuvent l'étendre.
- Elle utilise des nouveaux mots-clés : `sealed`, `non-sealed` et `permits`.

Remarques

- Les classes autorisées par une classe scellée doivent l'implémenter.
- La classe scellée et les classes autorisées doivent être déclarées dans le même package.

Java

Exemple

```
public sealed  class Personne implements IMiseEnForme permits
   Enseignant, Etudiant {

     // contenu précédent

}
```

Java

Exemple

```
public sealed  class Personne implements IMiseEnForme permits
   Enseignant, Etudiant {

     // contenu précédent

}
```

Explication

Seules les classes `Etudiant` et `Enseignant` peuvent étendre `Personne`.

Java

Le code suivant engendrera l'erreur ci-dessous

```
public class Vacataire extends Personne {

}
```

Java

Le code suivant engendrera l'erreur ci-dessous

```
public class Vacataire extends Personne {

}
```

L'erreur

```
The type Vacataire extending a sealed class Personne
should be a permitted subtype of Personne.
```

Java

Les sous-classes d'une classe scellée doivent se déclarer

- soit `final`,
- soit `sealed`,
- soit `non-sealed`.

Java

Les sous-classes d'une classe scellée doivent se déclarer

- soit `final`,
- soit `sealed`,
- soit `non-sealed`.

Explication

Faites donc votre choix pour les classes `Etudiant` et `Enseignant`.

Java

Interface scellée [**Java 15**]

Une interface qui précise les classes et/ou les interfaces et/ou les records qui peuvent l'implémenter.

Java

Interface scellée [**Java 15**]

Une interface qui précise les classes et/ou les interfaces et/ou les records qui peuvent l'implémenter.

Remarques

- Les classes ou interfaces autorisées par une interface scellée doivent l'étendre/implémenter.
- L'interface scellée et les classes/interfaces autorisées doivent être déclarées dans le même package.

Java

Exemple

```
package org.eclipse.interfaces;

public sealed interface IAffichage permits IMiseEnForme {
        public void afficherNomPrenomEnMajuscule();
}
```

Java

Exemple

```
package org.eclipse.interfaces;

public sealed interface IAffichage permits IMiseEnForme {
        public void afficherNomPrenomEnMajuscule();
}
```

Dans ce cas, il faut spécifier si `IMiseEnForme` **est** `sealed` **ou** `non-sealed`

```
package org.eclipse.interfaces;

public non-sealed interface IMiseEnForme extends IAffichage {
        int i = 5;

        default public void afficherNomMajuscule() {
                System.out.println("Doe");
        }

        public void afficherPrenomMajuscule();
}
```

Java

Question

Pourquoi la classe qui hérite d'une classe `sealed` doit explicitement déclarer son propre type de permission de dérivation (`sealed`, `non-sealed`, ou `final`) ?

Java

Question

Pourquoi la classe qui hérite d'une classe `sealed` doit explicitement déclarer son propre type de permission de dérivation (`sealed`, `non-sealed`, ou `final`) ?

Réponse

En forçant les sous-classes à déclarer leur propre permission de dérivation, **Java** garantit que l'intention de conception derrière une classe scellée est maintenue tout au long de la hiérarchie d'héritage.

Java

Énumération

- Ensemble de constantes
- Introduite dans **Java 5**
- Pouvant être déclarée, dans une fichier toute seule, dans une classe ou dans une interface

Java

Énumération

- Ensemble de constantes
- Introduite dans **Java 5**
- Pouvant être déclarée, dans une fichier toute seule, dans une classe ou dans une interface

Pour créer une énumération sous **Eclipse**

- Aller dans `File > New > Enum`
- Saisir `org.eclipse.enums` dans `Package name`
- Saisir `Sport` dans `Name`
- Valider

Java

Code généré

```
package org.eclipse.enums;

public enum Sport {

}
```

Java

Code généré

```
package org.eclipse.enums;

public enum Sport {

}
```

Ajoutons des constantes à cette énumération `Sport`

```
package org.eclipse.enums;

public enum Sport {
        FOOT,
        RUGBY,
        TENNIS,
        CROSS_FIT,
        BASKET
}
```

Java

Pour utiliser cette énumération dans la classe `Main`

```
Sport sport = Sport.BASKET;
System.out.println(sport); // affiche BASKET
```

Java

Pour utiliser cette énumération dans la classe `Main`

```
Sport sport = Sport.BASKET;
System.out.println(sport); // affiche BASKET
```

N'oublions pas d'importer l'énumération dans la classe `Main`

```
import org.eclipse.enums.Sport;
```

Java

On peut aussi définir une énumération comme une classe avec un ensemble d'attributs et de méthodes

```
public enum Sport {

    FOOT("foot", 1),
    RUGBY("rugby", 7),
    TENNIS("tennis", 3),
    CROSS_FIT("cross_fit", 4),
    BASKET("basket", 6) ;
    private final String nom;
    private final int code;

    Sport(String nom, int code) {
        this.nom = nom;
        this.code = code;
    }
    public String getNom(){ return this.nom; }
    public int getCode(){ return this.code; }
};
```

Java

Récupérer la valeur de l'attribut `code` de la constante `BASKET`

```
Sport sport = Sport.BASKET;
System.out.println(sport.getCode());
// affiche 6
```

Java

Récupérer la valeur de l'attribut `code` de la constante `BASKET`

```
Sport sport = Sport.BASKET;
System.out.println(sport.getCode());
// affiche 6
```

Récupérer l'indice de la constante `BASKET` dans l'énumération

```
Sport sport = Sport.BASKET;
System.out.println(sport.ordinal());
// affiche l'indice ici 4
```

Java

Récupérer la valeur de l'attribut `code` de la constante `BASKET`

```
Sport sport = Sport.BASKET;
System.out.println(sport.getCode());
// affiche 6
```

Récupérer l'indice de la constante `BASKET` dans l'énumération

```
Sport sport = Sport.BASKET;
System.out.println(sport.ordinal());
// affiche l'indice ici 4
```

Transformer l'énumération en tableau et récupérer l'élément d'indice 2

```
System.out.println(Sport.values()[2]);
// affiche Tennis
```

Java

Classe : inconvénients

- Code souvent très long
- Code répétitif : getter et setter presque identiques pour tous les attributs
- Modifier un attribut $\Rightarrow$ modifier getter/setter, `toString`...

Java

Record

- Mot-clé introduit dans **Java 14**
- Simplifiant la création d'un modèle de données
- Défini comme une classe finale
- Plus besoin de définir getter, setter, constructeur ou `toString`
- Pouvant implémenter des interfaces (héritage d'une classe non autorisé)

Java

Record

- Mot-clé introduit dans **Java 14**
- Simplifiant la création d'un modèle de données
- Défini comme une classe finale
- Plus besoin de définir getter, setter, constructeur ou `toString`
- Pouvant implémenter des interfaces (héritage d'une classe non autorisé)

Record : limite

Les attributs sont des constantes : une fois initialisé, impossible de les modifier.

Java

Pour créer un record sous **Eclipse**

- Aller dans `File > New > Enum`
- Saisir `org.eclipse.record` dans `Package name`
- Saisir `Person` dans `Name`
- Valider

Java

Code généré

```
package org.eclipse.record;

public record Person() {

}
```

Java

Code généré

```
package org.eclipse.record;

public record Person() {

}
```

Ajoutons des attributs à ce record `Person`

```
package org.eclipse.record;

public record Person(int num, String nom, String prenom) {

}
```

Java

L'instanciation d'un record est équivalente à celle d'une classe

```
Person person = new Person(100, "wick", "john");
```

Java

L'instanciation d'un record est équivalente à celle d'une classe

```
Person person = new Person(100, "wick", "john");
```

Affichons l'objet `person`

```
System.out.println(person);
// affiche Person[num=100, nom=wick, prenom=john]
```

Java

L'instanciation d'un record est équivalente à celle d'une classe

```
Person person = new Person(100, "wick", "john");
```

Affichons l'objet `person`

```
System.out.println(person);
// affiche Person[num=100, nom=wick, prenom=john]
```

Pour accéder à un attribut on utilise le getter

```
System.out.println(person.prenom());
// affiche john
```

Java

Certaines méthodes opérateurs pour les classes sont aussi applicables pour les records

```
System.out.println(person instanceof Person);
// affiche true

System.out.println(person.getClass());
// affiche class model.Person

System.out.println(person.getClass().getName());
// affiche model.Person
```

Java

Comme pour les classes on peut définir plusieurs constructeurs

```
public record Person(int num, String nom, String prenom) {

        public Person(String nom, String prenom) {
                this(0, nom, prenom);
        }

        public Person(int num) {
                this(num, "doe", "john");
        }
}
```

Java

Testons tous les constructeurs

```
Person person = new Person(100, "wick", "john");
System.out.println(person);
// affiche Person[num=100, nom=wick, prenom=john]

Person person2 = new Person(200, "dalton", "jack");
System.out.println(person2);
// affiche Person[num=200, nom=dalton, prenom=jack]

Person person3 = new Person(300);
System.out.println(person3);
// affiche Person[num=300, nom=doe, prenom=john]
```

Java

On peut aussi définir un attribut statique

```
public record Person(int num, String nom, String prenom) {

        public static int nbrPersonnes = 0;

        public Person {
                nbrPersonnes++;
        }

        public Person(String nom, String prenom) {
                this(0, nom, prenom);
        }

        public Person(int num) {
                this(num, "doe", "john");
        }
}
```

Java

Testons dans `main`

```
System.out.println(Person.nbrPersonnes);
// affiche 0

Person person = new Person(100, "wick", "john");
System.out.println(Person.nbrPersonnes);
// affiche 1

Person person2 = new Person(200, "dalton", "jack");
System.out.println(Person.nbrPersonnes);
// affiche 2

Person person3 = new Person(300);
System.out.println(Person.nbrPersonnes);
// affiche 3
```

Java

Remarque

Un `Record` peut aussi contenir des méthodes personnalisées.