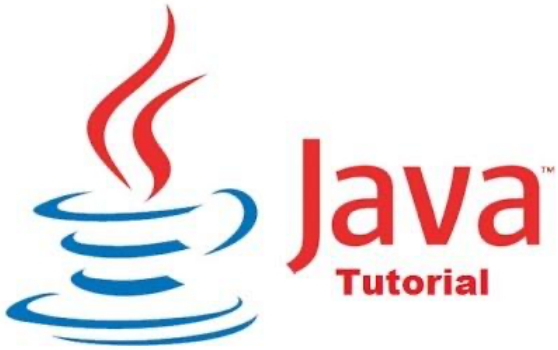
ÉCOLE
NATIONALE
INGÉNIEURS
TARBES

Programmation Orientée Objet

- 1IGENI-EC0941 -



Préparé par : Linda Elmhadhbi, Hedi Karray, Maroua Masmoudi

INP ENIT ÉCOLE NATIONALE D'INGÉNIEURS DE TARBES
47, av. d'Azereix BP 1629 tél: +33 (0)5 62 44 27 00
65016 Tarbes Cedex France fax: +33 (0)5 62 44 27 27
Université Fédérale Toulouse Midi-Pyrénées

Programmation Orienté Objet (POO)

La Programmation Orienté Objet (POO) consiste à modéliser une application informatique sous la forme d'objets, ayant des propriétés et pouvant interagir entre eux.
En POO, on identifie les acteurs (les entités comportementales) du problème puis on détermine la façon dont ils doivent interagir pour que le problème soit résolu. Cette façon de modéliser les choses permet également de découper une application gestion, généralement floue, en une multitude d'objets afin de la développer plus facilement.

Un programme Orienté Objet est écrit à partir d'un langage Orienté Objet (OO). Les langages O.O. les plus utilisés sont : C++, C#, Java, PHP, Python, etc.
Dans ce Tutoriel, nous allons utiliser le langage Java.

En POO, programmer revient donc à décrire des **classes d'objets**, à caractériser leur **structure** et leur **comportement**, puis à **instancier** ces classes pour créer des **objets réels** et ensuite les **manipuler**.

Qu'est-ce qu'une classe ?

Le monde réel regroupe des objets du même type. Il est pratique de concevoir une maquette (un moule) d'un objet et de produire les objets à partir de cette maquette. En POO, une maquette se nomme une classe. Une classe est donc une abstraction commune d'un ensemble d'objets ayant les mêmes caractéristiques (attributs). Par exemple, la voiture de Sara qui est une Nissan immatriculée CX344AB, la voiture de Pierre qui est une BMW immatriculée DF345OP et la voiture de David qui est une Mercedes immatriculé DF980PO sont des voitures (objets) ayant des caractéristiques en commun comme le propriétaire, la marque et le numéro d'immatriculation et donc sont considérées comme des instances de la **classe voiture**.

Une classe regroupe un ensemble de données et un ensemble de méthodes de traitement de ces données et/ou de données extérieures à la classe. On parle d'encapsulation pour désigner le regroupement de données dans une classe.
Une classe se compose de deux parties : un en-tête et un corps. L'en-tête déclare le nom de la classe. Le corps peut être divisé en 2 sections :

- La déclaration des attributs qui décrivent les objets. Les attributs sont des variables qui stockent des informations sur les propriétés de l'objet.
- La déclaration des méthodes qui décrivent les opérations qui sont applicables aux instances de la classe. Parmi les méthodes on trouve le(s) constructeurs et les getters et setters.

Graphiquement, la classe **« Compte »,** par exemple, serait représentée de la manière suivante (Figure 1) :

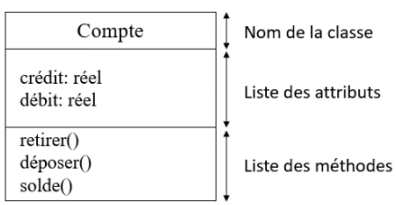


Figure 1. Représentation graphique d'une classe

Les méthodes et les attributs sont pourvus de modificateurs appelés aussi attributs de visibilité qui gèrent leur accessibilité par les composants hors de la classe. Ça consiste donc à définir des étiquettes pour les attributs et les méthodes afin de préciser si celles-ci sont accessibles à partir d'autres classes ou non.

Les attributs de visibilité sont :

Modificateur	Rôle
« Public »	L'attribut est accessible depuis toutes les autres classes et sans aucune restriction.
« Private »	L'attribut est accessible uniquement depuis sa propre classe.
« Protected »	L'attribut est accessible depuis une classe parente et ses classes dérivées

Qu'est-ce qu'un objet ?

Un objet est une instance de classe. Les attributs décrivent les propriétés de l'objet. Les méthodes décrivent le comportement de l'objet.

Pour qu'un objet ait une existence, il faut qu'il soit instancié. Une même classe peut être instanciée plusieurs fois, chaque instance ayant des propriétés ayant des valeurs spécifiques. Cette instanciation se fait à travers une méthode appelée ***constructeur***.

Reprenons l'exemple de la classe « Voiture », nous pourrons avoir plusieurs objets voitures (instances) bien distincts. La voiture de Jonathan, qui est une BMW de couleur rouge, est un objet. De même, la voiture de Louise, qui est une Ferrari de couleur blanche, est un autre objet. Nous pouvons donc avoir plusieurs objets pour une même classe (autrement dit : deux instances de la même classe).

Qu'est-ce qu'un constructeur ?

Pour instancier une classe, c'est-à-dire créer un objet à partir d'une classe, on fait appel à une méthode spéciale de la classe appelée constructeur.
Cette méthode particulière peut être vide pour effectuer les opérations nécessaires à l'initialisation d'un objet. Chaque constructeur doit avoir le même nom que la classe où il est défini et n'a aucune valeur de retour [pas de *return*] (c'est l'objet créé qui est renvoyé).

Exemples :

Constructeur avec paramètres

```
public class Compte {

// attributs
   private float credit;
   private float debit;
// constructeur

     public Compte (float c, float d) {
         credit = c;   // affectation de la valeur de c dans l'attribut credit
         debit = d;   // affectation de la valeur de d dans l'attribut debit

    }
```

Constructeur sans paramètres

```
 // attributs
  private float credit ;
  private float debit ;
// constructeur

    public Compte () {

   }
```

Qu'est-ce que les accesseurs (Getters) et les mutateurs (Setters) ?

Un accesseur (Get) est une méthode permettant de récupérer le contenu d'un attribut privé (private).

Un mutateur (Set) est une méthode permettant de modifier le contenu d'un attribut privé (private).

Qu'est-ce qu'un héritage

Une particularité notable dans l'organisation des classes en programmation orientée objets est l'héritage de propriétés. L'héritage est un mécanisme permettant à une nouvelle classe de posséder automatiquement les variables et les méthodes de la classe dont elle dérive.
Les attributs et méthodes de la classe mère sont accessibles depuis la classe fille (si les accesseurs sont « public » ou « protected »).

Le mot clé « extends » permet de spécifier une superclasse éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage.

Graphiquement, la classe « Compte Epargne » qui hérite de la classe « Compte » par exemple serait représentée de la manière suivante (Figure 2) **:**

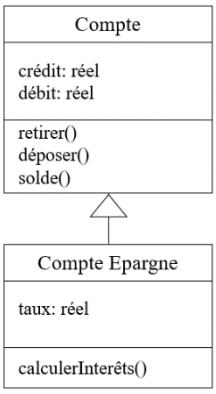


Figure 2. Représentation graphique d'un héritage entre deux classes

La classe « compte » est appelée classe mère et la classe « compte épargne » est appelée classe fille.

Qu'est-ce qu'une méthode principale (main) ?

Cette méthode main a un statut particulier : c'est elle qui est appelée en premier quand on lance un programme Java.

```
public static void main (String[] args).
```

Le programme va exécuter les instructions qui sont à l'intérieur de cette méthode. Ces instructions contiennent généralement l'instanciation des objets et font appel aux méthodes développées dans les différentes classes.

Qu'est-ce qu'un élément statique ?

On appelle élément statique d'une classe tout élément attaché à cette classe plutôt qu'à l'une de ses instances. Un élément statique peut exister, être référencé, ou s'exécuter même si aucune instance de cette classe n'existe. Un élément statique ne peut référencer *this*. Il est possible de définir quatre types d'éléments statiques :

- des champs statiques : la valeur de ce champ est la même pour tous les objets instance de la classe ;
- des méthodes statiques ;
- des blocs statiques ;
- des classes membre statiques.

Par défaut, chaque instance de chaque classe occupe son propre espace mémoire. Deux objets instances d'une même classe occupent donc deux espaces mémoire qui ne se recouvrent pas. Lorsque l'on déclare un membre statique *static,* il est au contraire placé dans un espace mémoire commun à tous les objets de la classe. Si un des objets modifie la valeur d'un champ statique (par exemple), tous les objets verront la valeur de ce champ modifiée. Nous avons déjà rencontré une méthode statique : la méthode main, appelée lors du lancement d'une application Java. Dans la mesure où un élément statique ne dépend d'aucune instance de la classe à laquelle il appartient, il est possible de l'appeler avec une syntaxe particulière, sans passer par une instance particulière de la classe. Il est même possible d'invoquer un élément statique d'une classe sans que celle-ci n'ait jamais été instanciée. Il est une bonne pratique de n'appeler les membres statiques d'une classe que de façon "statique", c'est-à-dire en utilisant le nom de la classe plutôt qu'une de ses instances : *nomDeLaClasse.methodeStatique() ;*

Qu'est-ce qu'un *this* ?

Le mot-clé *this* désigne en permanence l'objet dans lequel on se trouve. Il existe dès l'instant que l'on se trouve dans l'instance d'une classe. Il ne peut pas être défini dans un élément statique. Ce mot-clé est utilisé pour lever l'ambiguïté qui peut exister entre le champ d'une classe et un paramètre d'une méthode dans laquelle on se trouve comme dans cet exemple :

```
 public class Compte {

// attributs
   private float credit;
   private float debit;
// constructeur

     public Compte (float credit, float d) {
         this.credit = credit;
         this.debit = debit;   // utilisation de this pour lever l'ambigüité
                               // entre le paramètre de la fonction () et le champ

         }
    }
```

Instanciation d'un objet

Pour instancier un objet d'une classe, il faut déclarer une variable (objet) de type la classe et faire appel au constructeur. Dans l'exemple suivant on peut voir l'instanciation de deux objets de la classe compte.

```
public static void main (String[] args) {

// création de l'objet c1 de type compte avec le constructeur vide
public Compte C1= new Compte() ;

// création de l'objet c2 de type compte avec le constructeur à paramètres
public Compte C2= new Compte(3500.23, 2500.10) ;

}
```

UML : un langage de modélisation pour la POO

Programmer en orientée objet est un peu difficile au début. On ne sait pas trop comment lier nos classes ni comment conceptualiser et formaliser notre domaine d'étude. L'UML (Unified Modeling Language) est justement l'un des moyens pour y parvenir. C'est un langage de représentation graphique qui permet de modéliser conceptuellement les classes et leurs interactions. Concrètement, cela s'effectue par le biais des diagrammes. Il existe plusieurs types de diagrammes dans UML. Dans ce tutoriel, on va s'intéresser au diagramme de classe.

Le diagramme de classe montre une collection d'éléments statiques (classes), leur contenu (attributs et méthodes) et les relations entre eux.

Les associations binaires (les relations entre classes) sont représentées par des lignes connectant les classes. La terminaison d'une association où elle se connecte à une classe est appelée "cardinalité". La cardinalité d'une association permet de représenter le nombre minimum et maximum d'instances qui sont autorisées à participer à la relation. Il existe plusieurs types de cardinalité :

0..1	Aucune ou une instance
1	Une instance exactement
0..* ou **0..n**	Aucune ou plusieurs instances
1..n ou **1..***	Une instance ou plusieurs (au moins une)

Par exemple, le diagramme de classe de la figure 3 nous montre que :

- Un client peut posséder un ou plusieurs comptes (1...*) et que le compte est possédé par un seul client (1).
- Une banque gère plusieurs comptes (1...*) et un compte est géré par une seule banque (1).
- Chaque compte est associé à un seul RIB et un RIB est associé à un seul Compte.

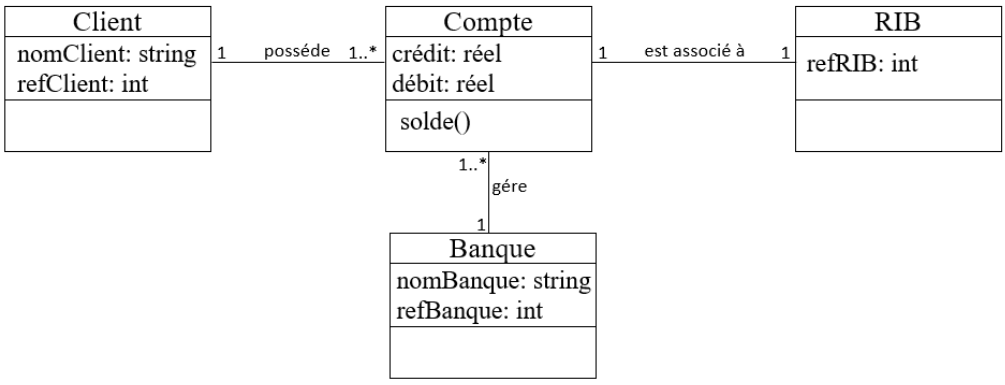


Figure 3. Exemple de diagramme de classe

Les cardinalités des associations impactent le contenu des classes en termes d'attributs quand on passe à la programmation. Le fait qu'une banque gère plusieurs comptes (1...*) et un compte est géré par une seule banque (1), se traduit par l'ajout d'un attribut *banque* de type "Banque"

dans la classe *Compte* ainsi que l'ajout d'un attribut *listeCompte* de type liste contenant des objets *Compte* dans la classe Banque comme indiqué sur la figure 4.

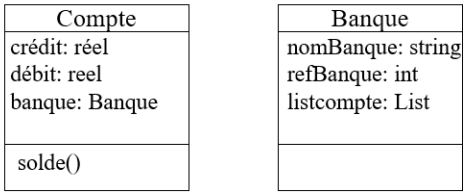


Figure 4. Exemple de transformation des cardinalités en attributs

Pour modéliser un diagramme de classe, il existe plusieurs logiciels de modélisation UML gratuits tels que: StarUML, Visual Paradigm Online, ArgoUML, etc.

Mini projet découverte

Dans ce tutoriel, l'objectif est de développer une petite application de gestion de commande. Le diagramme de classe est présenté dans la figure 5. Il est composé de six classes : client, produit, produitAlimentaire, produitNonAlimentaire, ligneCommande et commande.
Un client possède comme attributs un *id*, un *nom*, un *prénom*, un *numéro de téléphone*, et a un *nombre de points de fidélité*.
Un produit possède comme attributs une *référence*, un *nom*, une *marque*, un *prix*, et une *quantité*.
Il y a deux sortes de produits : produit alimentaires et produits ménagers qui héritent de la classe produit.
La classe ligne commande représente chaque item d'une commande. Elle possède comme attributs : Un produit, une quantité.
Une commande concerne un ou plusieurs produits. Elle possède comme attributs un *numéro*, la liste des lignes commande (produit, quantité), un *prix total*, une *date*. On peut aussi connaître si la commande sera livrée à domicile ou non.

L'application à développer ensemble doit être capable de :

- Créer / modifier / supprimer des instances des chaque classe.
- Établir un programme de fidélité et calculer le nombre de points de fidélité de chaque client.
- Calculer le prix total d'une commande, la réduction (par rapport au programme de fidélité) et le prix après réduction.
- Afficher le stock des produits

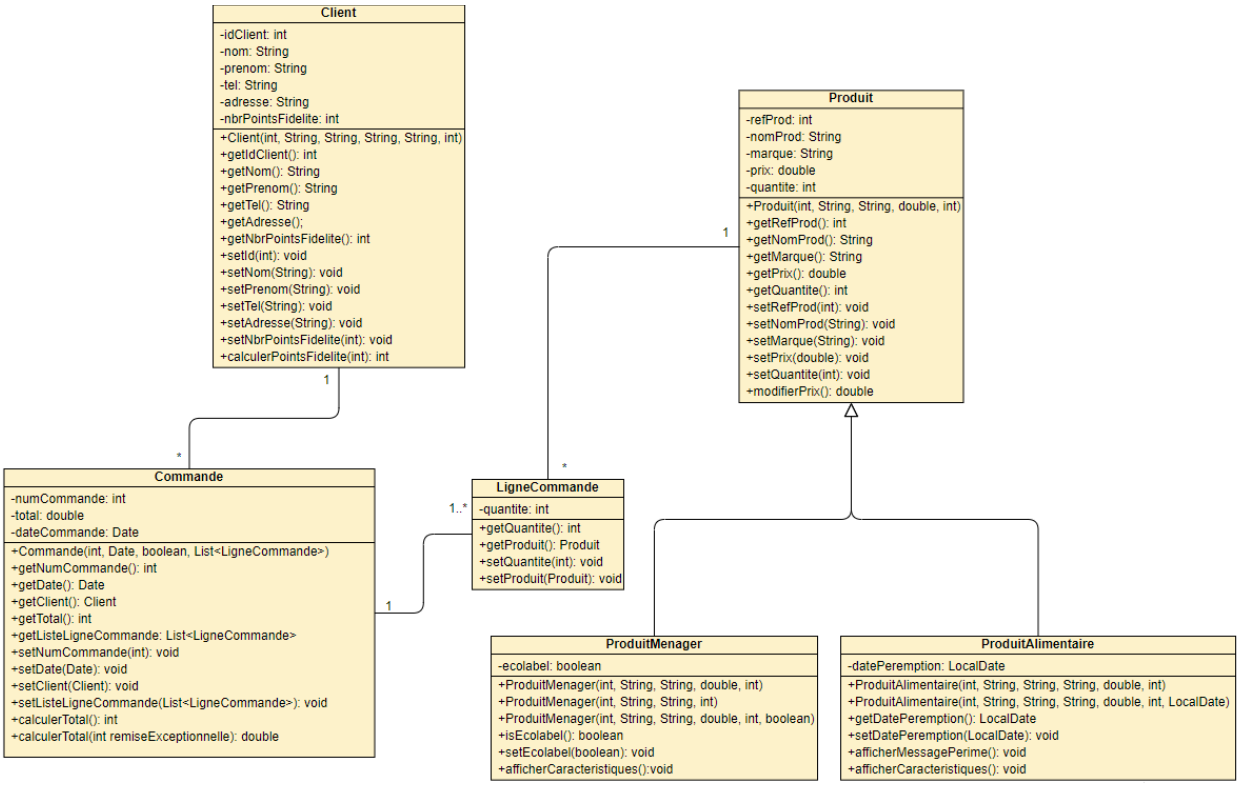


Figure 5. Diagramme de classe : Gestion des commandes

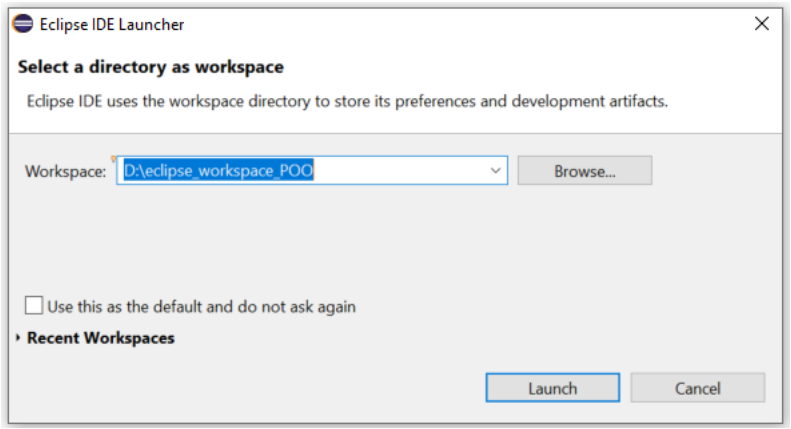
De UML vers Java (Mini projet)

Pour coder l'application en Java, nous utilisons comme environnement de développement intégré (IDE) **Eclipse.**

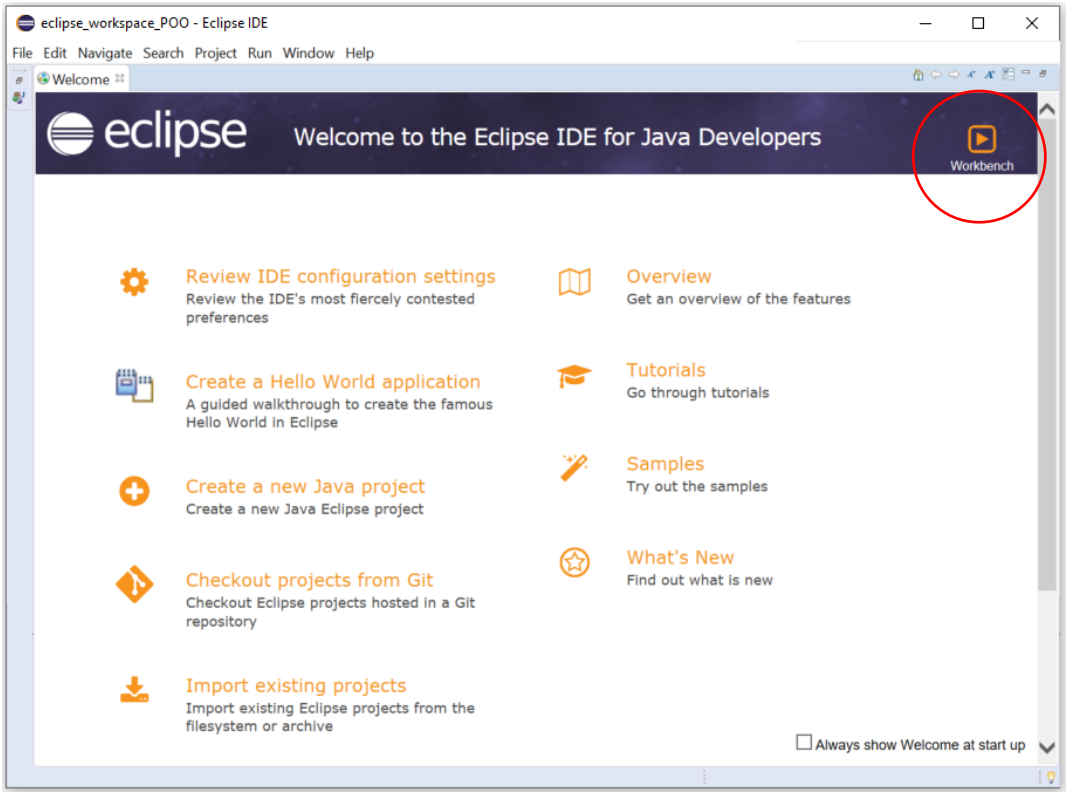
Partie I : Démarrage d'Eclipse

Lancement d'Eclipse

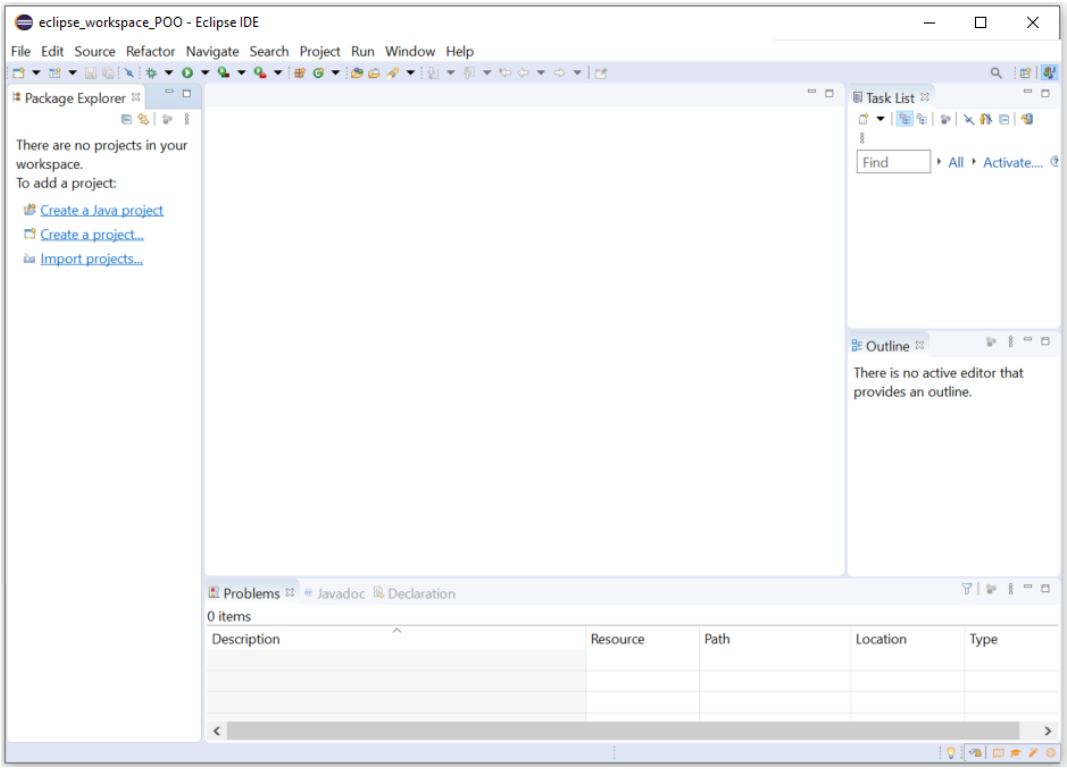
Une fois Eclipse est lancé, un écran apparaît ensuite pour vous demander le '**workspace**' (Répertoire de travail). C'est dans ce répertoire que Eclipse stockera les programmes (*.java) et fichiers de configuration (*.xml, *.properties, ...) de vos projets.
Pour ne pas perdre vos travaux quand vous fermez votre session en salle de TP ou par VPN, enregistrez le workspace sur le *bureau* ou dans *Mes documents* après avoir cliqué sur *Browse*. Puis cliquez sur le bouton launch.



La page d'accueil d'Eclipse s'affiche.
Cliquez sur l'icône Workbench représentant une flèche (en haut à droite).



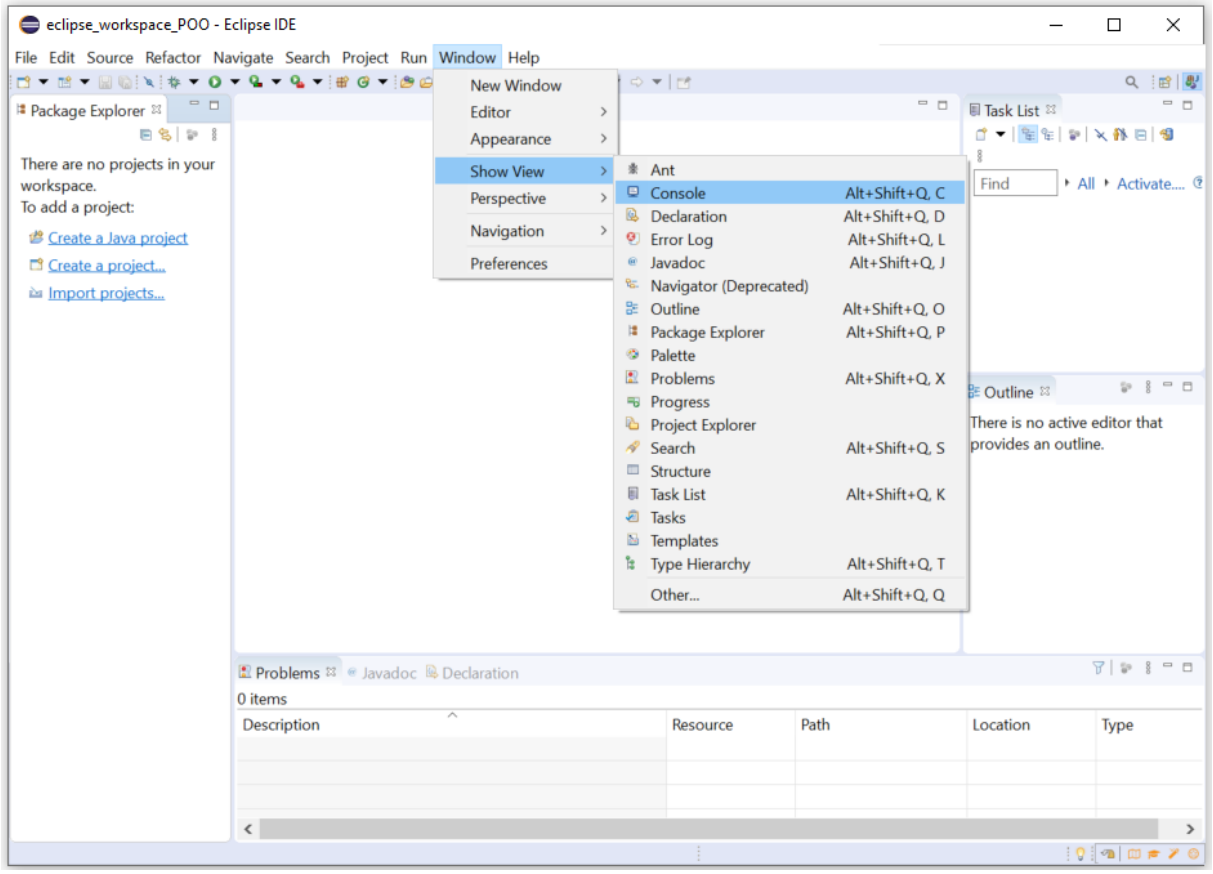
L’espace de travail (Workbench) s’ouvre.



Ça y est, Eclipse est lancé. Il faut maintenant comprendre deux notions : Perspective et vue.

Partie II : Perspectives et vues

Une fois Eclipse est lancé, vous vous trouvez dans la “perspective” Java.
Une perspective est en ensemble de ’vues’.
Une vue est une petite fenêtre possédant des informations particulières. Par exemple, vous pouvez voir les vues “Problems”, “Javadoc” et “Declaration”.
La vue console est équivalente à l’invite de commande MS-DOS.
Pour afficher la vue Console, il suffit de cliquer sur window/show view/console.

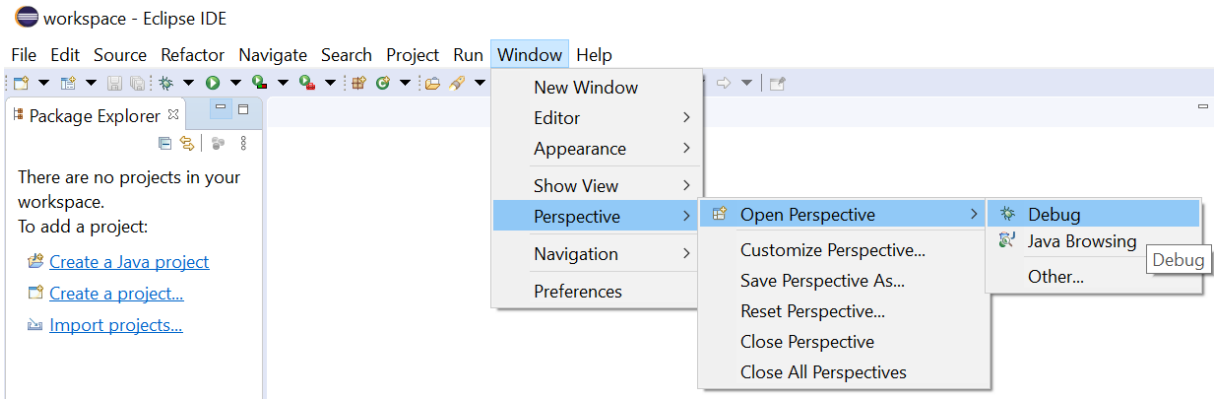


La vue console s’affiche à la suite des autres vues.

S’il y a un problème ou une erreur dans le code, il est possible d’exécuter le code afin de détecter l’instant précis où la divergence commence en utilisant le debugger d’Eclipse.
Un debugger permet d’exécuter un programme pas-à-pas afin d’en vérifier le comportement et l’état.

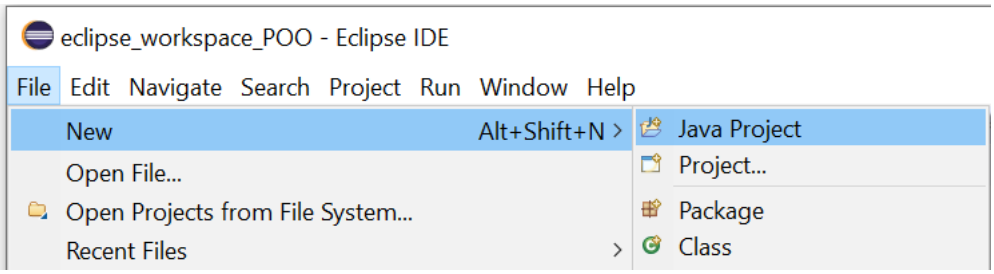
Pour lancer le debugger d'Eclipse, cliquez sur Window/Perspective/Open perspective/Debug. Vous remarquez alors que tout l'environnement Eclipse change de peau. Plusieurs 'vues' apparaissent, comme la vue 'debug', la vue 'variables' ou encore la vue 'Breakpoint' (points d'arrêt).

Revenez à la perspective 'JAVA' en cliquant en haut à droite sur JAVA, ou alors Window/Show perspective/Java.



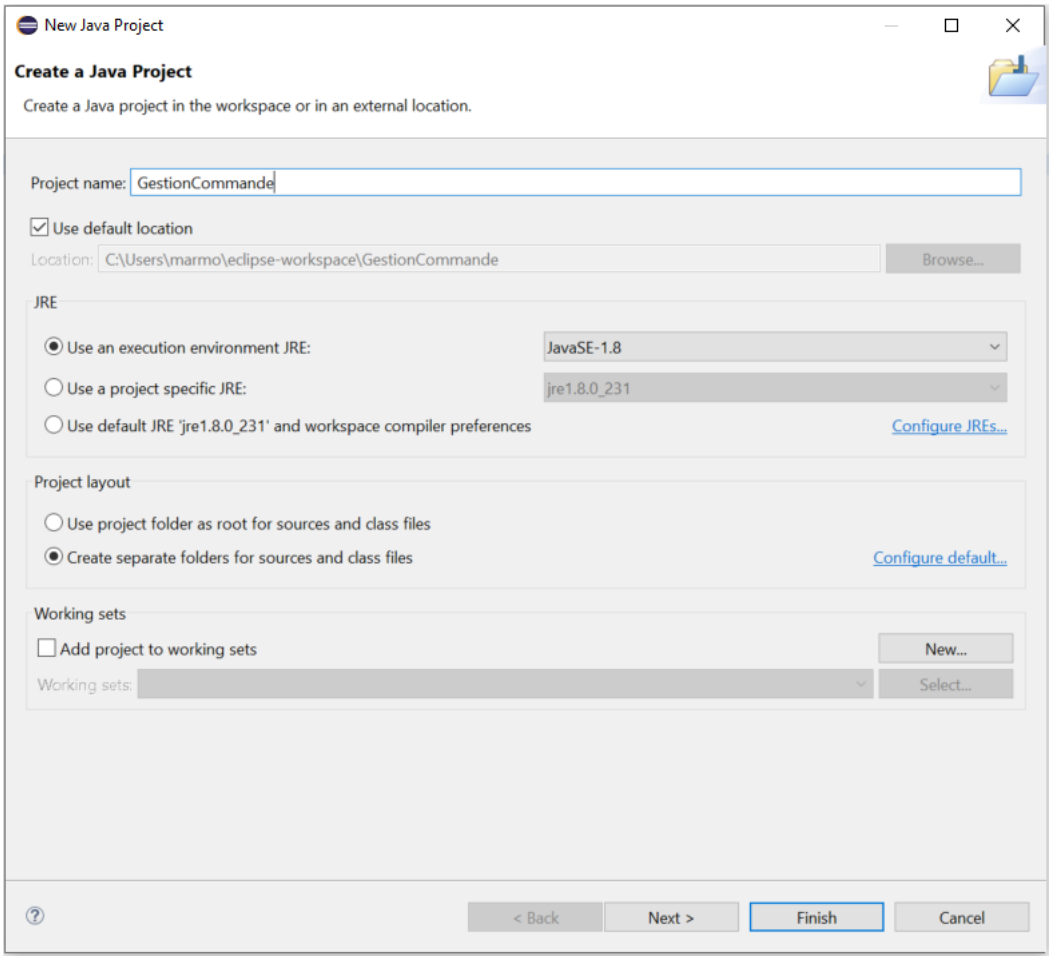
Partie III : Création projet Java

Pour créer un nouveau projet, cliquez sur Menu File/new/Java project.



L'écran suivant s'affiche.

Entrez le nom du projet dans le champ "Project name". Dans notre exemple, le nom du projet est GestionCommande.

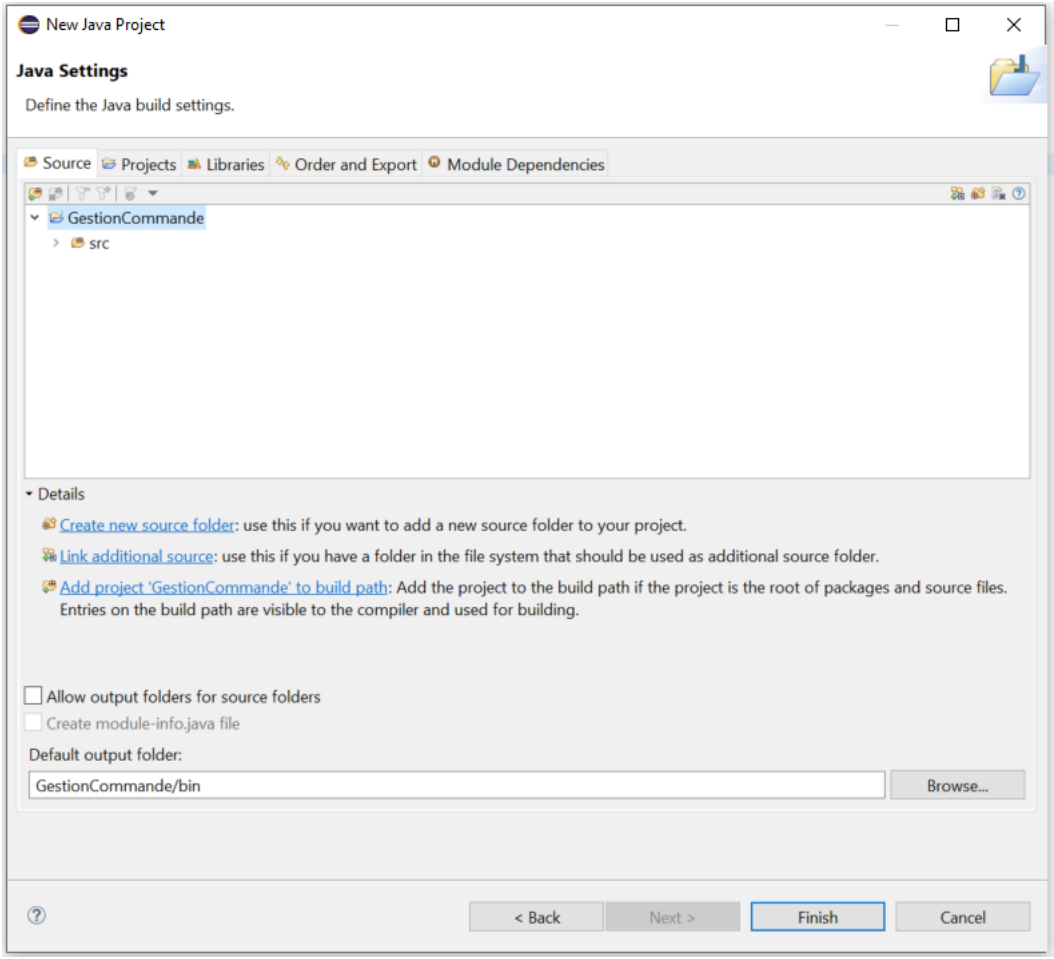


REMARQUE : Eclipse a identifié l'environnement Java préinstallé sur votre ordinateur. Sélectionnez 'Use Default JRE'. Pour les autres options vous pouvez ne pas toucher.

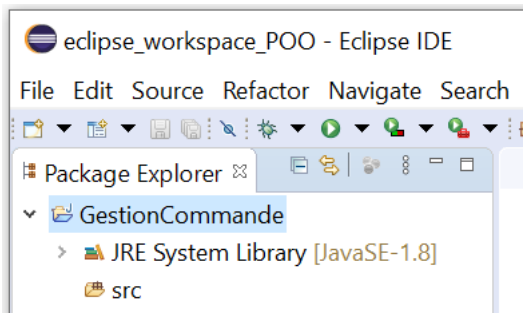
REMARQUE : Eclipse par défaut va créer dans le répertoire Workspace un répertoire spécifique (src) pour les fichiers .java, et un autre répertoire spécifique (bin) où il mettra les fichiers compilés (*.class). Les classes que vous allez développer seront donc stockées dans des fichiers nomClasse.java dans le répertoire scr.

Cliquez sur Next.

Cliquez sur Finish.



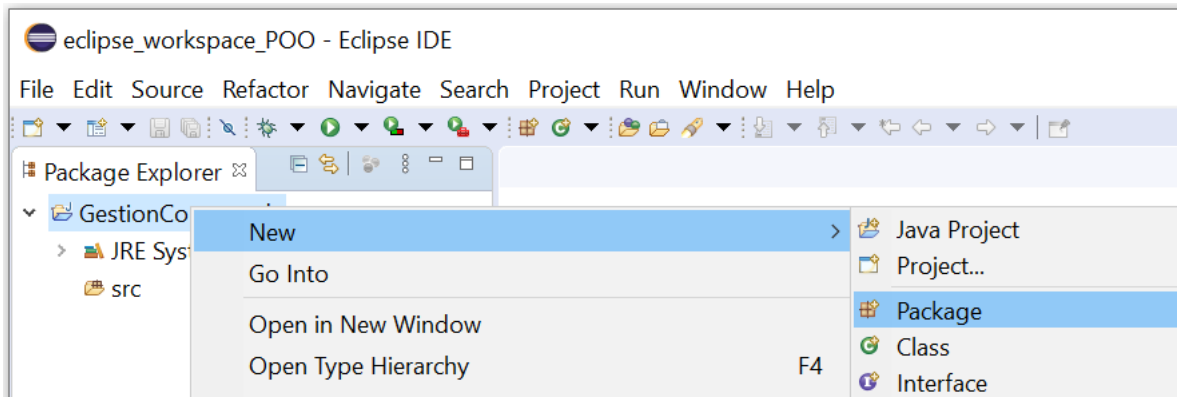
Eclipse crée alors le squelette du projet.



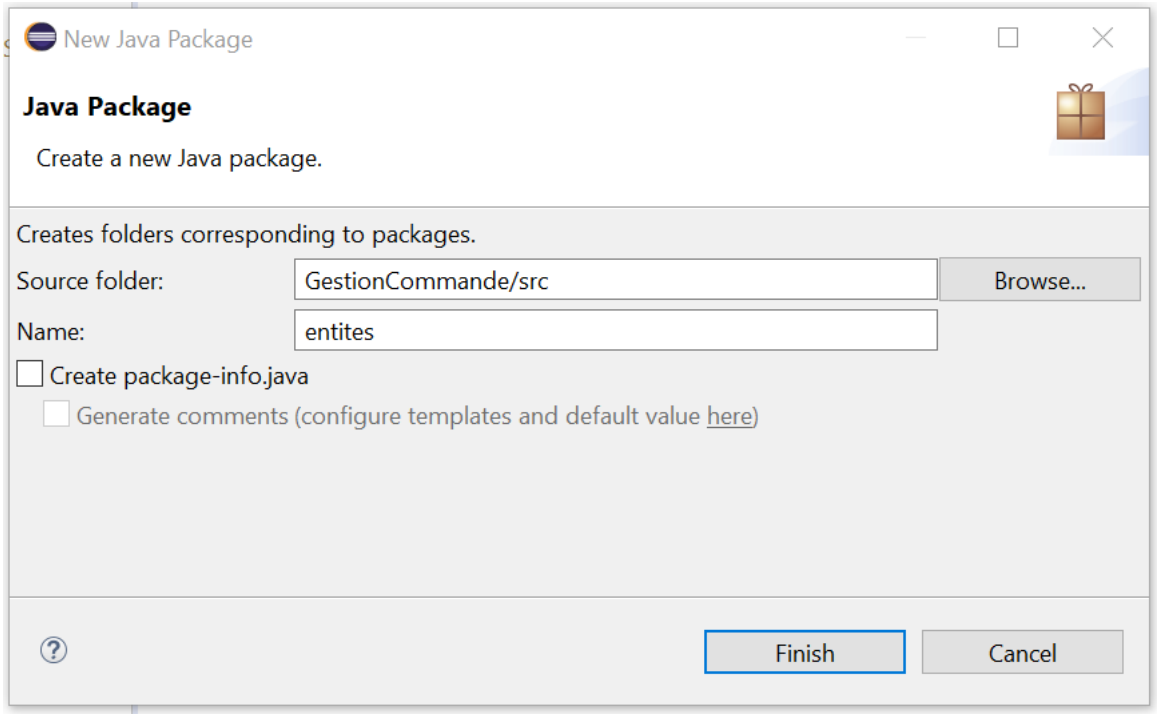
Création d'un package

Il vous faut désormais créer un "package" Java dans lequel vous allez ranger/créer la classe principale de l'application. En effet, un package est une unité (un fichier) regroupant des classes.

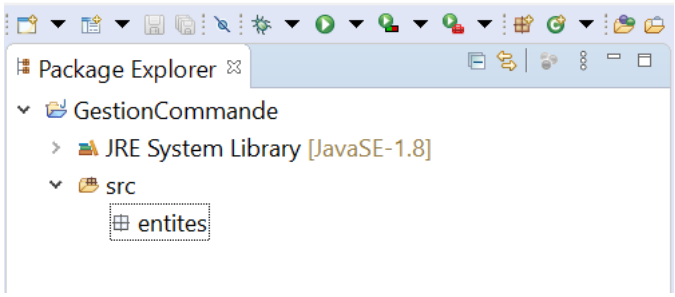
Pour créer un package, cliquez droit sur le projet (GestionCommande) dans la vue "project Explorer" puis new/package.



L'écran suivant s'affiche.
Entrez "entites" comme nom de package. Puis cliquez sur le bouton Finish.

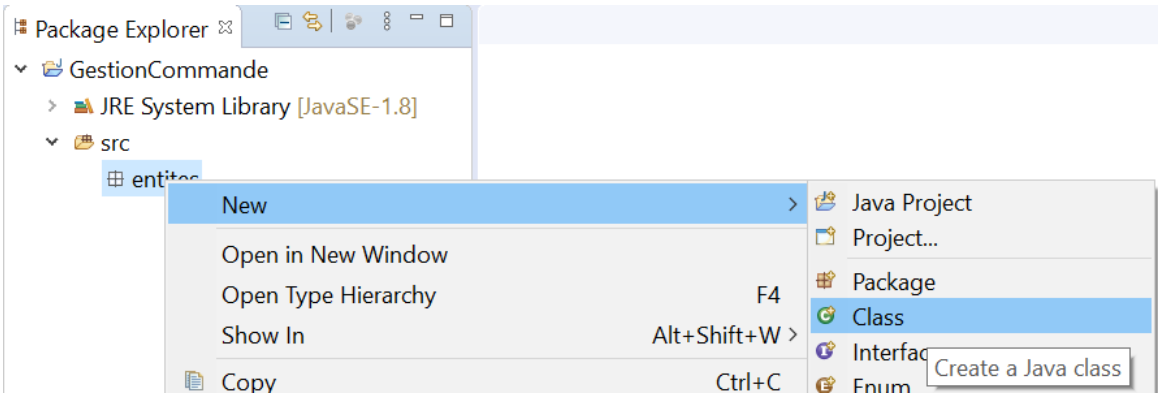


Eclipse crée alors un package.

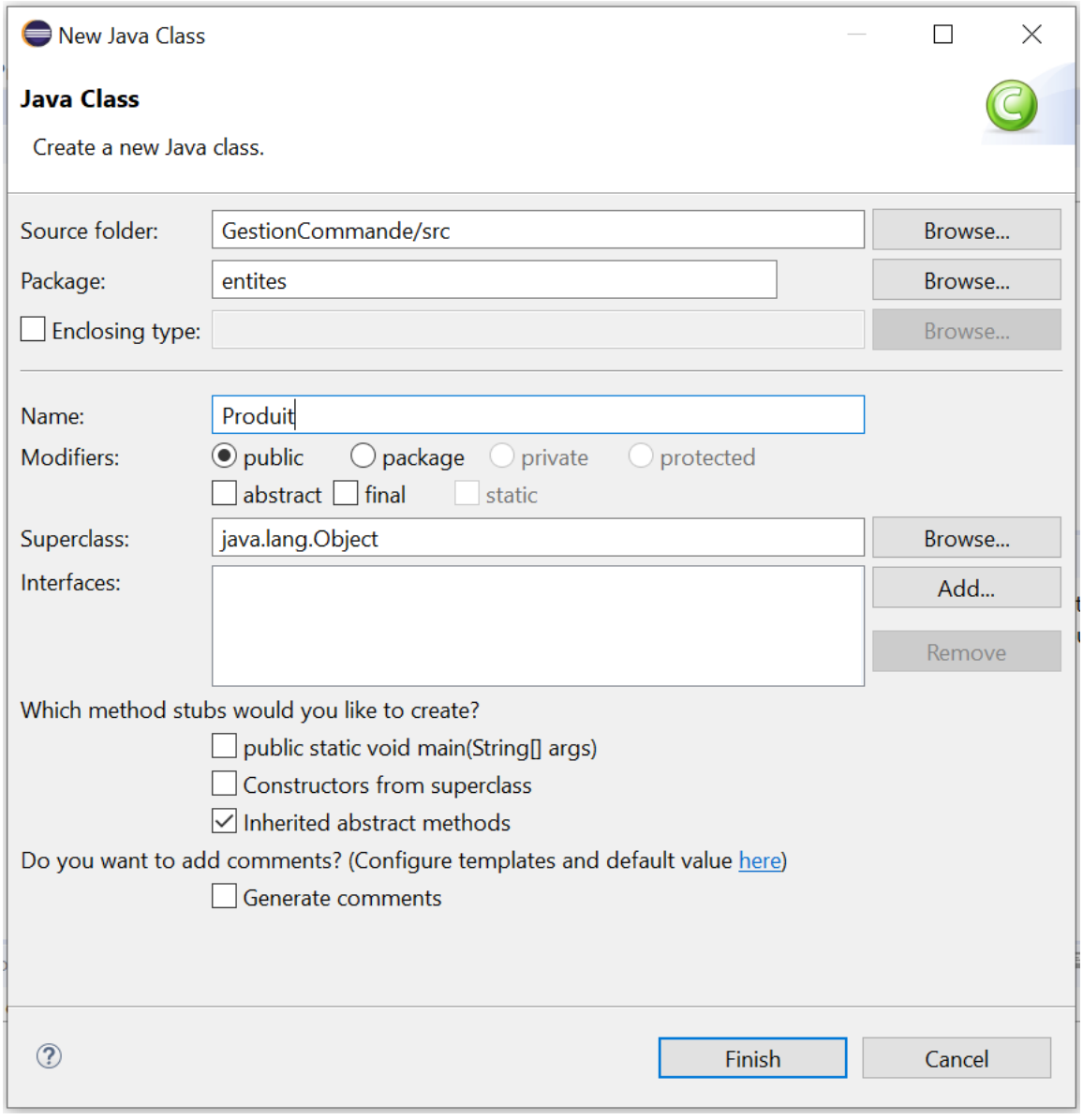


Création d'une classe

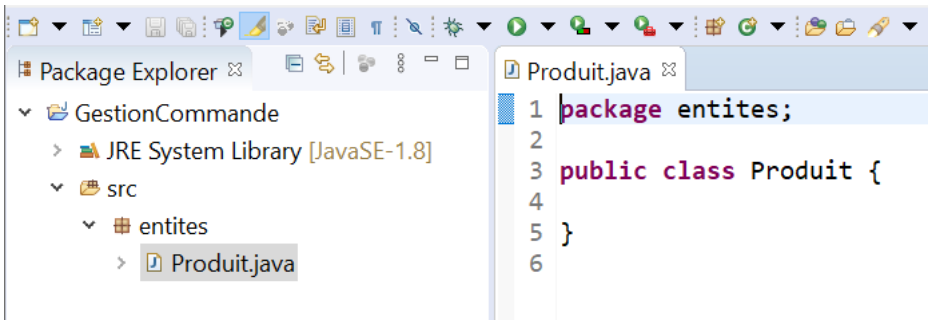
Il vous faut désormais créer une classe principale.
Pour ce faire, cliquez droit sur le package "entites", puis new/class.



Écrire le nom de la classe, ici "Produit" et cliquez sur le bouton Finish.



La classe “Produit” est créée dans le package “entites”.



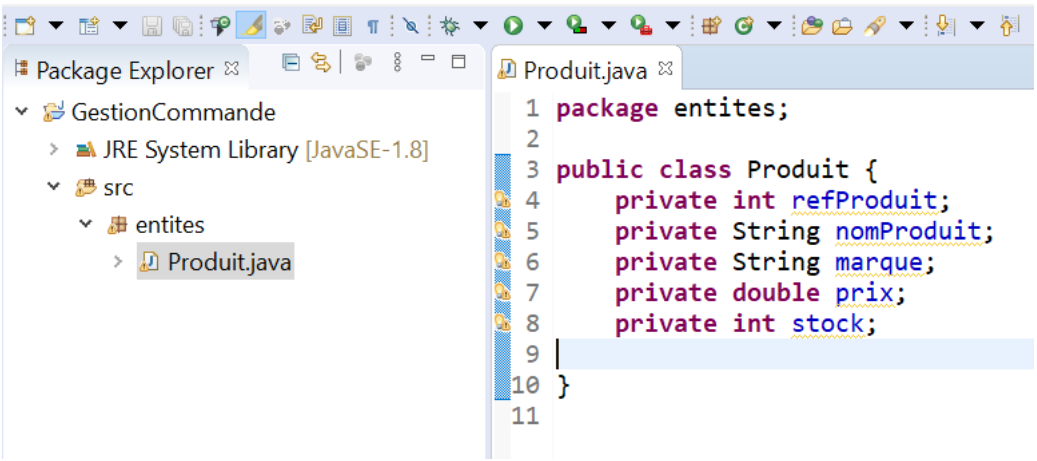
Ajoutez les attributs de la classe “Produit” qui sont :

- refProduit de type entier (int)
- nomProduit de type chaîne de caractères (String)
- marque de type chaîne de caractères (String)
- prix de type double (double)
- stock de type entier (int)

Pour déclarer un attribut d’une certaine classe, il suffit de choisir son attribut de visibilité (e.g. “private”), son type (e.g. “int”) et son nom selon la syntaxe suivante :

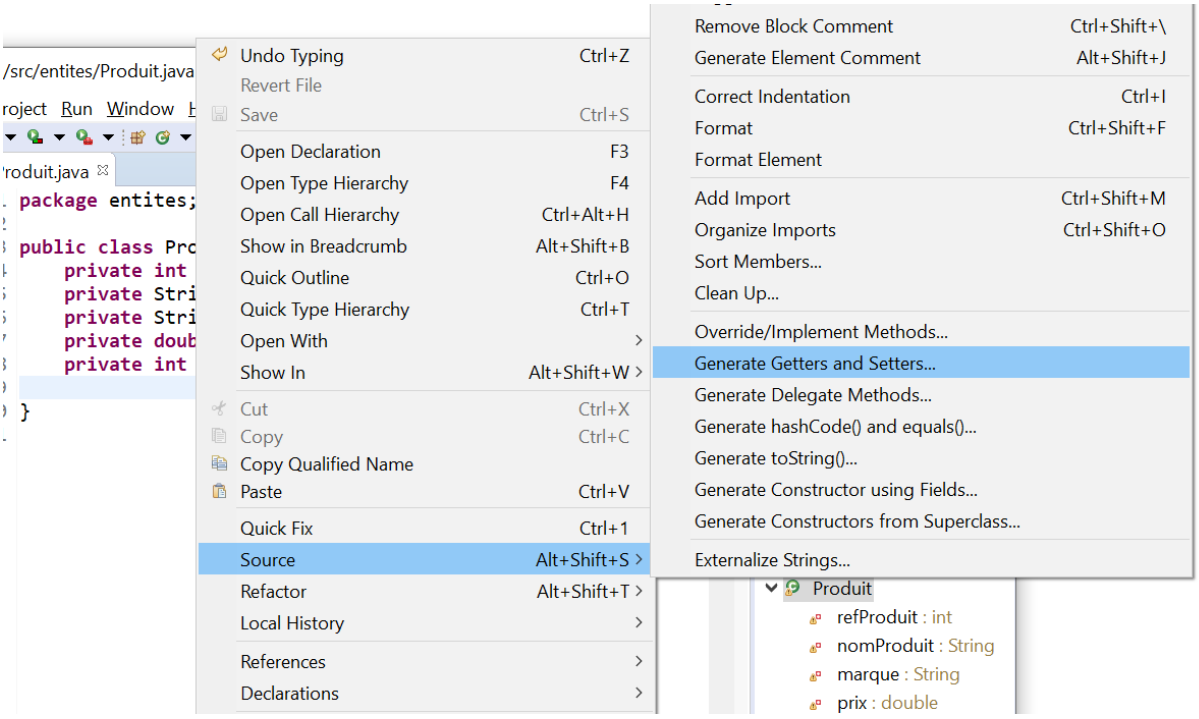
visibilité_Attribut type_Attribut nom_Attribut ;

- N’oubliez pas le point-virgule à la fin.
- Contrairement aux types int, long, float, double, boolean... le type String commence toujours par une majuscule.

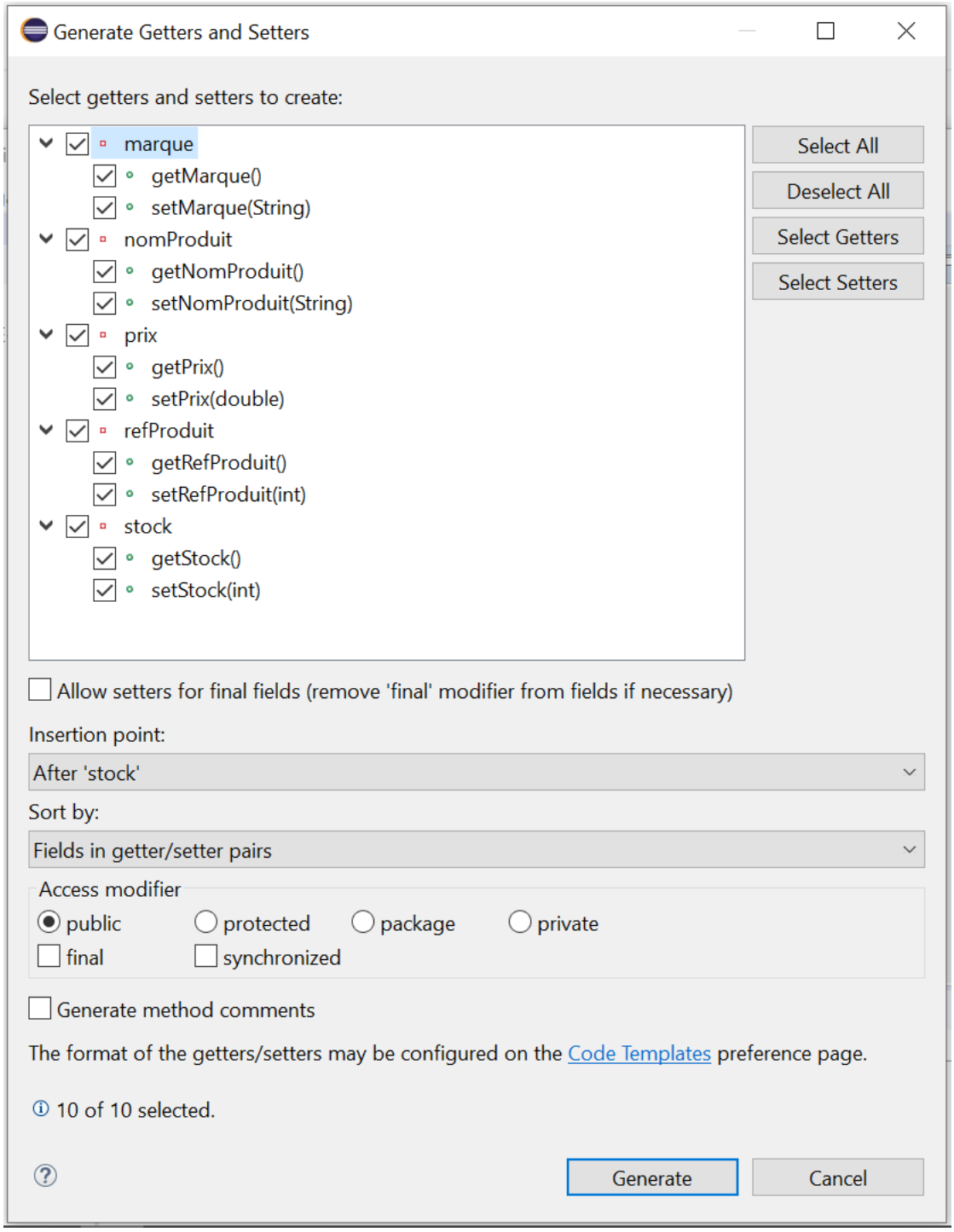


Génération des Getters et Setters :

Faites un clic droit à l'intérieur de la classe et sélectionnez "Source\Generate Getters and Setters" pour générer les getters et setters de la classe.



Vous pouvez sélectionner tous les attributs en cliquant sur le bouton "SELECT ALL" en haut à droite. Puis cliquez sur le bouton "Generate".


Generate Getters and Setters
Select getters and setters to create:
marque
getMarque()
setMarque(String)
nomProduit
getNomProduit()
setNomProduit(String)
prix
getPrix()
setPrix(double)
refProduit
getRefProduit()
setRefProduit(int)
stock
getStock()
setStock(int)
Select All
Deselect All
Select Getters
Select Setters
Allow setters for final fields (remove 'final' modifier from fields if necessary)
Insertion point:
After 'stock'
Sort by:
Fields in getter/setter pairs
Access modifier
public
protected
package
private
final
synchronized
Generate method comments
The format of the getters/setters may be configured on the Code Templates preference page.
10 of 10 selected.
Generate
Cancel

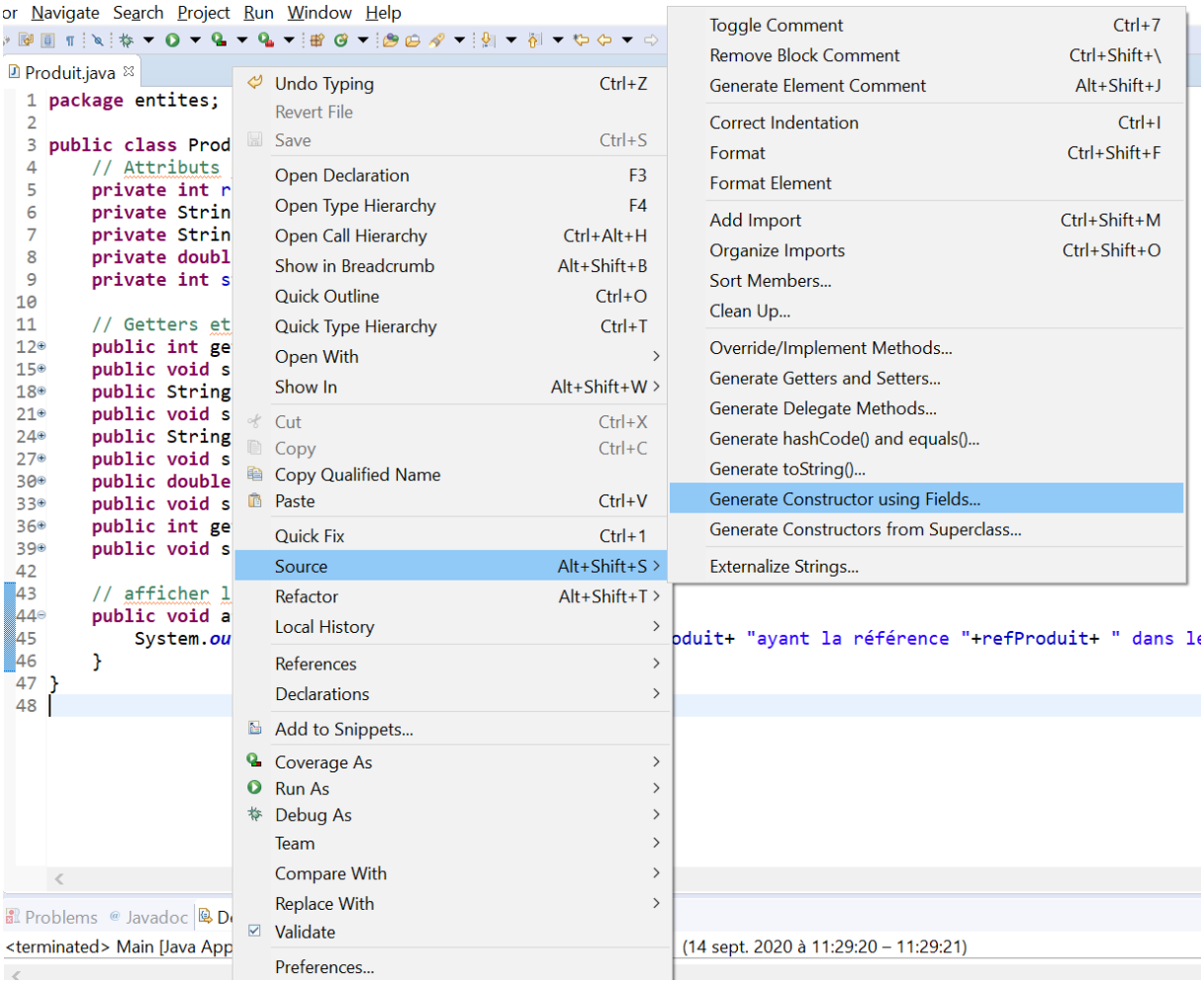
Les getters et les setters sont générés comme ci-dessous.

*Produit.java

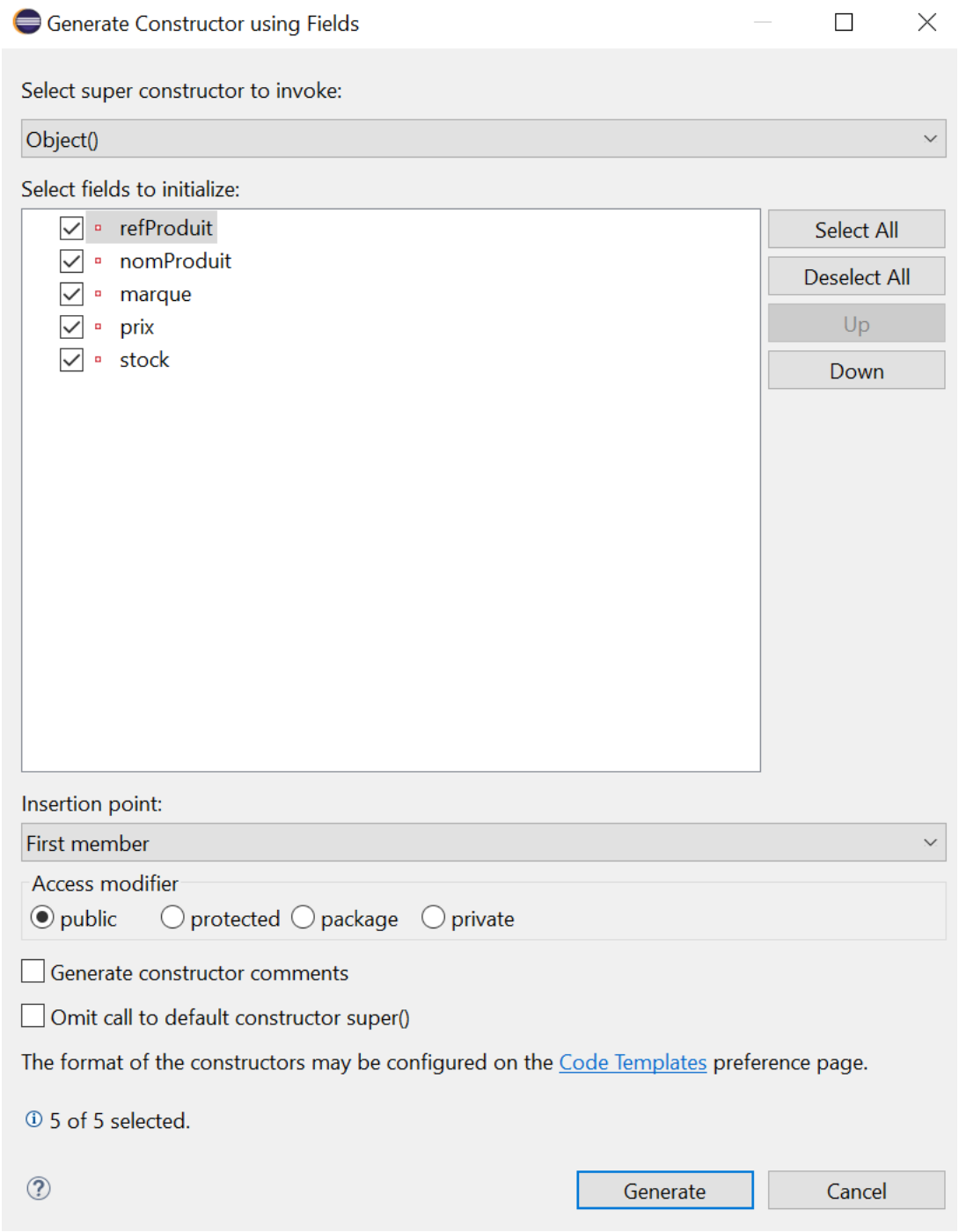
```java
package entites;

public class Produit {
    private int refProduit;
    private String nomProduit;
    private String marque;
    private double prix;
    private int stock;
    public int getRefProduit() {
        return refProduit;
    }
    public void setRefProduit(int refProduit) {
        this.refProduit = refProduit;
    }
    public String getNomProduit() {
        return nomProduit;
    }
    public void setNomProduit(String nomProduit) {
        this.nomProduit = nomProduit;
    }
    public String getMarque() {
        return marque;
    }
    public void setMarque(String marque) {
        this.marque = marque;
    }
    public double getPrix() {
        return prix;
    }
    public void setPrix(double prix) {
        this.prix = prix;
    }
    public int getStock() {
        return stock;
    }
    public void setStock(int stock) {
        this.stock = stock;
    }

}

```

Génération d'un constructeur :

Pour ajouter un constructeur, faites un clic droit à l'intérieur de la classe après les Getters et Setters puis cliquez sur "Source\Generate Constructor using Fields".



Sur l'écran, cliquez sur le bouton 'SELECT ALL' pour sélectionner tous les attributs, puis cliquez sur "Generate".



Le constructeur de la classe *Produit* est généré.

*Produit.java

```java
package entites;

public class Produit {

    // Attributs du Produit
    private int refProduit;
    private String nomProduit;
    private String marque;
    private double prix;
    private int stock;

    // Getters et setters
    public int getRefProduit() {
    public void setRefProduit(int refProduit) {
    public String getNomProduit() {
    public void setNomProduit(String nomProduit) {
    public String getMarque() {
    public void setMarque(String marque) {
    public double getPrix() {
    public void setPrix(double prix) {
    public int getStock() {
    public void setStock(int stock) {

    // Constructeur du Produit
    public Produit(int refProduit, String nomProduit, String marque, double prix, int stock) {
        super();
        this.refProduit = refProduit;
        this.nomProduit = nomProduit;
        this.marque = marque;
        this.prix = prix;
        this.stock = stock;
    }

}
```

Le mot clé *"this"* en Java fait toujours référence à l'objet (instance de la classe) en cours d'utilisation.

Une classe doit toujours avoir un constructeur vide par défaut, qui ne prend aucun paramètre. Faites-le !

Déclaration des méthodes :

Avant d'être utilisée, une méthode doit être définie (son nom, ses arguments et les instructions qu'elle contient). La déclaration d'une méthode se fait selon la syntaxe suivante :

```
type_de_donnee Nom_De_La_Methode(type1 argument1, type2 argument2, ...) {
 liste d'instructions
}
```

La méthode peut renvoyer une valeur (et donc se terminer) grâce au mot-clé ***return***. La syntaxe de l'instruction return est simple :

```
return valeur_ou_variable;
```

Si la méthode ne renvoie aucune valeur, on la fait alors précéder du mot-clé ***void***.

Les arguments sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes.

Ajouter la méthode afficherStock() dans la classe *Produit*. Cette méthode permet d'afficher le nombre de certains produits dans le stock.

Produit.java

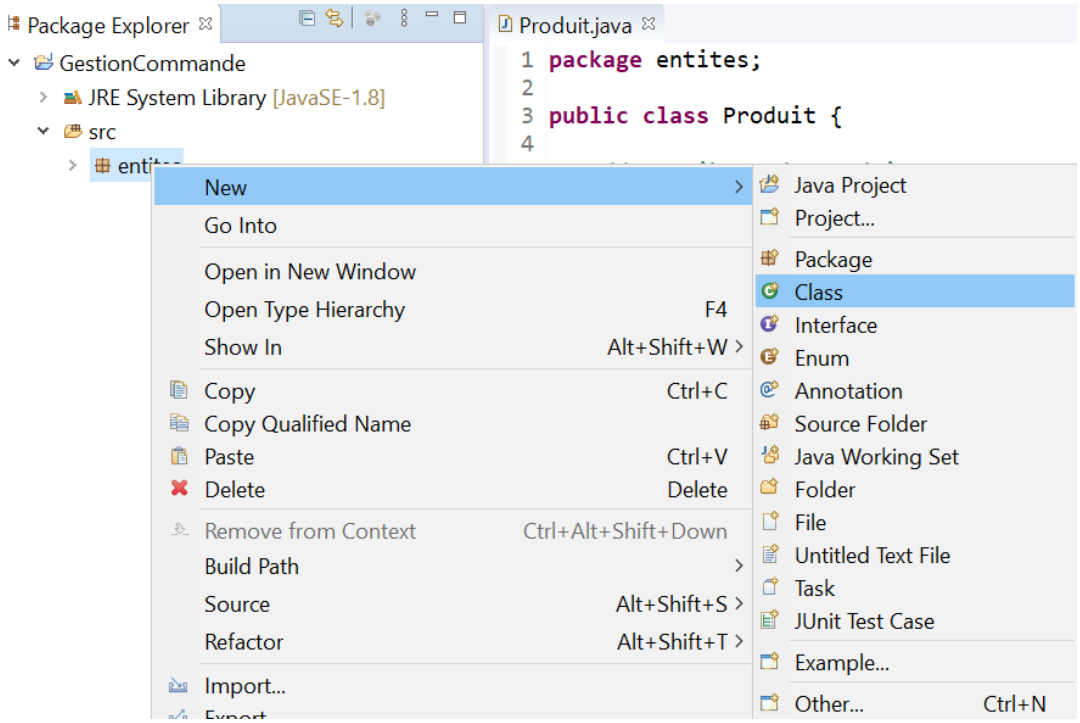
```java
package entites;

public class Produit {
    // Attributs du Produit
    private int refProduit;
    private String nomProduit;
    private String marque;
    private double prix;
    private int stock;

    // Getters et setters
    public int getRefProduit() {
    public void setRefProduit(int refProduit) {
    public String getNomProduit() {
    public void setNomProduit(String nomProduit) {
    public String getMarque() {
    public void setMarque(String marque) {
    public double getPrix() {
    public void setPrix(double prix) {
    public int getStock() {
    public void setStock(int stock) {

    // afficher le stock du produit
    public void afficherStock () {
        System.out.println("Le nombre de produits "+ nomProduit+ "ayant la référence "+refProduit+ " dans les stock est "+stock);
    }
}
```

La fonction System.out.print() est une fonction prédéfinie par le langage java. Elle permet d'afficher toujours la chaîne de caractères qui se trouve entre ses parenthèses.

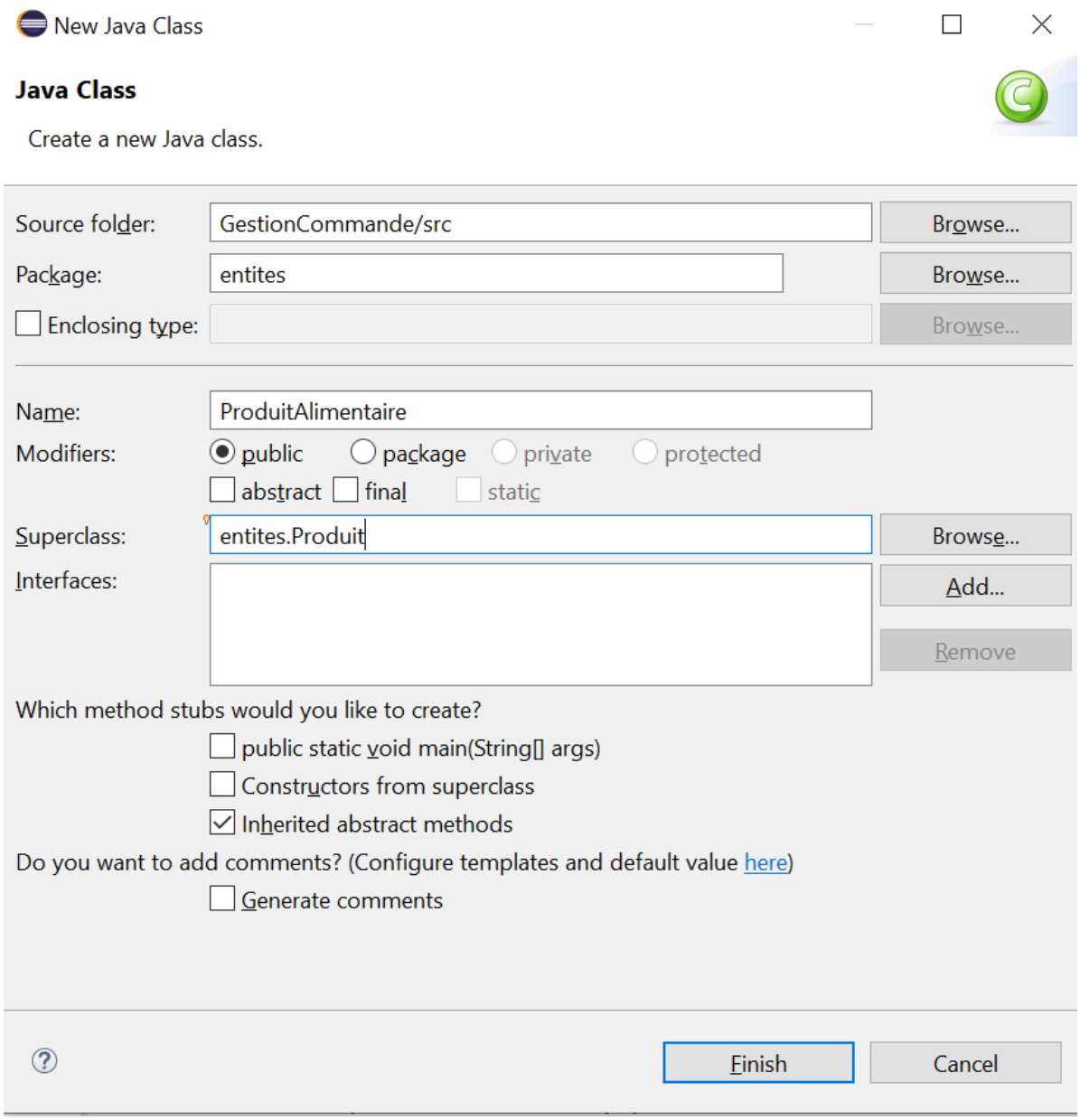
La notion d'héritage

Dans ce qui suit, vous allez mettre en oeuvre la notion d'héritage en Java.
Vous allez créer deux classes "ProduitAlimentaire" et "ProduitMénager" qui héritent de la classe *Produit*.

Pour ce faire, cliquez droit sur le package "entites", puis new/class.



Une fois le nom de la classe “ProduitAlimentaire” saisi, il faut mentionner le nom du package suivie par le nom de la classe mère dans le champs de saisie “superclass” (ici entites.Produit).

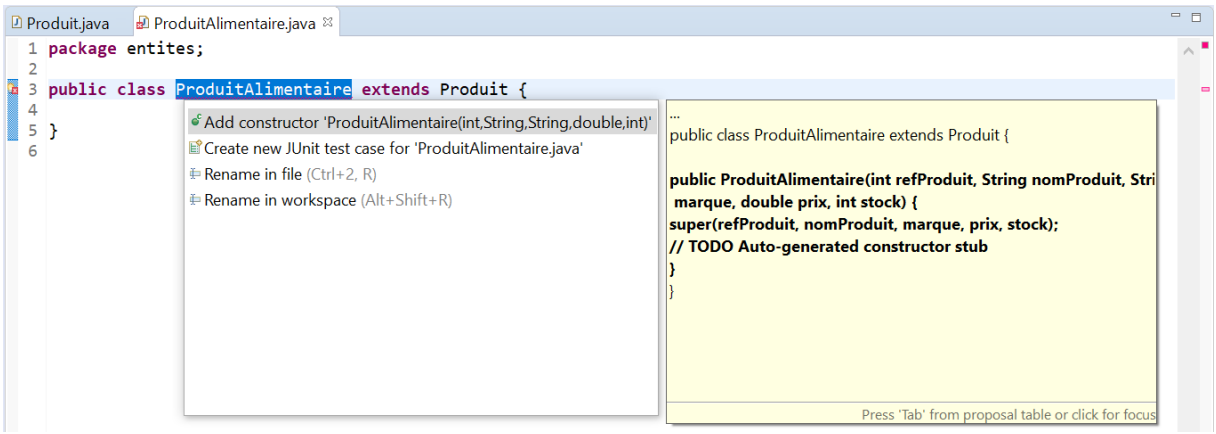


La classe “ProduitAlimentaire” est créée dans le package “entites”.
Le mot clé “extends” dans la déclaration de la classe indique l’héritage de la classe mère.

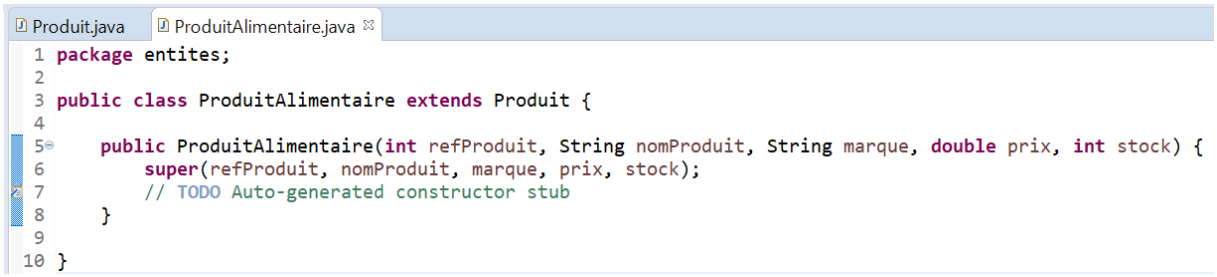
Produit.java ProduitAlimentaire.java

```
1 package entites;
2
3 public class ProduitAlimentaire extends Produit {
4
5 }
6
```

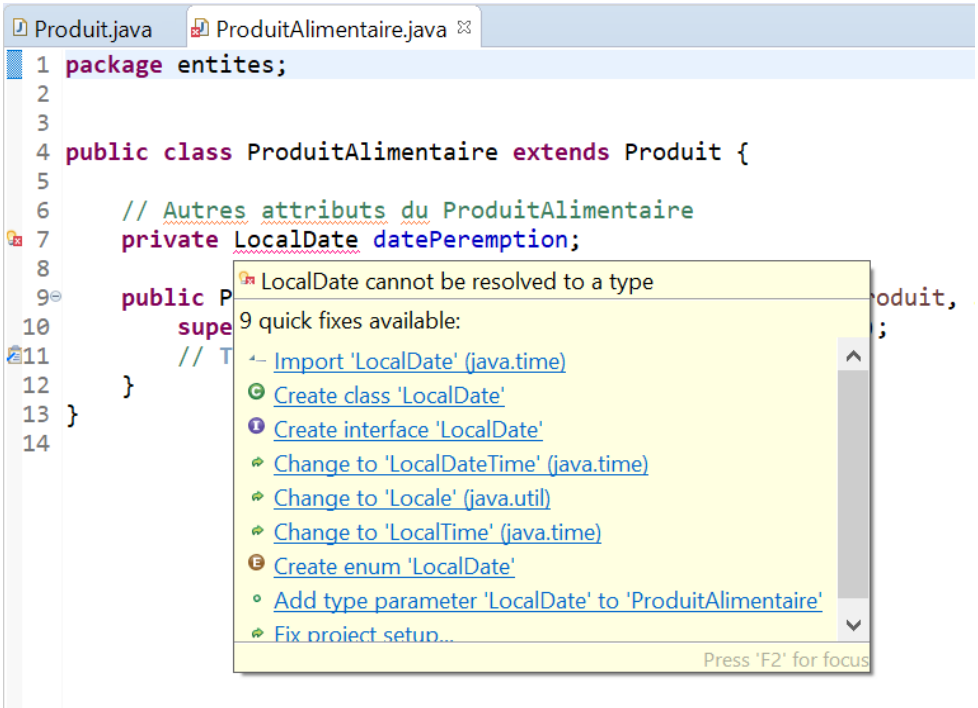
On peut remarquer que le nom de la classe est souligné en rouge. En effet, l’erreur est due à l’absence du constructeur dans la classe (parfois l’erreur ne s’affiche pas).
En positionnant le curseur sur l'élément souligné, plusieurs solutions sont proposées.



Choisissez la première solution (“Add constructor”) et tapez sur entrée.
Le constructeur est généré automatiquement de la classe mère.



Ajoutez l’attribut datePeremption de type LocalDate dans la classe fille ProduitAlimentaire.
Pour utiliser le type localDate, il faut importer le package java.time.LocalDate.



Générez le getter et le setter de l'attribut datePeremption ainsi que les constructeurs de la classe ProduitAlimentaire.

Produit.java ProduitAlimentaire.java

```java
package entites;

import java.time.LocalDate;

public class ProduitAlimentaire extends Produit {

    // Autres attributs du ProduitAlimentaire
    private LocalDate datePeremption;

    public ProduitAlimentaire(int refProduit, String nomProduit, String marque, double prix, int stock) {
        super(refProduit, nomProduit, marque, prix, stock);
        // TODO Auto-generated constructor stub
    }

    public ProduitAlimentaire(int refProduit, String nomProduit, String marque, double prix, int stock,
            LocalDate datePeremption) {
        super(refProduit, nomProduit, marque, prix, stock);
        this.datePeremption = datePeremption;
    }

    public LocalDate getDatePeremption() {
    public void setDatePeremption(LocalDate datePeremption) {
}
```

Ajoutez une méthode qui permet d'afficher un message lorsqu'un produit alimentaire est périmé.

La méthode compare la date actuelle à la date de péremption du produit et affiche un message selon ceci.

ProduitAlimentaire.java

```java
package entites;

import java.time.LocalDate;

public class ProduitAlimentaire extends Produit {

    // Autres attributs du ProduitAlimentaire
    private LocalDate datePeremption;

    public ProduitAlimentaire(int refProduit, String nomProduit, String marque, double prix, int stock) {

    public ProduitAlimentaire(int refProduit, String nomProduit, String marque, double prix, int stock,

    public LocalDate getDatePeremption() {

    public void setDatePeremption(LocalDate datePeremption) {

    // Méthode pour afficher un message sur la date de peremption
    public void afficherMessagePerime() {
        if (datePeremption.isBefore(LocalDate.now()))
        {
            System.out.println("le produit alimentaire " + nomProduit + "est périmé");
        } else {
            System.out.println("le produit alimentaire " + nomProduit + "est non périmé");
        }
    }
}
```

On remarque qu'il faut changer la visibilité de l'attribut *nomProduit* de "Private" à "Protected" dans la classe mère pour qu'il soit utilisé dans la méthode de la classe fille.

Lorsque vous changez les attributs de visibilité dans la classe *Produit* et vous sauvegardez, vous pouvez constater que l'erreur dans la classe *ProduitAlimentaire* a disparue.

*Produit.java ProduitAlimentaire.java

```java
package entites;

public class Produit {

    // Attributs du Produit
    protected int refProduit;
    protected String nomProduit;
    protected String marque;
    protected double prix;
    protected int stock;

    // Getters et setters
    public int getRefProduit() {
    public void setRefProduit(int refProduit) {
    public String getNomProduit() {
    public void setNomProduit(String nomProduit) {
    public String getMarque() {
    public void setMarque(String marque) {
    public double getPrix() {
    public void setPrix(double prix) {
    public int getStock() {
    public void setStock(int stock) {

    // Constructeur du Produit
    public Produit(int refProduit, String nomProduit, String marque, double prix, int stock) {
        super();
        this.refProduit = refProduit;
        this.nomProduit = nomProduit;
        this.marque = marque;
        this.prix = prix;
        this.stock = stock;
    }
```

À vous de jouer. Créez la classe *ProduitMenager* qui hérite de la classe *Produit* et qui a comme attribut "ecolabel" de type boolean. N'oubliez pas le constructeur, le getter et le setter.

Produit.java ProduitAlimentaire.java ProduitMenager.java

```java
package entites;

public class ProduitMenager extends Produit {


    private boolean ecolabel;

    public ProduitMenager(int refProduit, String nomProduit, String marque, double prix, int stock) {
        super(refProduit, nomProduit, marque, prix, stock);
        // TODO Auto-generated constructor stub
    }

    public boolean isEcolabel() {
        return ecolabel;
    }

    public void setEcolabel(boolean ecolabel) {
        this.ecolabel = ecolabel;
    }

}
```

À vous de jouer! Créez la classe *Client*.

- *idClient* et *nbrPointsFidelite* de type entier
- *nom*, *prenom*, *adresse*, et *tel* de type chaine de caractères

Générez le constructeur, les getter et setters.

Produit.java Client.java

```java
package entites;

public class Client {

    // Attributs du client
    private int idClient;
    private String nom;
    private String prenom;
    private String tel;
    private String adresse;
    private int nbrPointsFidelite;

    // Constructeur du client
    public Client(int idClient, String nom, String prenom, String tel, String adresse, int nbrPointsFidelite) {
        super();
        this.idClient = idClient;
        this.nom = nom;
        this.prenom = prenom;
        this.tel = tel;
        this.adresse = adresse;
        this.nbrPointsFidelite = nbrPointsFidelite;
    }

    // Getters et setters
    public int getIdClient() {
        return idClient;
    }

    public void setIdClient(int idClient) {
        this.idClient = idClient;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public String getTel() {
        return tel;
    }

    public void setTel(String tel) {
        this.tel = tel;
    }

    public String getAdresse() {
        return adresse;
    }

    public void setAdresse(String adresse) {
        this.adresse = adresse;
    }

    public int getNbrPointsFidelite() {
        return nbrPointsFidelite;
    }

    public void setNbrPointsFidelite(int nbrPointsFidelite) {
        this.nbrPointsFidelite = nbrPointsFidelite;
    }

}

```

Redéfinition :

La redéfinition de méthode se produit dans deux classes ayant une relation d'héritage.
La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques).
Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge (on parlera de ça plus tard).

Dans notre exemple, ajoutez la méthode afficherCaractéristiquesProduit() dans la classe *Produit* qui permet d'afficher les caractéristiques d'un produit telles que son nom, sa référence, sa marque, le stock et son prix.

Utilisez "\n" pour assurer le retour à ligne.

Produit.java

```
22+    public void setNomProduit(String nomProduit) {
25+    public String getMarque() {
28+    public void setMarque(String marque) {
31+    public double getPrix() {
34+    public void setPrix(double prix) {
37+    public int getStock() {
40+    public void setStock(int stock) {
43
44     // Constructeur du Produit
45     public Produit(int refProduit, String nomProduit, String marque, double prix, int stock) {
46         super();
47         this.refProduit = refProduit;
48         this.nomProduit = nomProduit;
49         this.marque = marque;
50         this.prix = prix;
51         this.stock = stock;
52     }
53
54     // afficher les caractéristiques d'un produit
55     public void afficherCaracteristiques() {
56         System.out.println("Les caractéristiques du produit ayant la référence " + refProduit + " sont : \n nom :"
57                 + nomProduit + "\n prix :" + prix + "\n stock: " + stock);
58     }
59 }
```

La redéfinition de cette méthode dans les deux classes *ProduitAlimentaire* et *ProduitMenager* exprime le fait que les caractéristiques des produits sont différentes dans les deux classes filles. Dans la classe *ProduitAlimentaire,* la méthode afficherCaractéristiquesProduit () est redéfinit pour afficher aussi la date de péremption. Alors que dans la classe *ProduitMenager*, elle affiche si le produit est écolabel ou non.

Produit.java | ProduitAlimentaire.java

```
1  package entites;
2
3  import java.time.LocalDate;
4
5  public class ProduitAlimentaire extends Produit {
6
7      // Autres attributs du ProduitAlimentaire
8      private Date datePeremption;
9
10+    public ProduitAlimentaire(int refProduit, String nomProduit, String marque, double prix, int stock) {
14
15+    public ProduitAlimentaire(int refProduit, String nomProduit, String marque, double prix, int stock,
20
21+    public LocalDate getDatePeremption() {
24
25+    public void setDatePeremption(LocalDate datePeremption) {
28
29     // Méthode pour afficher un message sur la date de peremption
30+    public void afficherMessagePerime() {
33
34     // afficher les caractéristiques d'un produit alimentaire
35     public void afficherCaracteristiques() {
36         System.out.println("Les caractéristiques du produit ayant la référence " + refProduit + " sont : \n nom :"
37                 + nomProduit + "\n prix :" + prix + "\n stock : " + stock + "\n date de peremption " + datePeremption);
38     }
39
40 }
```

Produit.java | ProduitAlimentaire.java | ProduitMenager.java

```java
package entites;

public class ProduitMenager extends Produit {

    private boolean ecolabel;

    public ProduitMenager(int refProduit, String nomProduit, String marque, double prix, int stock) {

    public ProduitMenager(int refProduit, String nomProduit, String marque, int stock) {

    public ProduitMenager(int refProduit, String nomProduit, String marque, double prix, int stock, boolean ecolabel) {

    public boolean isEcolabel() {

    public void setEcolabel(boolean ecolabel) {
    // afficher les caractéristiques d'un produit menager
    public void afficherCaracteristiques() {
        System.out.println("Les caractéristiques du produit ayant la référence " + refProduit + " sont : \n nom :"
                + nomProduit + "\n prix :" + prix + "\n stock : " + stock + "\n ecolabel " + ecolabel);
    }
}
```

Création de la classe ligne commande

Une ligne de commande est composée d'un seul produit.
Créez la classe *LigneCommande* qui a comme attribut :

- *quantite* de type entier
- *produit* de type Produit

Générez le constructeur, les getter et setters.

LigneCommande.java | Client.java

```java
package entites;

public class LigneCommande {
    private Produit produit;
    private int quantite;

    // Constructeur
    public LigneCommande(Produit produit, int quantite) {
        super();
        this.produit = produit;
        this.quantite = quantite;
    }

    public Produit getProduit() {
        return produit;
    }

    public void setProduit(Produit produit) {
        this.produit = produit;
    }

    public int getQuantite() {
        return quantite;
    }

    public void setQuantite(int quantite) {
        this.quantite = quantite;
    }

}
```

Une commande est composée d'une ou plusieurs lignes de commande.

Créez la classe *Commande* qui a comme attribut :

- *numCommande* de type entier
- *dateCommande* de type Date
- *client* de type Client
- *total* de type double
- *listeLignesCommande* de type List

Générez le constructeur, les getter et setters.

Ajouter la méthode *calculerTotal()* dans la classe *Commande*. Cette méthode permet de calculer le prix total d'une commande.

Utilisez une boucle *For* pour parcourir la liste des lignes de commande et récupérer le prix de chacune.

Commande.java

```java
package entites;

import java.util.ArrayList;

public class Commande {

    // Attributs
    private int numCommande;
    private Date dateCommande;
    private Client client;
    private List<LigneCommande> listeLignesCommande = new ArrayList<LigneCommande>();
    private double total;

    // Constructeur
    public Commande(int numCommande, Date dateCommande, Client client, List<LigneCommande> listeLignesCommande) {
        super();
        this.numCommande = numCommande;
        this.dateCommande = dateCommande;
        this.client = client;
        this.listeLignesCommande = listeLignesCommande;
    }

    // Getters et Setters
    public int getNumCommande() {
        return numCommande;
    }

    public void setNumCommande(int numCommande) {
        this.numCommande = numCommande;
    }

    public Date getDateCommande() {
        return dateCommande;
    }

    public void setDateCommande(Date dateCommande) {
        this.dateCommande = dateCommande;
    }

    public Client getClient() {
        return client;
    }

    public void setClient(Client client) {
        this.client = client;
    }

    public List<LigneCommande> getListeLignesCommande() {
        return listeLignesCommande;
    }

    public void setListeLignesCommande(List<LigneCommande> listeLignesCommande) {
        this.listeLignesCommande = listeLignesCommande;
    }

    // Methode pour calculer le total de la commande
    public double calculerTotal() {
        total = 0;
        for (LigneCommande ligneCommande : listeLignesCommande) {
            total = total + (ligneCommande.getProduit().getPrix() * ligneCommande.getQuantite());
        }
        total = total - (client.getNbrPointsFidelite()/30);
        return total;
    }
}
```

La surcharge :

La surcharge d'une méthode ou d'un constructeur permet de définir plusieurs fois une même méthode ou constructeur avec des arguments différents dans la même classe. Le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments.

La **surcharge** survient lorsque deux méthodes ou plus dans une même classe ont le même nom de méthode mais des paramètres différents. La **redéfinition** signifie avoir deux méthodes avec le même nom et les mêmes paramètres, l'une des méthodes est dans la classe parente et l'autre dans la classe fille.

Comme pour n'importe quelle méthode, il est possible de surcharger les constructeurs, c'est-à-dire définir plusieurs constructeurs avec des signatures différentes (des paramètres différents). Ainsi, il sera possible d'initialiser différemment un même objet.

Exemple 1 : Surcharge la méthode *calculerTotal()*

Créez la méthode CalculerTotal() dans la classe *Commande* qui permet de calculer le prix total de la commande en effectuant une remise par rapport aux points de fidélité du client (par exemple, pour 150 points de fidélité, le client aura une réduction de 5 euros).

Surchargez la méthode CalculerTotal() afin d'ajouter une remise exceptionnelle à un client.

Commande.java

```
53
54      public void setListeLignesCommande(List<LigneCommande> listeLignesCommande) {
55          this.listeLignesCommande = listeLignesCommande;
56      }
57
58      // Methode pour calculer le total de la commande
59      public double calculerTotal() {
60          total = 0;
61          for (LigneCommande ligneCommande : listeLignesCommande) {
62              total = total + (ligneCommande.getProduit().getPrix() * ligneCommande.getQuantite());
63          }
64          total = total - (client.getNbrPointsFidelite() / 30);
65          return total;
66      }
67
68      // Surchager la methode calculer le total de la commande
69      public double calculerTotal(int remiseExceptionnelle) {
70          total = 0;
71          double totalSansRemise = calculerTotal();
72          total = totalSansRemise - (totalSansRemise * remiseExceptionnelle / 100);
73          return total;
74      }
75  }
```

Exemple 2 : Surcharge du constructeur de la classe *ProduitMenager*

Surchargez le constructeur de la classe *ProduitMenager* en fixant le prix du produit ménager à 0 (l'attribut initialisé ne figure pas dans les paramètres du constructeur) et de nouveau en initialisant l'attribut "ecolabel".

Produit.java | ProduitAlimentaire.java | ProduitMenager.java

```
1  package entites;
2
3  public class ProduitMenager extends Produit {
4
5      private boolean ecolabel;
6
7      public ProduitMenager(int refProduit, String nomProduit, String marque, double prix, int stock) {
8
9          super(refProduit, nomProduit, marque, prix, stock);
10     }
11
12     public ProduitMenager(int refProduit, String nomProduit, String marque, int stock) {
13
14         super(refProduit, nomProduit, marque, 0, stock);
15     }
16
17     public ProduitMenager(int refProduit, String nomProduit, String marque, double prix, int stock, boolean ecolabel) {
18
19         super(refProduit, nomProduit, marque, 0, stock);
20         this.ecolabel = ecolabel;
21     }
```

Le programme principal

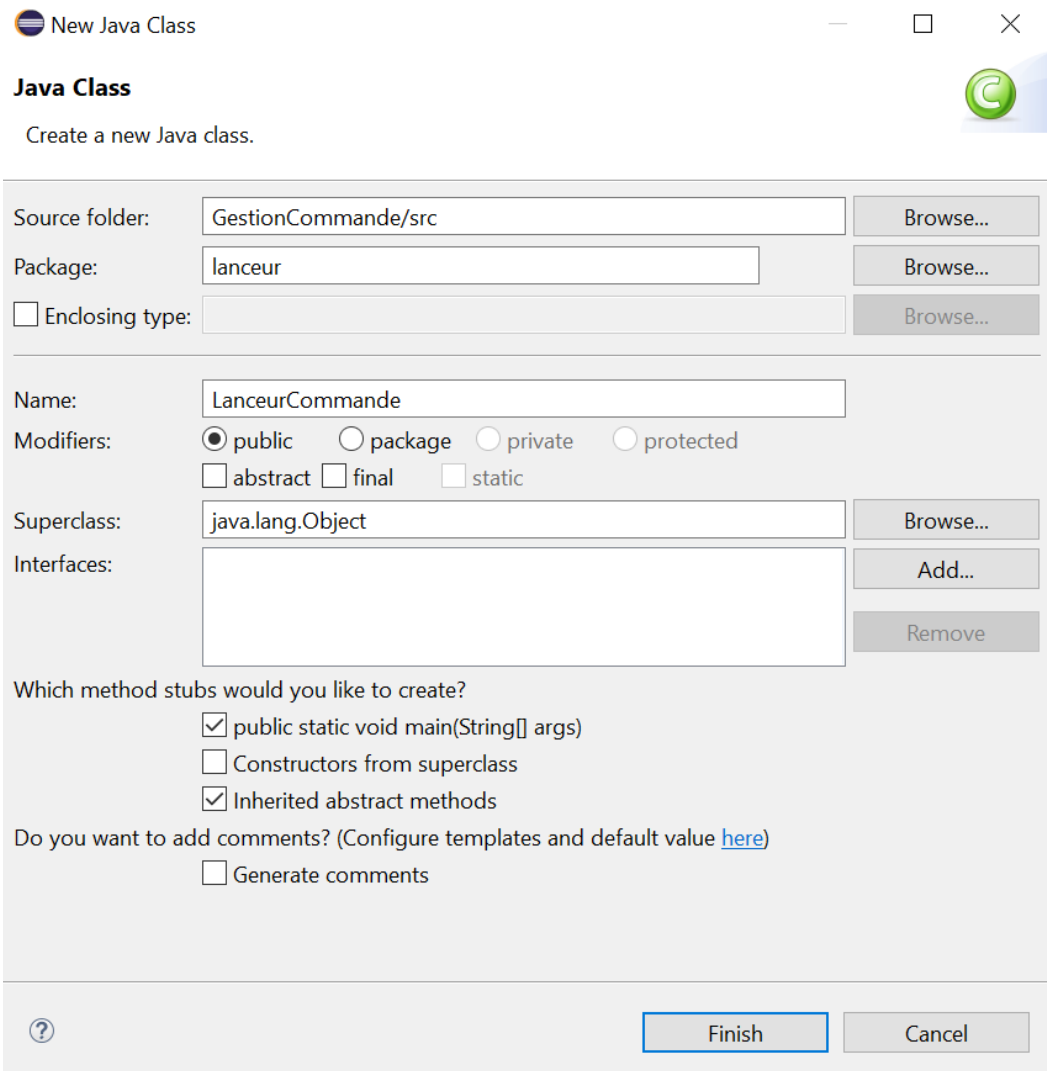
Créez le deuxième package *lanceur*.

Un programme doit avoir au moins une classe contenant une définition de méthode principale **main**. L'exécution du programme commence par cette méthode. C'est elle qui invoquera éventuellement les autres fonctions du programme.

Concrètement, une classe dite *"Main"* possède la méthode statique suivante :

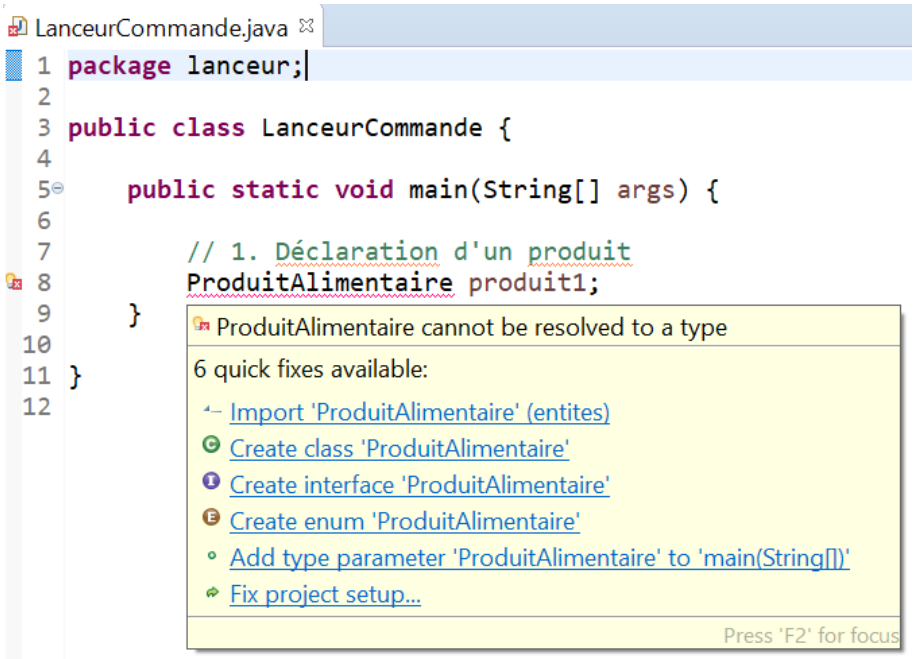
public static void main(String[] args) {}

Pour ce faire, créez une classe principale *LanceurCommande*; une fois le nom de la classe saisi, cochez la case "*public static void main(Strings [] args)*" pour qu'elle soit déclarée dans la classe.



Déclarez *produit1* de type *ProduitAlimentaire.*

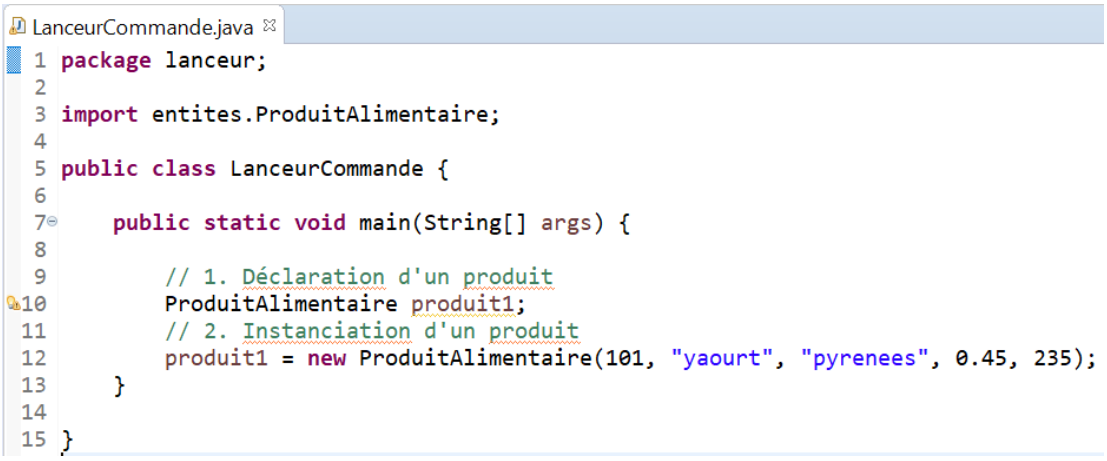
Eclipse signale qu'il y a une erreur. En passant la souris au-dessus de la croix rouge indiquant l'erreur, vous voyez les détails de l'erreur. Il vous propose quelques solutions pour fixer l'erreur.



Choisissez la première proposition *"import 'ProduitAlimentaire' (entites)"* pour importer le package qui comporte la classe *ProduitAlimentaire*.

Eclipse génère le code qui permet d'importer le package *entites*.

Instanciez *produit1* avec des valeurs pour refProduit, nomProduit, marque, prix, et stock.
Pour ce faire il faut d'abord déclarer l'objet *produit1* de type *ProduitAlimentaire* (Ligne 10) et ensuite faire appel au constructeur *ProduitAlimentaire* en mettant dedans les valeurs des paramètres pour initialiser l'objet *produit1* (ligne 12).



Déclarez et instanciez *produit2*, *produit3*, et *produit4*. En effet, la déclaration et l'instanciation d'un objet peuvent se faire en même temps (sur la même ligne, c.à.d. déclaration et initialisation) comme indiqué dans la figure ci-dessous.

LanceurCommande.java

```java
package lanceur;

import java.time.LocalDate;
import entites.ProduitAlimentaire;
import entites.ProduitMenager;

public class LanceurCommande {

    public static void main(String[] args) {

        // 1. Déclaration d'un produit
        ProduitAlimentaire produit1;
        // 2. Instanciation d'un produit
        produit1 = new ProduitAlimentaire(101, "yaourt", "pyrenees", 0.45, 235);
        // 3. Déclaration et instanciation d'autres produits
        ProduitAlimentaire produit2 = new ProduitAlimentaire(102, "haricot", "tarbais", 3.29, 120, LocalDate.of(2021, 04, 15));
        ProduitMenager produit3 = new ProduitMenager(103, "liquideVaisselle", "OMO", 8.89, 70, true);
        ProduitMenager produit4 = new ProduitMenager(104, "liquideLessive", "OMO", 13.45, 95, true);
    }
}
```

Déclarez et instanciez *client1*.

LanceurCommande.java

```java
package lanceur;

import java.time.LocalDate;

import entites.Client;
import entites.ProduitAlimentaire;
import entites.ProduitMenager;

public class LanceurCommande {

    public static void main(String[] args) {

        // 1. Déclaration d'un produit
        ProduitAlimentaire produit1;
        // 2. Instanciation d'un produit
        produit1 = new ProduitAlimentaire(101, "yaourt", "pyrenees", 0.45, 235);
        // 3. Déclaration et instanciation d'autres produits
        ProduitAlimentaire produit2 = new ProduitAlimentaire(102, "haricot", "tarbais", 3.29, 120, LocalDate.of(2021, 04, 15));
        ProduitMenager produit3 = new ProduitMenager(103, "liquideVaisselle", "OMO", 8.89, 70, true);
        ProduitMenager produit4 = new ProduitMenager(104, "liquideLessive", "OMO", 13.45, 95, true);

        // 4. Déclaration et instanciation d'un client
        Client client1 = new Client(1, "DUPOND", "Jean", "0607060706", "47 Avenue Azereix", 10);

    }
}
```

Déclarez et instanciez trois lignes de commandes *ligne1*, *ligne2, et ligne3.*

Déclarez *listeCommande1* de type liste.

Pour manipuler les listes, il faut importer le package *ArrayList* et le package *Collection*.
Ajoutez *ligne1*, *ligne2, et ligne3* dans la liste *listeCommande.*

Déclarez et instanciez la commande *commande1*.

Appelez la méthode *calculerTotal()* pour calculer le montant total de *commande1* et afficher le résultat comme montré dans la figure ci-dessous.

Pour exécuter une méthode, il suffit de faire appel à elle en écrivant son nom suivie d'une parenthèse ouverte puis d'une parenthèse fermée :

Nom_De_La_méthode(); // si la méthode est static
***objet**.Nom_De_La_méthode() ;*

Comme le cas *commande1.calculerTotal(); // Ligne 47*

Si jamais vous avez défini des arguments dans la déclaration de la méthode, il faudra veiller à les inclure lors de l'appel de la méthode (le même nombre d'arguments séparés par des virgules)

objet.Nom_De_La_méthode(argument1, argument2);

LanceurCommande.java

```
26          // 4. Déclaration et instanciation d'un client
27          Client client1 = new Client(1, "DUPOND", "Jean", "0607060706", "47 Avenue Azereix", 10);
28
29          // 5. Déclaration et instanciation des lignes de commande
30          LigneCommande ligne1 = new LigneCommande(produit1, 8);
31          LigneCommande ligne2 = new LigneCommande(produit2, 3);
32          LigneCommande ligne3 = new LigneCommande(produit3, 2);
33
34          // 6. Déclaration d'une liste de lignes de commande
35          List<LigneCommande> listeLignesCommande1 = new ArrayList<LigneCommande>();
36
37          // 7. Ajout des lignes de commande dana la liste
38          listeLignesCommande1.add(ligne1);
39          listeLignesCommande1.add(ligne2);
40          listeLignesCommande1.add(ligne3);
41
42          // 8. Déclaration et instanciation d'une commande
43          Date aujourdhui = new Date();
44          Commande commande1 = new Commande(1, aujourdhui, client1, listeLignesCommande1);
45
46          // 9. Calcul et affichage du total de la commande
47          double total = commande1.calculerTotal();
48          System.out.println("le montant total de la commande est de : " + total + " euros");
49      }
50  }
51
```

Problems Javadoc Declaration Console

```
<terminated> LanceurCommande [Java Application] C:\Program Files\Java\jre1.8.0_231\bin\javaw.exe  (18 sept. 2020 à 17:23:50 – 17:23:50)
le montant total de la commande est de : 31.25 euros
```

Avant de passer à l'étape suivante, il faut mettre toute la partie 9 (lignes 48 et 49) en commentaires en utilisant les //.

Dans ce qui suit, nous allons créer un menu principal avec une liste à choix multiple : gestion des produits, gestion des clients, ou gestion des commandes. L'utilisateur est invité à choisir une des trois fonctionnalités.

Pour que Java puisse lire ce que vous tapez au clavier, il faut utiliser l'objet **Scanner**. Lorsque vous faites *System.out.println()*, vous appliquez la méthode *println()* sur la sortie standard; ici, nous allons utiliser l'entrée standard *System.in*.

Pour commencer, instanciez un objet *Scanner*.

Scanner sc = new Scanner(System.in);

Eclipse vous signale une erreur et souligne le mot scanner en rouge.
Cliquez sur la croix rouge et faites un double-clic sur Import java.util.Scanner. Et là, l'erreur disparaît !
L'affichage du menu principal est comme suit (Figure suivante) :

Saisissez le numéro de l'action souhaitée :

1. Gestion des produits,
2. Gestion des clients
3. Gestion des commandes

Voici l'instruction qui permet à Java de récupérer un entier que vous avez saisi :

int choix = sc.nextInt();

En effet, il y a une méthode de récupération de données pour chaque type :

- Pour lire un int, vous devez utiliser nextInt()
- Pour lire une chaîne de caractères, vous devez utiliser nextLine()
- Pour lire un double, vous devez utiliser nextDouble()
- Pour lire un long, vous devez utiliser nextLong()
- Pour lire un byte, vous devez utiliser nextByte()

Une fois l'application lancée, le message que vous avez écrit s'affiche dans la console.
Pensez à cliquer dans la console afin de saisir votre choix.

```
47        // 9. Calcul et affichage du total de la commande
48        // double total = commande1.calculerTotal();
49        // System.out.println("le montant total de la commande est de : " + total + " euros");
50
51        // 10. Création menu principal
52        Scanner sc = new Scanner(System.in);
53        System.out.println(" Saisissez le numéro de l'action souhaitée :\n 1. Gestion des produits. \n "
54                + "2. Gestion des clients. \n 3. Gestion des commandes");
55        int choix = sc.nextInt();
56        System.out.println("Vous avez choisi l'action " + choix);
57    }
58 }
59
```

Problems Javadoc Declaration Console

```
LanceurCommande [Java Application] C:\Program Files\Java\jre1.8.0_231\bin\javaw.exe  (18 sept. 2020 à 16:35:36)
Saisissez le numéro de l'action souhaitée :
1. Gestion des produits.
2. Gestion des clients.
3. Gestion des commandes
```

Pour choisir une des trois fonctionnalités, l'utilisateur est invité à saisir un entier (1, 2 ou 3). Si l'utilisateur se trompe et saisie un caractère et non pas un entier comme demandé par exemple, une erreur sera signalée.
En Java, les erreurs d'exécution sont appelées des exceptions et le traitement des erreurs est appelé gestion des exceptions.
Le mécanisme de gestion des erreurs se compose de deux mots clés qui permettent de détecter et de traiter ces erreurs (try et catch).
Si un événement indésirable survient dans le bloc try, la partie éventuellement non exécutée de ce bloc est abandonnée et le bloc catch est traité (Figure suivante).

```
        // 10. Création menu principal
        Scanner sc = new Scanner(System.in);
        System.out.println(" Saisissez le numéro de l'action souhaitée :\n 1. Gestion des produits. \n "
                + "2. Gestion des clients. \n 3. Gestion des commandes");
        try {
            int choix = sc.nextInt();
            System.out.println("Vous avez choisi l'action " + choix);

        } catch (Exception e) {
            System.out.println("Il faut saisir un entier!");
        }
    }
}
```

Problems Javadoc Declaration Console

```
<terminated> LanceurCommande [Java Application] C:\Program Files\Java\jre1.8.0_231\bin\javaw.exe  (18 sept. 2020 à 16:44:06 – 16:44:14)
 Saisissez le numéro de l'action souhaitée :
 1. Gestion des produits.
 2. Gestion des clients.
 3. Gestion des commandes
R
Il faut saisir un entier!
```

Pour répondre au choix de l'utilisateur, nous allons utiliser l'instruction Switch.

L'instruction Switch est utile pour gérer beaucoup de if / else if / else. Elle a une syntaxe plus courte et est plus appropriée pour ce type de cas.

L'instruction switch permet d'exécuter du code selon l'évaluation de la valeur d'une expression. La syntaxe générale est de la forme :

```
switch (expression) {
        case constante1 :
            instr1;
            instr2;
            break;

        case constante2 :
            ...
            break;
        default :
            ...
    }
```

Dans notre exemple, nous allons implémenter le deuxième cas : Gestion des clients.
Pour ajouter un client, l'utilisateur est invité à saisir son identifiant, nom, prénom, téléphone et adresse.
Une fois la saisie est terminée, le système affiche "Client ajouté avec succès".

LanceurCommande.java

```
50
51          // 10. Création menu principal
52          Scanner sc = new Scanner(System.in);
53          System.out.println(" Saisissez le numéro de l'action souhaitée :\n 1. Gestion des produit
54                  + "2. Gestion des clients. \n 3. Gestion des commandes");
55          try {
56              int choix = sc.nextInt();
57              System.out.println("Vous avez choisi l'action " + choix);
58
59              // Switch Case
60              switch (choix) {
61              case 1:
62                  System.out.println("Cette action n'est pas encore implémentée");
63                  break;
64              case 2: {
65                  Scanner sc1 = new Scanner(System.in);
66                  System.out.println("Saisissez l'identifiant du client");
67                  int id = sc1.nextInt();
68                  Scanner sc2 = new Scanner(System.in);
69                  System.out.println("Saisissez le nom du client");
70                  String nom = sc2.nextLine();
71                  Scanner sc3 = new Scanner(System.in);
72                  System.out.println("Saisissez le prenom du client");
73                  String prenom = sc3.nextLine();
74                  Scanner sc4 = new Scanner(System.in);
75                  System.out.println("Saisissez le numéro de télephone");
76                  String tel = sc4.nextLine();
77                  Scanner sc5 = new Scanner(System.in);
78                  System.out.println("Saisissez l'adresse du client");
79                  String adresse = sc5.nextLine();
80                  Client client2 = new Client(id, nom, prenom, tel, adresse, 10);
81                  System.out.println("Client ajouté avec succès");
82              }
83                  break;
84              case 3:
85                  System.out.println("Cette action n'est pas encore implémentée");
86                  break;
87              default:
88                  break;
89              }
90          } catch (Exception e) {
91              System.out.println("Il faut saisir un entier!");
92          }
93      }
94  }
95
```

Problems Javadoc Declaration Console

```
<terminated> LanceurCommande [Java Application] C:\Program Files\Java\jre1.8.0_231\bin\javaw.exe  (18 sept. 2020 à 17:02:40 – 17:04:05)
 Saisissez le numéro de l'action souhaitée :
 1. Gestion des produits.
 2. Gestion des clients.
 3. Gestion des commandes
2
Vous avez choisi l'action 2
Saisissez l'identifiant du client
2
Saisissez le nom du client
Ronaldo
Saisissez le prenom du client
Cristiano
Saisissez le numéro de télephone
0511051105
Saisissez l'adresse du client
Juventus stadium turin
Client ajouté avec succès
```

Une fois le système affiche “Client ajouté avec succès” l’exécution s’arrête ce qui oblige à relancer l’exécution si on souhaite réaliser une autre action. Pour cela nous pouvons ajouter une boucle qui permet au programme de s’exécuter jusqu’à ce que l’utilisateur choisie l’option exit.

L'instruction ***DO … While*** crée une boucle qui exécute une instruction jusqu'à ce qu'une condition de test ne soit plus vérifiée. La condition est testée après que l'instruction a été exécutée, le bloc d'instructions défini dans la boucle est donc exécuté au moins une fois.
La syntaxe générale est de la forme : dans notre exemple

```
do
  instruction
while (condition);
```

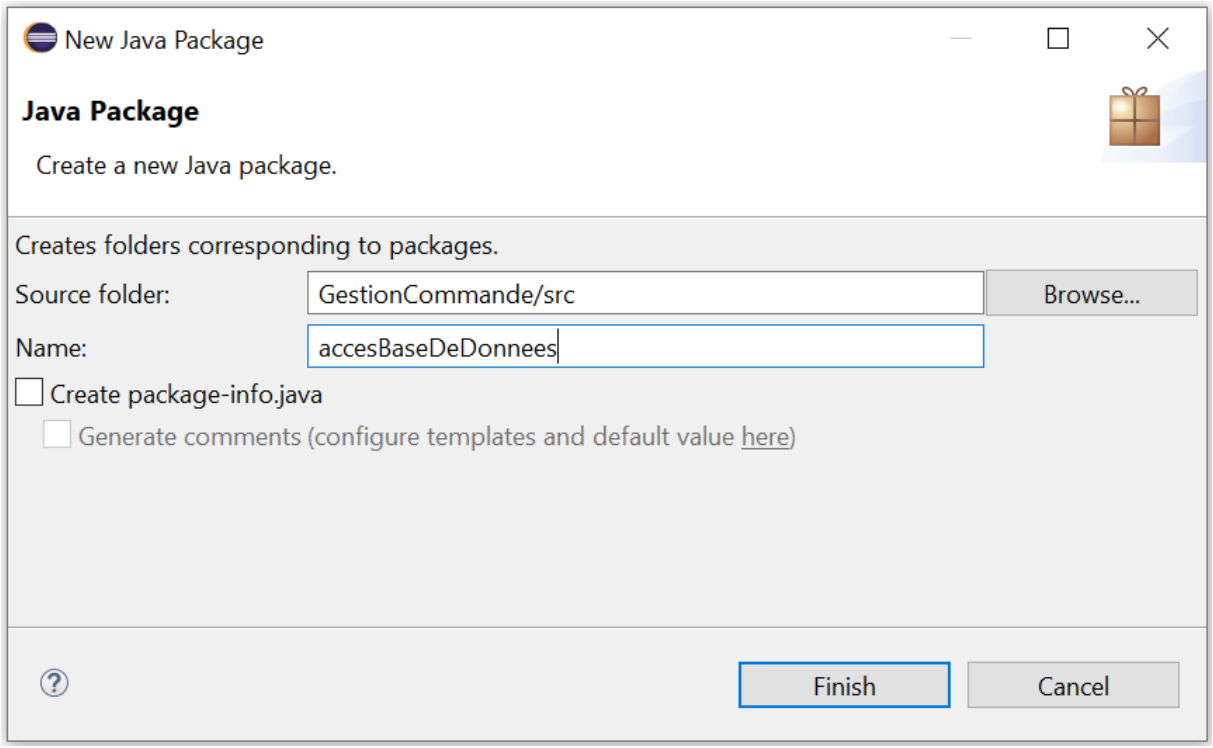
Dans notre exemple (ligne 57) nous ajoutons la boucle ***do*** et nous ajoutons l'option exit dans l'affichage (ligne 59). Nous ajoutons ensuite le case 4 dans ***Switch*** (ligne 94) et enfin la condition ***while*** (ligne 105).

```
55      //creation menu principal
56      int choix=0 ;
57      do{
58      Scanner sc=new Scanner(System.in);
59              System.out.println("Saisissez le numéro de l'action souhait0e : \n 1.Gestion des produits. \n 2. Gestion client. \n 3. Gestion des commandes. \n 4. exit");
60      try {
61
62      choix = sc.nextInt();
63
64      System.out.println("vous avez choisi l'action : " + choix);
65
66      //swicth case
67      switch (choix) {
68      case 1:
69          System.out.println("Cette action n'est pas encore impl0ment0e");
70          break;
71      case 2: {
72          Scanner sc1= new Scanner(System.in);
73          System.out.println("saisissez l'identifiant du client");
74          int id = sc1.nextInt();
75          Scanner sc2= new Scanner(System.in);
76          System.out.println("saisissez le nom du client");
77          String nom = sc2.nextLine();
78          Scanner sc3= new Scanner(System.in);
79          System.out.println("saisissez le prenom du client");
80          String prenom = sc3.nextLine();
81          Scanner sc4= new Scanner(System.in);
82          System.out.println("saisissez le num0ro de telephone");
83          String tel = sc4.nextLine();
84          Scanner sc5= new Scanner(System.in);
85          System.out.println("saisissez l'adresse du client");
86          String adresse = sc5.nextLine();
87          Client client2 = new Client(id, nom, prenom, tel, adresse, 10);
88          System.out.println("Client ajout0 avec succ0s");
89      }
90          break;
91      case 3:
92          System.out.println("Cette action n'est pas encore impl0ment0e");
93          break;
94      case 4:
95          System.out.println("Au revoir");
96          break;
97      default:
98          break;
99      }
100
101     }catch (Exception e) {
102         System.out.println("Il faut saisir un entier!");
103     }
104     }
105     while (choix!=4);
```

Accès base de données MySQL

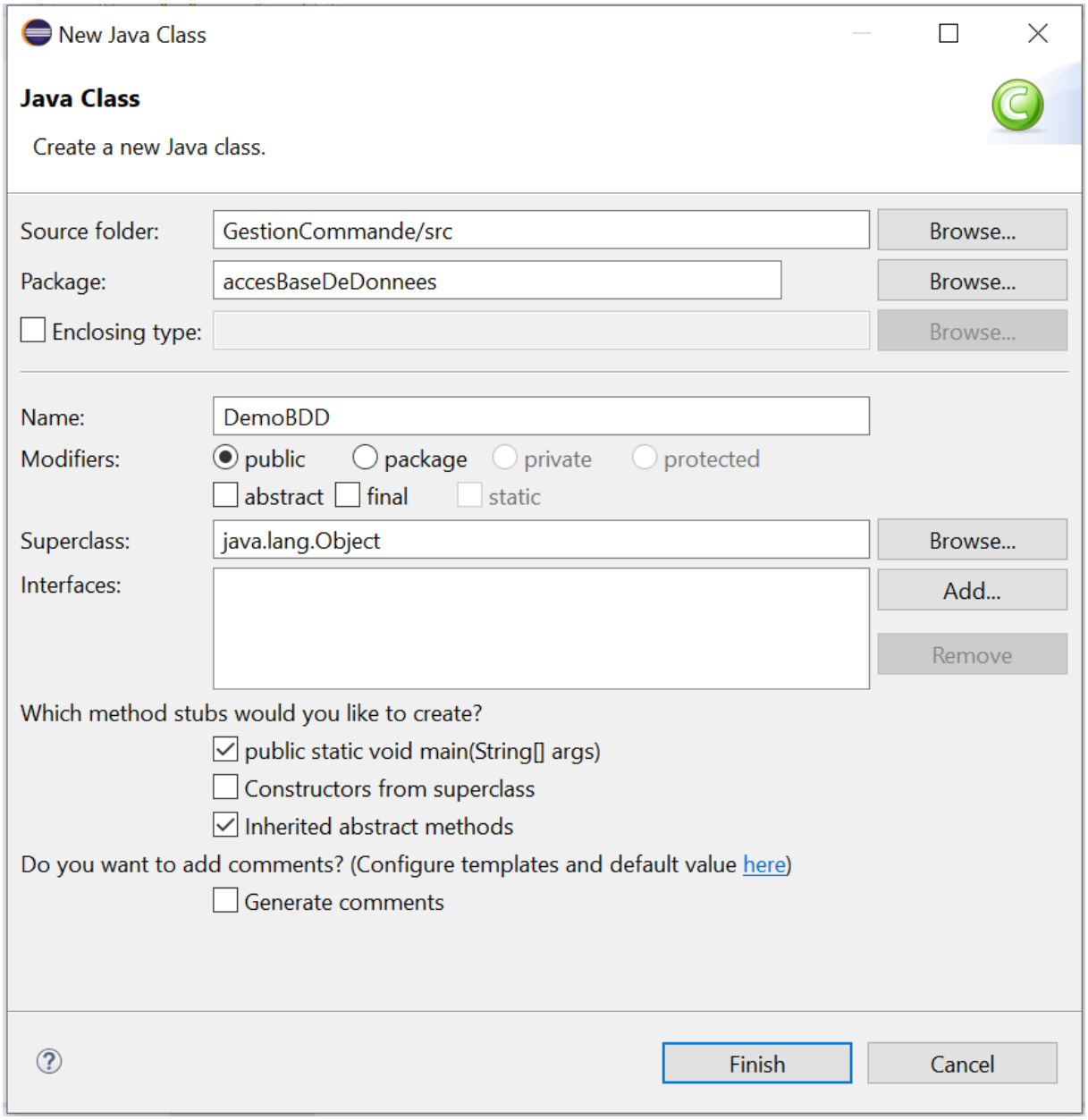
Dans cette partie, vous allez apprendre l'insertion et la récupération de données dans une base de données MySQL à travers un programme Java.

Créez un nouveau package *accesBaseDeDonnees*.



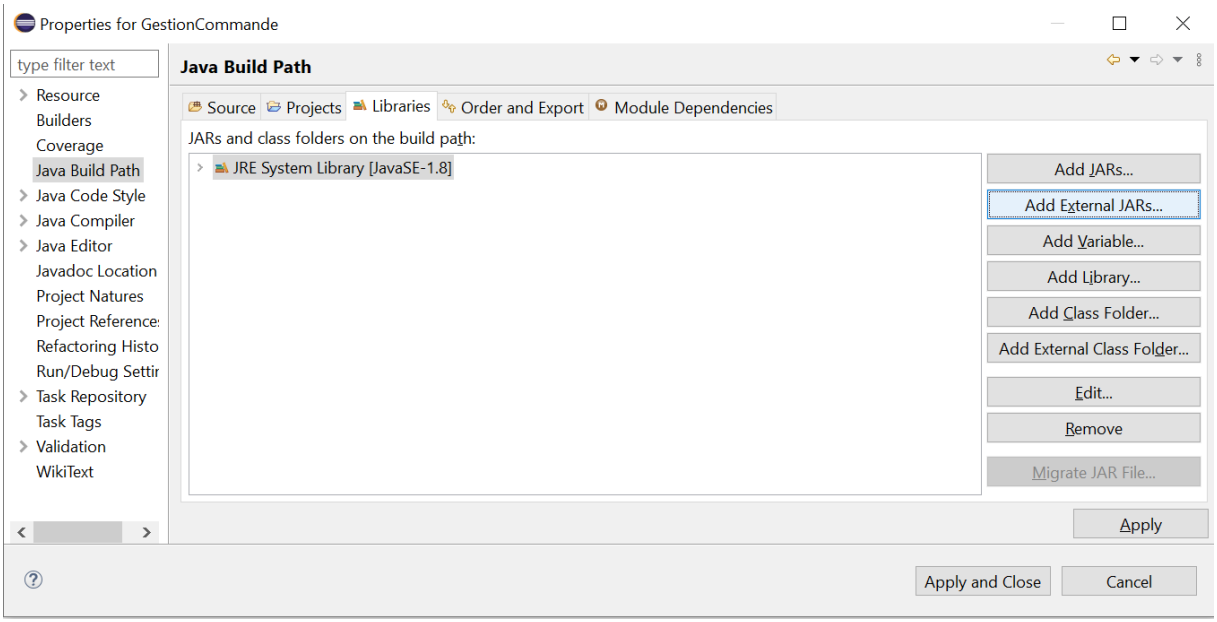
Créez ensuite la classe *DemoBDD*.

Cochez la case *"public static void main()"*.

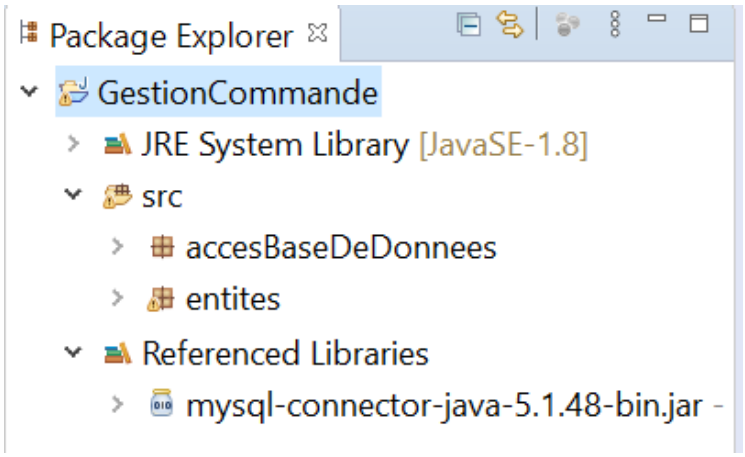


Ajout du driver JDBC

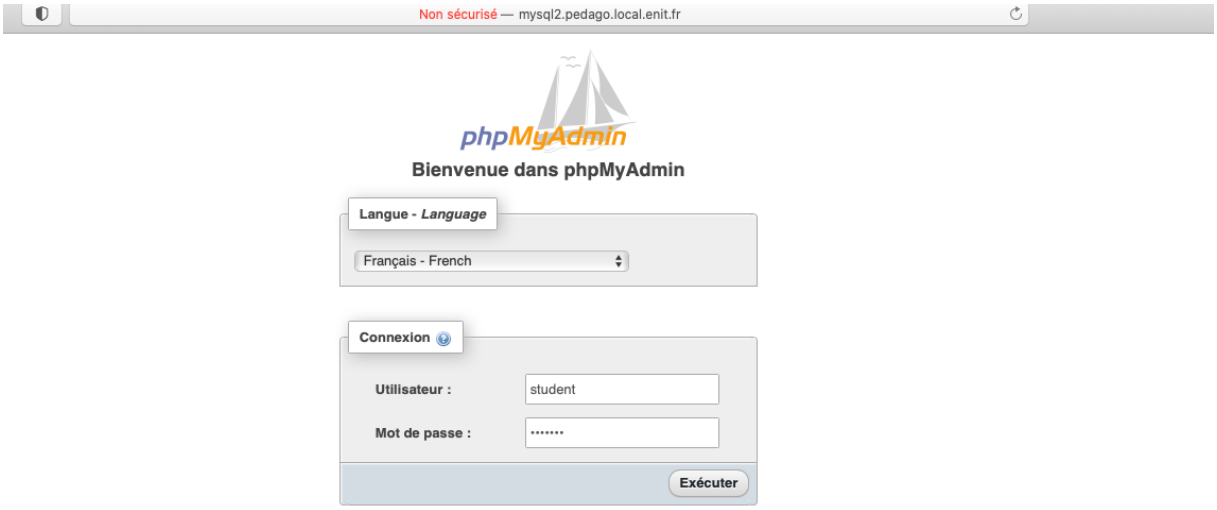
Un pilote JDBC est un composant logiciel permettant à une application Java d'interagir avec une base de données.
Il faut télécharger le driver «mysql-connector-java-5.1.48.jar » sur Moodle.
Il faut ensuite importer la librairie Driver JDBC dans CLASSPATH eclipse comme indiqué dans les deux figures ci-dessous. La librairie est disponible dans le répertoire partagé POO.
Cliquez-droit sur le projet -> properties -> Java Build Path. L'écran suivant apparaît :
Dans l'onglet "Librairies' cliquez sur bouton 'Add External Jars'. Sélectionner le driver à partir de l'endroit ou vous l'avez enregistrer après téléchargement puis OK.



Le driver apparaît désormais dans “Referenced librairies”.

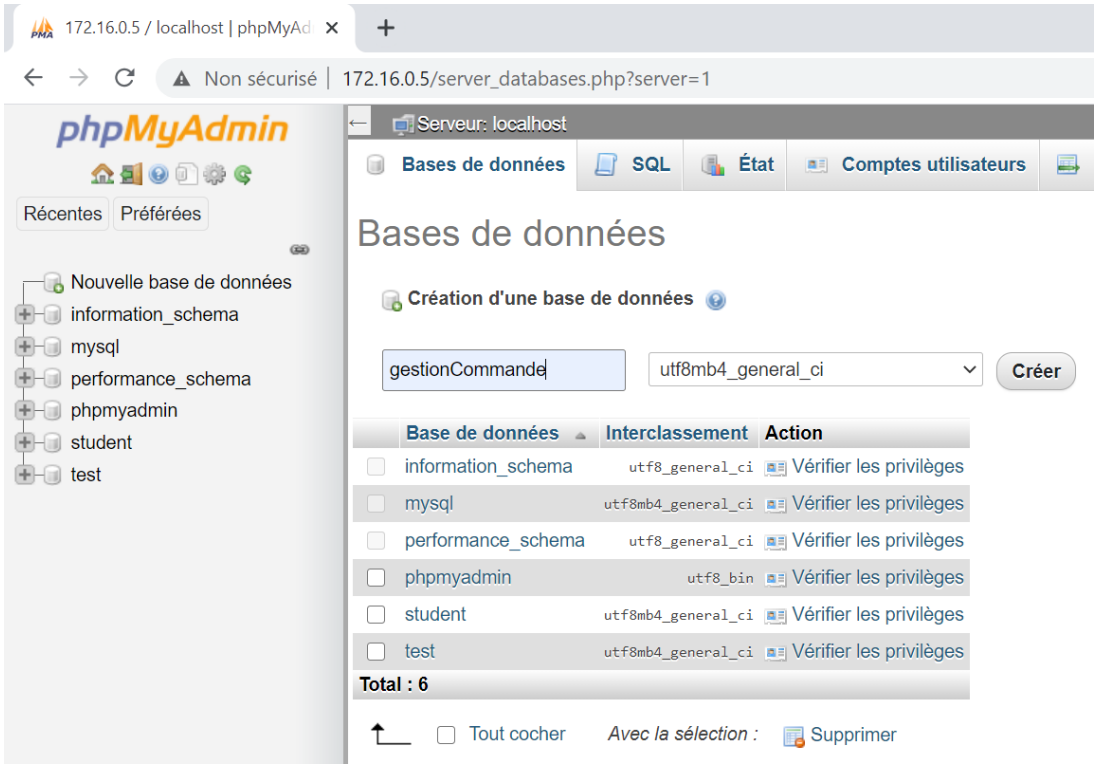


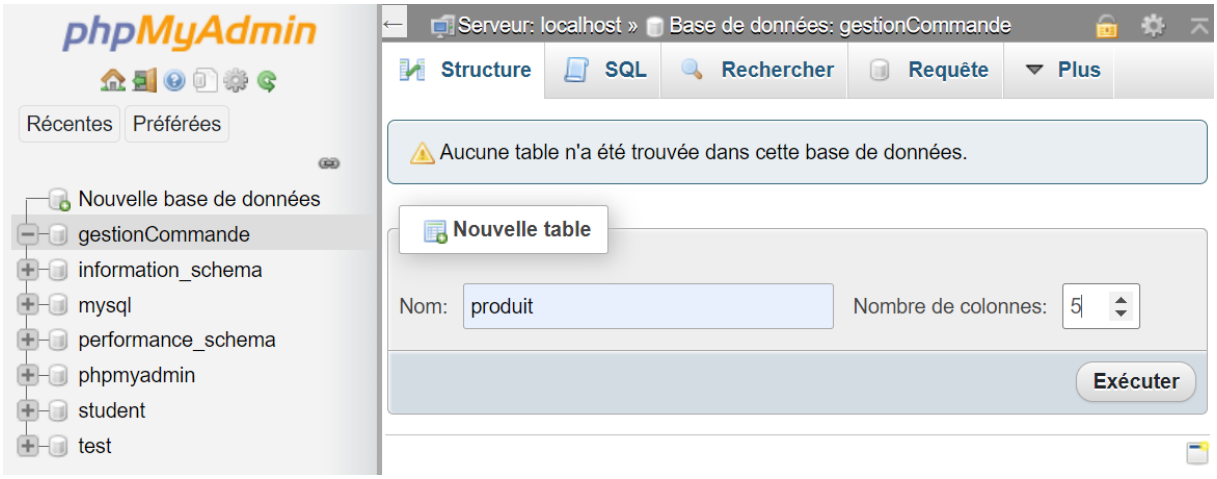
Pour créer une base de données il faut lancer le client phpMyAdmin. Il faut donc mettre le lien suivant dans votre navigateur : http://mysql2.pedago.local.enit.fr



Pour se connecter au client PhpMyAdmin, tapez “student” dans le champ correspondant à Utilisateur et “Enit@65” dans le champ correspondant au Mot de passe et exécuter. La fenêtre ci-dessous s’ouvre.

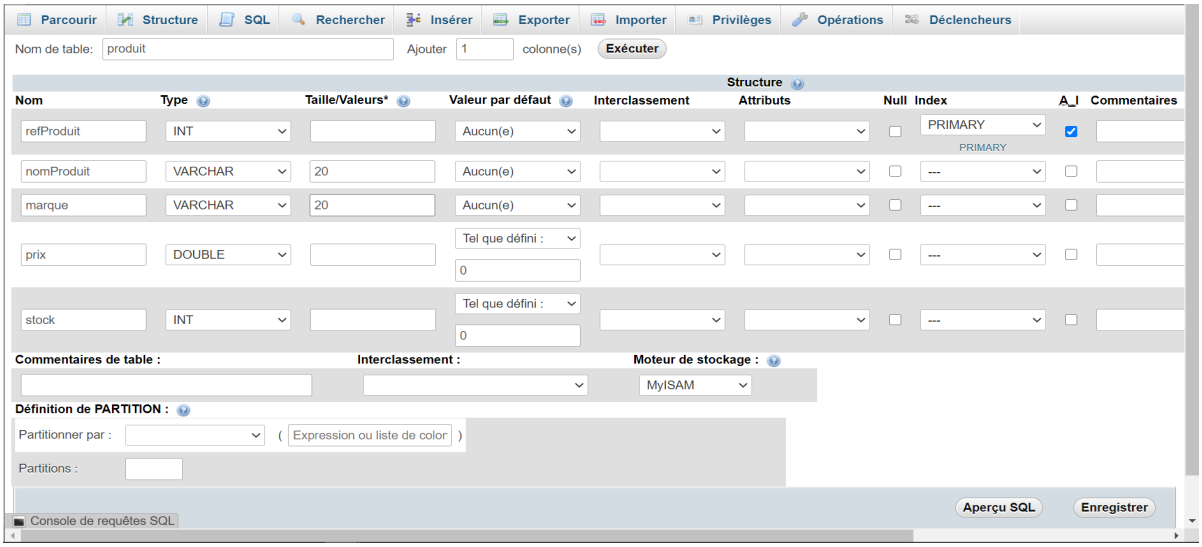
Vous allez trouver la base de données “gestionCommande” et la table “produit” possédant les 5 colonnes “refProduit”, “nomProduit”, “marque”, “prix”, et “stock”.
Les deux figures suivantes sont à titre inductifs pour vous montrer comment créer une BDD et une Table. Vous n’avez pas besoin de le refaire pour ce tutoriel.





Comme montré sur la figure suivante, on coche A_I (auto-increment) pour avoir l'auto-incrémentation automatique du référence produit. Nous n'avons plus besoin d'attribuer une référence produit.

Sur les attributs de type chaîne de caractères (VARCHAR), il faut saisir une longueur maximale (ici on initialise à 20 caractères).



La base de données est créé, maintenant on passe à la configuration de la classe DemoBDD pour connecter votre programme Java à la base de données.

*DemoBDD.java *test.java

```java
package accesBaseDeDonnees;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class DemoBDD {

    private static Connection cn = null;

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

        String url = "jdbc:mysql://mysql2.pedago.local.enit.fr/gestionCommande";
        String login = "student";
        String password = "Enit@65";
        try {
            //Etape 1 : chargement driver
            Class.forName("com.mysql.jdbc.Driver");

            //etape 2 : récupération de la connexion
            cn = DriverManager.getConnection(url, login, password);

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                cn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }


}
```

Après avoir connecté le code à la BDD, maintenant nous allons développer le code permettant d'utiliser les requêtes SQL. Pour cela nous allons voir 2 exemples pour insérer un produit à la BDD et ensuite extraire des informations de la BDD.

Pour manipuler des requêtes, nous utilisons l'interface ***PreparedStatement*** qui représente une instruction paramétrée. Cette interface est caractérisée par deux points principaux :

- Les instances de PreparedStatement contiennent une instruction SQL déjà compilée. D'où le terme prepared. Cela améliore notamment les performances si cette instruction doit être appelée de nombreuses fois ;

- Les instructions SQL des instances de PreparedStatement contiennent un ou plusieurs paramètres d'entrée, non spécifiés lors de la création de l'instruction. Ces paramètres sont représentés par des points d'interrogation (?). Ces paramètres doivent être spécifiés avant l'exécution.

*DemoBDD.java

```java
package accesBaseDeDonnees;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class DemoBDD {

    private static Connection cn = null;

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

        String url = "jdbc:mysql://mysql2.pedago.local.enit.fr/gestionCommande";
        String login = "student";
        String password = "Enit@65";
        try {
            //Etape 1 : chargement driver
            Class.forName("com.mysql.jdbc.Driver");

            //etape 2 : récupération de la connexion
            cn = DriverManager.getConnection(url, login, password);

        //ajouter un produit
            // creer requete insert
            String query_insert = " insert into produit (nomProduit, marque, prix, stock)"+ " values (?, ?, ?, ?)";

            //Etape 2 : creer le preparedstatement
            PreparedStatement preparedStmt_insert = cn.prepareStatement(query_insert);
            preparedStmt_insert.setString(1, "yaourt");
            preparedStmt_insert.setString(2, "pyrenees");
            preparedStmt_insert.setDouble(3, 0.45);
            preparedStmt_insert.setInt(4, 235);

            //Etape 3 : executer le preparedstatement
            preparedStmt_insert.execute();
            System.out.println("produit ajout@ avec succ@s");

        //selection des produits
            //etape 1 : creer la requete select
            String query_select = "select * from produit where prix>?";
            //Etape 2 creer le preparedstatement
            PreparedStatement preparedStmt_select = cn.prepareStatement(query_select);
            preparedStmt_select.setDouble(1,0.4);
            //etape 3 : exec preparedstatement
            ResultSet result = preparedStmt_select.executeQuery();

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                cn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }


}
```

PS : Si vous avez des erreurs, pensez à importer les bibliothèques nécessaires.

Le figure ci-dessous montre le deuxième exemple de requête qui interroge la BDD pour sélectionner/récupérer les produits dont le prix est supérieur à 0.4 euros (complétez votre code à partir de ligne 42).
L'instruction ***cn.close*** permet de fermer la session de connexion à la BDD afin de libérer les ressources de la mémoire.

DemoBDD.java

```
            preparedStmt_insert.setInt(3, 255);

            // Etape 3: executer le preparedstatement
            preparedStmt_insert.execute();
            System.out.println("produit ajouté avec succès");

        // Sélectionner les produits
            // Etape1: créer la requete select
            String query_select = "select * from produit where prix>?";
            // Etape 2: créer le preparedstatement
            PreparedStatement preparedStmt_select = cn.prepareStatement(query_select);
            preparedStmt_select.setDouble(1, 0.4);
            // Etape 3: executer le preparedstatement
            ResultSet result = preparedStmt_select.executeQuery();
            // Etape 4: parcour et afficher le résultat
            while (result.next()) {
                System.out.println(result.getString("nomProduit"));
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                cn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

Problems Javadoc Declaration Console

<terminated> DemoBDD [Java Application] C:\Program Files\Java\jre1.8.0_231\bin\javaw.exe (20 sept. 2020 à 20:02:13 – 20:02:14)

```
produit ajouté avec succès
yaourt
```

Création d'une classe spécifique pour la connexion à la base de données :

Pour des raisons de sécurité et d'optimisation de code il est préférable de se connecter à la base de données et de la fermer de façon indépendante de l'exécution d'une ou plusieurs requêtes. Il serait donc préférable de créer une classe de connexion à laquelle on fait appel à chaque fois que l'on veut se connecter à la base. Il faut donc créer la classe `ConnexionBDD` suivante dans le package *accesBaseDeDonnees*.

```
1  package accesBaseDeDonnees;
2  import java.sql.Connection;
3  import java.sql.DriverManager;
4  import java.sql.PreparedStatement;
5  import java.sql.ResultSet;
6  import java.sql.SQLException;
7
8  public class ConnexionBDD {
9
10     /* création d'une connection statique pour l'utiliser partout sans avoir besoin de créer une
11      * instance à chaque fois j'ai besoin à me connecter à la base
12      */
13     public static Connection cn = null;
14
15     // Constructeur vide
16     public ConnexionBDD () {
17
18     }
19
20     // méthode statique d'ouverture d'une connexion à appeler à chaque fois j'ai besoin de me connecter à la base
21     public static void OuvrirConnexion() {
22
23         String url = "jdbc:mysql://mysql2.pedago.local.enit.fr/gestionCommande";
24         String login = "student";
25         String password = "Enit@65";
26         try {
27             //Etape 1 : chargement driver
28             Class.forName("com.mysql.jdbc.Driver");
29
30             //etape 2 : récupération de la connexion
31             cn = DriverManager.getConnection(url, login, password);
32
33         } catch (ClassNotFoundException e) {
34             e.printStackTrace();
35         } catch (SQLException e) {
36             e.printStackTrace();
37         }
38         }
39
40      //  méthode statique de fermeture d'une connexion à appeler à chaque fois j'ai besoin de fermer la connexion à la base
41         public static void fermerConnexion  (){
42
43             try {
44                 cn.close();
45             } catch (SQLException e) {
46                 e.printStackTrace();
47             }
48     }
49
50 }
```

Dans la suite si on veut se connecter à la base il suffit de faire appel à ConnexionBDD.

Nous créons la classe *TestRequetes* dans le package *accesBaseDeDonnees*. Cette classe pourra servir pour écrire toutes les requêtes que nous souhaitons exécuter.
Chaque requête sera définie dans une méthode statique pour pouvoir l'appeler sans avoir besoin de créer une instance d'objet. Nous créons ici la méthode *insretClient* va permettre d'insérer un objet ***C*** de type client qu'elle reçoit en paramètre.
La méthode est statique et publique donc elle pourra être appelée de n'importe quelle classe ou méthode. Il suffira de lui envoyer en paramètre un objet de type client et elle exécute une requête SQL pour insérer un client dans la BDD.

```java
package accesBaseDeDonnees;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import entites.Client;

/* cette classe pourra servir pour écrire toutes les requêtes que nous souhaitons exécutées.
 * Chaque requête sera définie dans une méthode statique pour pouvoir l'appeler sans avoir besoin de créer
 * une instance d'objet.
*/
public class TestRequetes {

    /*/ une requête pour inserer un client dans la BDD.
    Throws remplace le fait de faire des try-catch sur les exceptions SQL qu'on pourra avoir dans le code.
    la méthode insretClient va permettre d'insérer un objet C de type client qu'elle recoit en parmètre.
    La méthode est statique et publique donc elle pourra être appelée de n'importe qu'elle classe ou méthode.
    Il suffit de lui envoyer en paramétre un objet de type client.
    */
     public static void insertClientBDD (Client C) throws SQLException{

        // Ouvrir une connexion en faisant appel à la méthode statique OuvrirConnexion
         ConnexionBDD.OuvrirConnexion();


            // Etape 1 creer requete insert
            String query_insertClient = " insert into client (nom,prenom)"+ " values (?, ?)";

            //Etape 2 : creer le preparedstatement
            PreparedStatement preparedStmt_insert = ConnexionBDD.cn.prepareStatement(query_insertClient);
             // avec le C.getNom() on récupère le nom de l'objet client et on l'affecte au champ nom de la table client dans la BDD
            preparedStmt_insert.setString(1, C.getNom());
             // avec le C.getPrenom() on récupère le prenom de l'objet client et on l'affecte au champ prenom de la table client dans la BDD
            preparedStmt_insert.setString(2, C.getPrenom());
            // Etape 3 executer la requete insert client
            preparedStmt_insert.execute();

            // fermer la connexion en faisant appel à la méthode statique fermerConnexion
            ConnexionBDD.fermerConnexion();
        }

}

```

Mode graphique (Java Swing)

Dans cette partie, nous allons développer notre première interface graphique avec avec la librairie Swing fournie par Java.
Les composants Java Swing que nous allons utiliser sont JFrame, JButton et JLabel.

- JFrame est une fenêtre avec un titre et une bordure qui correspond à la fenêtre principale de l'application. Elle est utilisée pour organiser d'autres composants, communément appelés composants enfants.
- JButton est un bouton poussoir utilisé pour effectuer une action.
- JLabel est un composant utilisé pour afficher un texte court ou une image, ou les deux.
- JTextField est un composant utilisé pour saisir du texte.

Tout d'abord, créez une classe nommée GUI possédant une méthode main qui hérite de la classe JFrame.

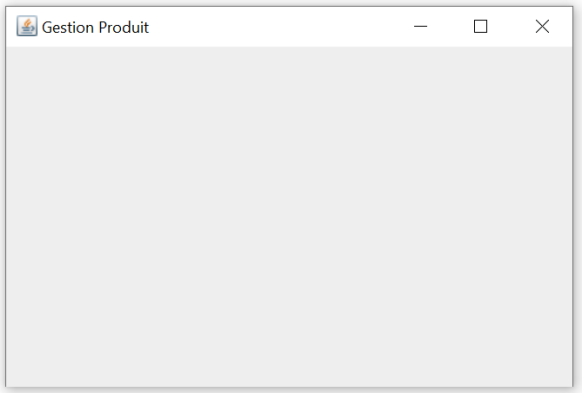
Déclarez la méthode *initUI()* qui va définir le titre de l'interface ("Gestion Client"), la taille (450,300), l'affichage centré de l'interface sur l'écran, et l'action à faire une fois l'interface est fermée en utilisant les méthodes suivantes:

- **setTitle(String title) :** Cette méthode modifie le titre de la fenêtre qui va apparaître dans la barre de titre et également dans la barre des tâches.
- **setSize(int width, int height) :** Cette méthode permet de modifier la taille de la fenêtre.
- **setLocationRelativeTo(Component c) :** Cette méthode permet de positionner la fenêtre par rapport à un composant. En indiquant un composant null, elle va se placer automatiquement au milieu de l'écran.
- **setDefaultCloseOperation(int operation) :** Cette méthode permet de configurer l'action qui va être exécutée lors de la fermeture de la fenêtre.

Dans le main, instanciez une nouvelle Jframe appelée *window.*
Par défaut, la fenêtre JFrame n'est pas visible. Pour la rendre visible, il suffit d'utiliser une méthode dédiée *setVisible(boolean)* pour changer sa propriété de visibilité.

GUI.java

```
1  package interfaceGraphique;
2
3  import java.awt.EventQueue;
4  import javax.swing.JFrame;
5
6  public class GUI extends JFrame {
7
8      public GUI() {
9
10         initUI();
11     }
12
13     private void initUI() {
14
15         setTitle("Gestion Produit");
16         setSize(450, 300);
17         setLocationRelativeTo(null);
18         setDefaultCloseOperation(EXIT_ON_CLOSE);
19     }
20
21     public static void main(String[] args) {
22
23         EventQueue.invokeLater(() -> {
24
25             GUI window = new GUI();
26             window.setVisible(true);
27         });
28     }
29 }
```



Il faut désormais ajouter dans la fenêtre les composants visuels avec lesquels l'utilisateur va interagir.

Déclarez trois JLabels *labelAjout*, *labelNom*, et *labelPrenom*.

Déclarez deux zones de texte *nomClient* et *prenomClient*. Pour ce faire, on fait référence à la classe JTextField du package javax.swing.

Ajoutez un JPanel *panel.* En effet, la classe JPanel est un conteneur utilisé pour regrouper et organiser des composants.

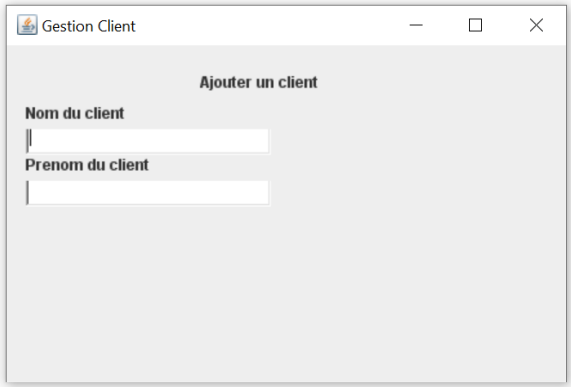
GUI.java

```
1  package interfaceGraphique;
2
3  import java.awt.EventQueue;
4  import java.awt.TextField;
5
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8  import javax.swing.JPanel;
9
10 public class GUI extends JFrame {
11
12     // Déclarer les label
13     private JLabel labelAjout, labelNom, labelPrenom;
14     // Déclarer les textfield
15     private TextField nomClient, prenomClient;
16     // Déclarer le Panel
17     private JPanel panel;
18
19     public GUI() {
20
21         initUI();
22     }
23
24     private void initUI() {
25
26         setTitle("Gestion Client");
27         setSize(450, 300);
28         setLocationRelativeTo(null);
29         setDefaultCloseOperation(EXIT_ON_CLOSE);
30
31     }
```

Dans la méthode initUI(), instanciez les labels, les zones de texte, et le panel déclarés comme montré dans la figure suivante.
Pour positionner ces composants, utilisez la méthode *setBounds(int x, int y, int largeur, int hauteur)*.
Ajoutez les labels et les zones de texte créés dans le panel. Puis, ajoutez le panel dans le JFrame.

GUI.java

```
    private void initUI() {

        setTitle("Gestion Client");
        setSize(450, 300);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // créer un label pour afficher "Ajouter un client"
        labelAjout = new JLabel("Ajouter un client");
        labelAjout.setBounds(150, 16, 190, 25);

        // créer un label pour afficher "Nom du client"
        labelNom = new JLabel("Nom du client");
        labelNom.setBounds(15, 40, 190, 25);

        // créer un textfield pour taper un texte
        nomClient = new TextField();
        nomClient.setBounds(15, 60, 190, 25);

        // créer un label pour afficher "Prenom du client"
        labelPrenom = new JLabel("Prenom du client");
        labelPrenom.setBounds(15, 80, 190, 25);

        // créer un textfield pour taper un texte
        prenomClient = new TextField();
        prenomClient.setBounds(15, 100, 190, 25);

        // créer un panel
        panel = new JPanel();
        panel.setLayout(null);

        // Ajouter les label et textfield dans le panel
        panel.add(labelAjout);
        panel.add(labelNom);
        panel.add(nomClient);
        panel.add(labelPrenom);
        panel.add(prenomClient);

        // Ajouter le panel au frame
        this.add(panel);
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(() -> {

            GUI window = new GUI();
            window.setVisible(true);
        });
    }
}
```



Le composant pour créer un bouton est le JButton.

Déclarez un nouveau JButton *button*.

GUI.java

```java
package interfaceGraphique;

import java.awt.EventQueue;
import java.awt.TextField;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class GUI extends JFrame {

    // Déclarer les label
    private JLabel labelAjout, labelNom, labelPrenom;
    // Déclarer les textfield
    private TextField nomClient, prenomClient;
    // Déclarer le Panel
    private JPanel panel;
    // Déclarer un boutton
    private JButton button;

```

En cas d'erreur vérifiez vos imports !!
Instanciez *button*, fixez sa position et ajoutez le dans le panel.

GUI.java

```java
        // créer un textfield pour taper un texte
        prenomClient = new TextField();
        prenomClient.setBounds(15, 100, 190, 25);

        // créer un bouton
        button = new JButton("Ajouter");
        button.setBounds(320, 150, 80, 25);

        // créer un panel
        panel = new JPanel();
        panel.setLayout(null);

        // Ajouter les label et textfield dans le panel
        panel.add(labelAjout);
        panel.add(labelNom);
        panel.add(nomClient);
        panel.add(labelPrenom);
        panel.add(prenomClient);
        panel.add(button);

        // Ajouter le panel au frame
        this.add(panel);
    }
```

On va maintenant attribuer une action au bouton. C'est-à-dire indiquer ce qui va devoir se passer en cas de clic sur le bouton.

En cas de clic sur le bouton, on veut afficher à l'utilisateur le message suivant : "Client ajouté avec succès". Pour ce faire, créez un nouveau JLabel *labelResult.*

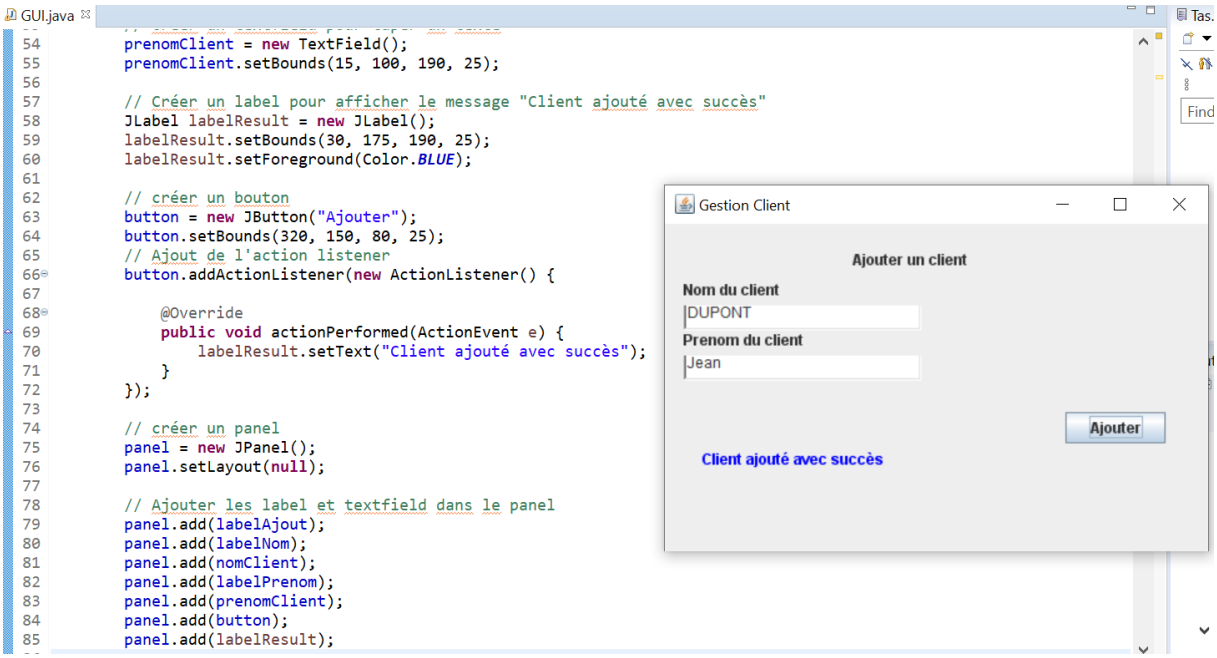
En effet, Swing utilise une architecture par événements. En effet, en cas de clic sur le bouton par exemple, un événement est provoqué. Un événement est ensuite transmis aux objets qui vont réagir en conséquence. Ces objets sont appelés des listeners.

Dans le code, déclarez un listener et ajoutez le au bouton en utilisant la méthode *addActionListener().*

ActionListener devra obligatoirement contenir la fonction actionPerformed. Cette fonction sera appelée lorsqu'un événement se produit sur le bouton.

Dans actionPerformed, mettez à jour le texte de *labelResult* pour afficher "Client ajouté avec succès".

Ajoutez *labelResult* dans le panel.



Comment passer d'une fenêtre à une autre en cliquant sur un bouton ?

On va créer une nouvelle class graphique pour la gestion des commandes pour l'appeler à partir d'une autre fenêtre.

```
1  package interfaceGraphique;
2  import java.awt.Color;
12
13 public class GUICommande extends JFrame {
14
15     // on déclare un objet statique pour l'appeler depuis n'importe qu'elle autre classe
16         static GUICommande window;
17     // Déclarer les label
18         private JLabel labelcmd;
19         // Déclarer le Panel
20         private JPanel panel;
21
22         public GUICommande() {
23
24             initUI();
25         }
26
27         private void initUI() {
28
29             setTitle("Gestion Commande");
30             setSize(450, 300);
31             setLocationRelativeTo(null);
32             setDefaultCloseOperation(EXIT_ON_CLOSE);
33
34             // créer un label pour afficher "Ajouter un client"
35             labelcmd = new JLabel("Commande");
36             labelcmd.setBounds(150, 16, 190, 25);
37
38                     // créer un panel
39             panel = new JPanel();
40             panel.setLayout(null);
41
42             // Ajouter les label et textfield dans le panel
43
44             panel.add(labelcmd);
45
46             // Ajouter le panel au frame
47             this.add(panel);
48         }
49
50         public static void main(String[] args) {
51
52             EventQueue.invokeLater(() -> {
53                 window = new GUICommande ();
54                 window.setVisible(true);
55             });
56         }
57 }
58
```

Dans la suite nous allons faire quelques modifications dans la classe GUI.

1- Déclarer une variable statique *window* de type GUI.

```
public class GUI extends JFrame {

    static GUI window;

    // Déclarer les label
    private JLabel labelAjout, labelNom, labelPrenom;
    // Déclarer les textfield
    private TextField nomClient, prenomClient;
    // Déclarer le Panel
    private JPanel panel;
    // Déclarer un boutton
    private JButton button;

    public GUI() {

        initUI();
    }
```

2- Modifier la méthode *main* de la classe

```
public static void main(String[] args) {

    EventQueue.invokeLater(() -> {

        window = new GUI();
        window.setVisible(true);
    });
}
```

3- Mettre à jour le code à l'intérieur de la méthode *actionPerformed*

```
@Override
public void actionPerformed(ActionEvent e) {
    labelResult.setText("Client ajouté avec succ\0s");
    // Pour fermer et faire disparaitre le window Gestion Client on utilise hide.
    window.hide();
    //Pour lancer une nouvelle fentre après la fermeture de window, on utilise le code suivant pour lancer le GUICommande
    GUICommande NouvelleWindow = new GUICommande ();
    NouvelleWindow.setVisible(true);
}
```

Lien du Mode graphique, Classes et BDD

Dans cette partie nous allons lier les différentes parties manipulées précédemment.
On va créer la classe *LinkGuiBDD* dans le package interfaceGraphqiue. Cette classe permet de récupérer les informations d'un client à travers l'interface graphie, ensuite on crée un objet de type client, et finalement on l'enregistre dans la base de données.

```
package interfaceGraphique;
import java.awt.Color;
import java.awt.EventQueue;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import accesBaseDeDonnees.ConnexionBDD;
import accesBaseDeDonnees.TestRequetes;
import entites.Client;

public class LinkGuiBDD  extends JFrame{

      // Déclarer un window static pour l'utiliser partout dans le code
        static LinkGuiBDD window;
      // D□clarer les label
            private JLabel labelAjout, labelNom, labelPrenom;
```

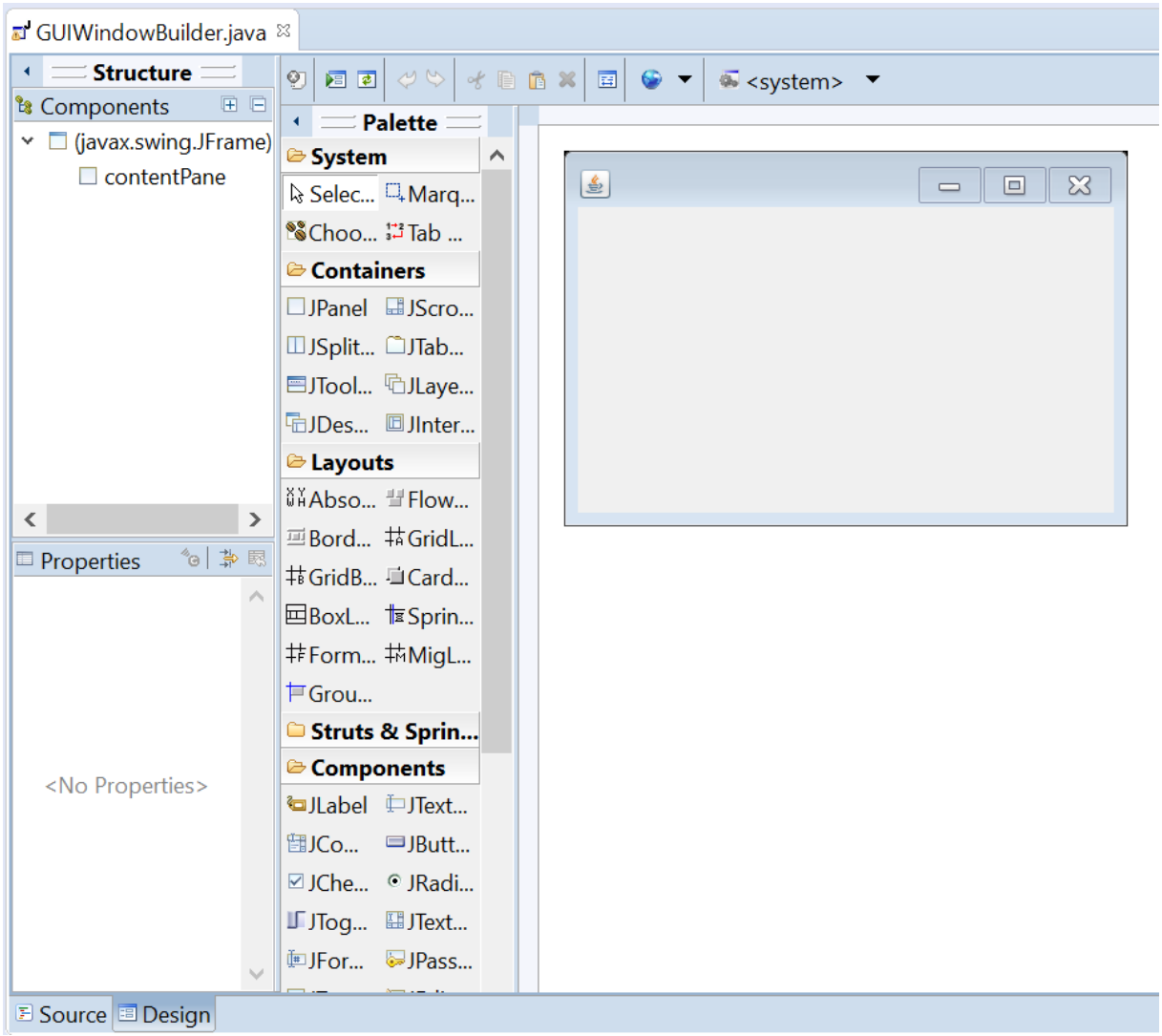
```
        // D□clarer les textfield en utilisant jtextfield
        private JTextField nomClient, prenomClient;
        // D□clarer le Panel
        private JPanel panel;
        // D□clarer un boutton
        private JButton button;
    // constructeur de la classe
        public LinkGuiBDD() {

            initUI();
        }

        private void initUI() {

            setTitle("Gestion Client BDD");
            setSize(450, 300);
            setLocationRelativeTo(null);
            setDefaultCloseOperation(EXIT_ON_CLOSE);

            // cr□er un label pour afficher "Ajouter un client"
            labelAjout = new JLabel("Ajouter un client");
            labelAjout.setBounds(150, 16, 190, 25);

            // cr□er un label pour afficher "Nom du client"
            labelNom = new JLabel("Nom du client");
            labelNom.setBounds(15, 40, 190, 25);

            // cr□er un textfield pour taper un texte
            nomClient = new JTextField();
            nomClient.setBounds(15, 60, 190, 25);

            // cr□er un label pour afficher "Prenom du client"
            labelPrenom = new JLabel("Prenom du client");
            labelPrenom.setBounds(15, 80, 190, 25);

            // cr□er un textfield pour taper un texte
            prenomClient = new JTextField();

            prenomClient.setBounds(15, 100, 190, 25);

            // Créer un label pour afficher le message "Client ajouté
avec succès"
            JLabel labelResult = new JLabel();
            labelResult.setBounds(30, 175, 190, 25);
            labelResult.setForeground(Color.BLUE);

            // créer un bouton
            button = new JButton("Ajouter");
            button.setBounds(320, 150, 80, 25);
            // Ajout de l'action listener
            button.addActionListener(new ActionListener() {
```

```
                        // actionPerformed va contenir el code qui sera
exécuté lors du click sur le bouton
                        @Override
                        public void actionPerformed(ActionEvent e) {
                              // Creation de l'objet client C
                              Client C= new Client ();
                              // initialiser le prénom de C avec le contneu
saisi des textfields
                              C.setPrenom(prenomClient.getText());
                              C.setNom(nomClient.getText());
                              System.out.println(C.getNom());
                              try {
            // on fait appel à la méthode statique insertClientBDD de la
classe TestRequetes pour executer la requère d'insertion de l'olobjet C
                                    TestRequetes.insertClientBDD(C);
                              } catch (SQLException e1) {
                                    // TODO Auto-generated catch block
                                    e1.printStackTrace();
                              }

                              labelResult.setText("client ajouté à la
BDD");
                        }
                  });

                  // créer un panel
                  panel = new JPanel();
                  panel.setLayout(null);

                  // Ajouter les label et textfield dans le panel
                  panel.add(labelAjout);
                  panel.add(labelNom);
                  panel.add(nomClient);
                  panel.add(labelPrenom);
                  panel.add(prenomClient);
                  panel.add(button);
                  panel.add(labelResult);

                  // Ajouter le panel au frame
                  this.add(panel);
            }

            public static void main(String[] args) {

                  EventQueue.invokeLater(() -> {

                        window = new LinkGuiBDD();
                        window.setVisible(true);
                  });
            }

}
```

Remarque :

Pour créer une interface graphique, vous pouvez utiliser le plugin Eclipse **WindowBuilder** qui est un concepteur Java GUI visuel. Il suffit d'utiliser le concepteur visuel et le code Java sera automatiquement généré pour vous.

Pour ce faire, cliquez-droit sur le package interfaceGraphique → New → Other → WindowBuilder → SwingDesigner → JFrame.
Donnez le nom **GUIWindowBuilder** à votre classe et cliquez sur Finish. Vous obtenez alors une classe déjà préparée.
Un onglet Design s'affiche à côté d'un onglet Source en dessous de votre code. En cliquant sur cet onglet, vous passez sur le modeleur graphique et vous allez pouvoir créer votre vue à partir de la palette des composants.
Vous pouvez alors glisser-déposer des composants dans l'interface et vous pourrez remarquer que le code source est modifié en conséquence.



Réalisation du projet et évaluation

L'évaluation de cet EC porte sur la réalisation d'un projet en équipe (Trinôme ou Binôme) dans lequel l'équipe projet choisit un domaine d'application pour la réalisation d'une application de gestion.

La première étape consiste à conceptualiser au moins 12 à 16 classes du domaine choisi via un diagramme de classe UML. La deuxième étape consiste au développement en JAVA du projet.

Dans chaque projet, il serait obligatoire de développer toutes les classes, de faire une classe principale (main), de faire une interface graphique avancée permettant de manipuler tous les objets et de faire une base de données avec différents types de requête (Insert, Select, Update, Delete) manipulant ces objets.

Indications sur le planning de réalisation du Projet

La réalisation du projet se déroule en mode synchrone (en classe) et asynchrone (chez vous) à partir de la deuxième ou la troisième séance de TP (selon avancement dans le tutoriel).

Troisième séance :

- 2h pour le choix de sujet et réalisation du diagramme de classe
- 2h pour le début du développement

Asynchrone (hors présentiel) : Avancement sur le développement

Quatrième séance

- 3h pour la finalisation du développement
- 1h pour l'évaluation

Pour que l'équipe projet peut avancer en parallèle, il est recommandé d'avoir un workspace partagé (dropbox ou nextcloud) mais les fichiers doivent être synchronisés sur la machine locale de chaque membre.