|  |  |
| --- | --- |
| |  | | --- | |  | |
| C34-VM Programmation Orientée Objet 1  **MODULE 01**  **Introduction à Java et Concepts Orienté Objet – Partie 1**  Automne 2023 |

Table des matières

[1. Préparation 3](#_Toc142330444)

[2. JAVA - Introduction au langage Java 4](#_Toc142330445)

[2.1 Qu’est-ce que Java 4](#_Toc142330446)

[Pourquoi Java? 5](#_Toc142330447)

[2.2 Futur de Java 5](#_Toc142330448)

[L01A\_Partie 1 - Première application en Java 6](#_Toc142330449)

[2.3 Portabilité du langage Java 7](#_Toc142330450)

[Utilisation d’une machine virtuelle Java : JVM. 7](#_Toc142330451)

[Compilation du programme Java en bytecode 7](#_Toc142330452)

[L01A\_Partie 2 – Compilation et exécution d’une application en Java 8](#_Toc142330453)

[3. OO - La programmation Orientée Objet 9](#_Toc142330454)

[3.1 Paradigme 9](#_Toc142330455)

[Comparaison avec un langage de programmation fonctionnel 9](#_Toc142330456)

[Les champs 10](#_Toc142330457)

[Les méthodes 10](#_Toc142330458)

[L01B\_Classes et objets en Java-1 12](#_Toc142330459)

[3.2 La documentation du code en Java 12](#_Toc142330460)

[4. JAVA - Les Packages en Java 13](#_Toc142330461)

[4.1 Le mot clé import 14](#_Toc142330462)

[4.2 Les modules Java 14](#_Toc142330463)

[L01C\_Packages Java 14](#_Toc142330464)

[5. JAVA - La hiérarchie des classes 15](#_Toc142330465)

[6. JAVA - Les modificateurs d’accès (Access Control Modifiers) 16](#_Toc142330466)

[6.1 Modificateur d’accès pour les membres d’une classe 16](#_Toc142330467)

[6.2 Modificateur d’accès pour une classe 19](#_Toc142330468)

[6.3 Meilleures pratiques pour les modificateurs d’accès 19](#_Toc142330469)

[7. Méthodes d’accès et de mutation 20](#_Toc142330470)

[L01D\_Modificateurs d’accès 20](#_Toc142330471)

[L01E\_Exercices sur les Modificateurs d’accès 21](#_Toc142330472)

[TRAVAIL PRATIQUE 1 21](#_Toc142330473)

# Préparation

Vous pouvez installer IntelliJ IDEA à la maison, on suggère la version portable disponible sur le site ci-dessous :

<https://portapps.io/app/intellij-idea-community-portable/>

Choisissez la version **2022.2-52**.

Pour un des modules, nous utiliserons l’application en ligne **draw.io** pour faire des diagrammes UML. Vous pouvez vous familiariser avec cet outil si vous désirez prendre un peu d’avance.

# JAVA - Introduction au langage Java

## Qu’est-ce que Java

**Langage de programmation :**

* Basé sur des classes et orienté objet
* Portable - la même application s’exécute sur plusieurs plateformes : Windows, Linux, macOS, UNIX…
* Distribué – conçu pour être facilement utilisé dans une structure client-serveur
* Syntaxe similaire à C++, mais plus simple : aisé à apprendre
* Grand nombre de librairies standard
* Multithread

**Historique :**

Développé à l’origine chez Sun Microsystems par James Gosling

Se nommait Oak à l’origine et était développé pour installations sur des petits appareils (consoles, télévisions, récepteurs) pour faire de la vidéo sur demande

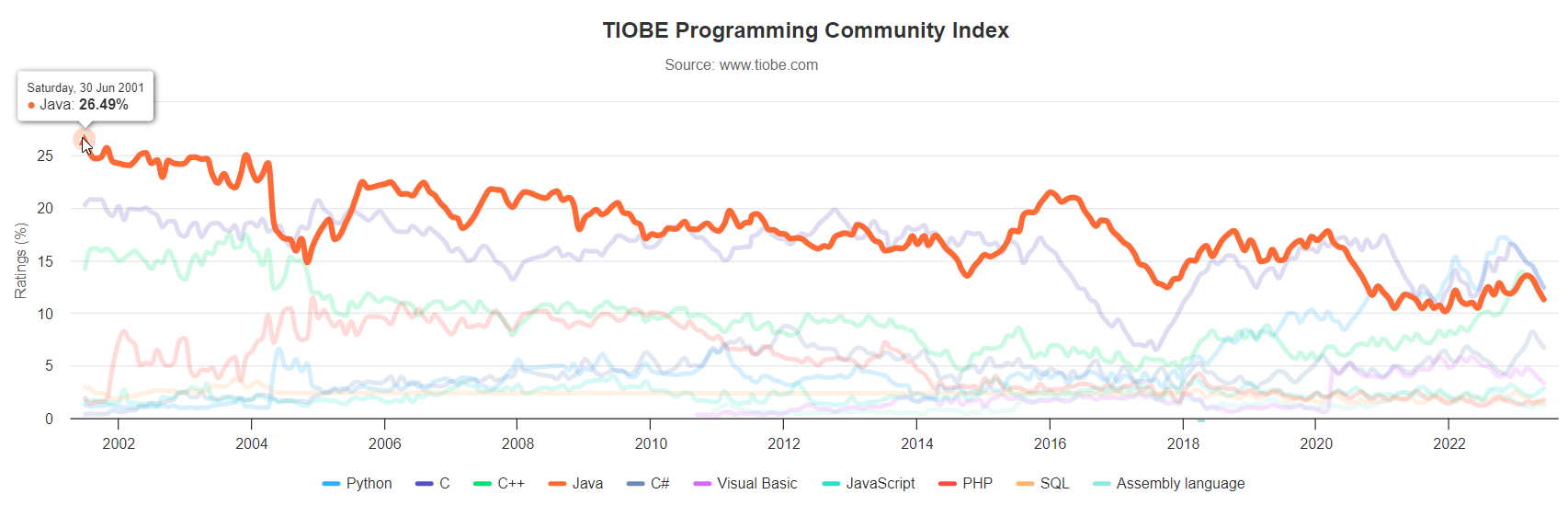
Nom changé pour Java : réorienté pour le web.

Repris par Oracle suite à son acquisition de SUN Microsystems

## Pourquoi Java?

Plus de 90% des compagnies Fortune 500 mentionnent Java comme un des langages de programmation utilisés dans leur liste de technologies.

L’index TIOBE (mesure de popularité, pas % d’utilisation) montre Java en haut de la liste des langages de programmation. Ci-dessous en date du 2 juin 2023.



Java peut être utilisé dans plusieurs contextes dont :

* Des applications Web
* Des applications mobiles
* Des applications de stations, serveurs et distribuées (desktop, server, distributed)
* Autres

## Futur de Java

La courbe d’adoption et de popularité de Java est décroissante.

Plusieurs autres langages de programmation prennent de l’essor (Python, Kotlin, Swift…)

On se pose la question à savoir si Java sera bientôt obsolète.

La réponse à court et moyen terme est : non.

* Java est un langage apprécié des programmeurs.
* C’est une compétence acquise par un très grands nombres de programmeurs.
* Grande communauté (beaucoup de support, tutoriels, exemples, ressources).
* Il y a un très grand nombre de logiciels écrits en Java qui ne seront pas réécrits dans un autre langage à court terme (très dispendieux).
* Oracle continue de développer activement Java et de l’adapter aux nouveaux besoins technologiques (récemment 2 versions par année).

### L01A\_Partie 1 - Première application en Java

Le laboratoire L01A\_Partie 1 permet de coder une première application en Java à titre exploratoire.

## Portabilité du langage Java

Caractéristique majeure de Java : le même programme peut être exécuté sur différents systèmes d’exploitation sans être recompilé.

Autres langages : code est compilé séparément pour Windows, Linux, MacOS etc. et produits plusieurs exécutables qui doivent être distribués et maintenus séparément ($$$).

Java accomplit ceci par :

### Utilisation d’une machine virtuelle Java : JVM.

Une **JVM** existe pour chaque système d’exploitation et sert de traducteur entre le programme Java et le SE. La JVM doit être installé sur un SE pour que des programmes Java puissent être exécutés.

On installe généralement sur un système d’exploitation le JRE (Java Runtime Environment) qui lui correspond, et qui inclus les librairies Java et la JVM.

On peut aussi installer le JDK (Java Development Kit) qui inclus le JRE, mais aussi les outils de développement.

### Compilation du programme Java en bytecode

Un fichier source java doit avoir l’extension **.java**.

Le compilateur java, **javac**, produit un résultat intermédiaire appelé **bytecode**, dont l’extension doit être **.class**.

Les fichiers bytecode ne sont pas natif au système d’exploitation, ils doivent être exécutés par une JVM.

Le bytecode doit être traduit en code natif pour le système d’exploitation hôte. Ce travail est fait par le compilateur de bytecode JIT (Just In Time compiler), qui fait partie de la JVM.

**Au développement :**

Code java -> compilateur javac -> fichier bytecode avec extension .class

**Avant d’exécuter un programme java sur un ordinateur :**

Installation du JRE correspondant au système d’exploitation, qui inclus la JVM. (JRE pour Windows, JRE pour Linus, JRE pour MacOS etc.)

**À l’exécution d’un programme Java :**

Fichier bytecode est compilé en langage machine natif au système d’exploitation par le compilateur JIT et exécuté.

**NOTE** : le système d’exploitation mobile Androide inclus sa propre machine virtuelle Java qui convertit les fichiers bytecode en un autre format avant de les utiliser, d’une façon optimisée pour les appareils mobiles (plus rapide, moins d’utilisation de la batterie).

**Relation entre les composants**

|  |  |  |  |
| --- | --- | --- | --- |
| JDK (javac, jar, outils débogage, APIs)   |  |  |  | | --- | --- | --- | | JRE (java, javaw, librairies)   |  |  | | --- | --- | | JVM   |  | | --- | | Compilateur JIT | | | |

L’aptitude à exécuter le même programme sur des plateformes différentes s’appelle la **portabilité** et est une des raisons principales de l’adoption à grand échelle de Java.

### L01A\_Partie 2 – Compilation et exécution d’une application en Java

Le laboratoire L01A-Partie 2 permet de compiler et exécuter une première application en Java à titre exploratoire.

# OO - La programmation Orientée Objet

## Paradigme

La programmation orientée objet permet de regrouper ensemble les **données** d’une entité/d’un sujet et les **actions** qu’on peut effectuer sur cette entité/sujet dans une même structure.

### Comparaison avec un langage de programmation fonctionnel

Soit l’exemple suivant :

On veut faire une application de vente en ligne et gérer les produits disponibles à la vente.

On veut pouvoir effectuer les opérations suivantes :

* Afficher le nom du produit
* Afficher le prix du produit
* Créer une commande avec les informations du produit et du client qui fait l’achat

Le **nom** et le **prix** du produit sont des **données**. Elles pourraient être conservées dans une base de données.

Le **nom** et **le courriel** du client sont aussi des **données** qui pourraient être conservées dans la base de données.

**L’affichage** et la **création de la commande** sont des **actions**. Ces actions ne sont pas des données conservées dans une base de données mais plutôt du code qui accomplit une tâche.

Dans un langage de programmation fonctionnel (basé sur les fonctions) on pourrait coder les fonctions suivantes :

**AfficherNomProduit**

Cette fonction inclurait le code pour accéder à la donnée Nom d’un produit spécifié, et le code requis pour en faire l’affichage dans la page web.

**AfficherPrixProduit**

Cette fonction inclurait le code pour accéder à la donnée Prix d’un produit spécifié, et le code requis pour en faire l’affichage dans la page web.

**CreerCommande**

Cette fonction inclurait le code pour accéder aux informations du Produit ET aux informations du Client et le code requis pour assembler ces informations ensemble pour créer une Commande et envoyer ces nouvelles informations dans la base de données.

En programmation fonctionnelle, la **fonction** est l’élément principal de la structure du code. Les **données** et les **actions** sont manipulées indépendamment, à l’intérieur de ces fonctions. Par exemple les données du Produit sont manipulées dans plus d’une fonction.

En **Programmation Orientée Objet (POO)**, on regroupe les données et les actions qui se rapportent à un sujet commun (une entité commune), dans une seule structure. C’est le développeur qui décide de cette structure.

Chaque sujet ou entité se nomme un **Objet**.

Dans notre exemple on pourrait définir 3 types d’objets :

1. Produit
2. Client
3. Commande

### Les champs

En Java, les **données** se nomment **champs (fields)**. D’autres langages et terminologies utilisent aussi **attributs** et **propriétés**.

Pour chacun de ces objets on pourrait établir les champs (données) :

**Produit**

* Nom
* Prix

**Client**

* Nom
* Courriel

**Commande**

* NoCommande
* Client
* Produit
* QteProduit

### Les méthodes

En POO, les actions/comportements qui peuvent être effectuées sur un objet se nomment **Méthodes (methods)**.

Pour chaque objet on pourrait établir les méthodes suivantes (en plus des champs) :

**Produit**

* Nom
* Prix
* AfficherPrix

**Client**

* Nom
* Courriel
* ModifierCourriel

**Commande**

* NoCommande
* Client
* Produit
* QteProduit
* CréerCommande
* AfficherCommande

Les langages de programmation qui ne sont pas orientés objet (fonctionnel, procédural) gèrent séparément les données et les actions que l’on peut effectuer sur ces données.

Le code pour accéder aux champs d’un objet (données) et le code pour exécuter les méthodes d’un objet (actions) sont regroupés au même endroit.

La définition de la structure d’un objet et le code qui y est rattaché s’appelle une **classe**.

La classe agit comme un modèle ou un plan pour construire un objet donné.

Lorsqu’on manipule les objets dans la partie principale du programme, on peut créer plusieurs copies d’objets basés sur la même classe, ces copies se nomment **instances**.

Dans notre exemple, on aurait une classe **Produit** avec tout le code requis pour définir les champs et méthodes d’un objet Produit.

Dans la partie principale du code, on pourrait manipuler 2 produits : **Ordinateur**, **Tablette**, soit 2 objets construits à partir de la même classe **Produit**.

Une **classe** définit la structure d’un objet (Produit) et tout son code, alors qu’une instance d’**objet** représente une entité spécifique de cet objet (Ordinateur, Tablette).

En POO, on définit d’abord les classes, puis on utilise ces nouvelles structures dans le code principal.

La classe agit alors comme un **type**, appliqué à un objet (un objet est de type Client ou de type Produit ou de type Commande etc.)

On utilisation la notation du point **(.)** pour identifier l’attribut ou la méthode d’un objet.

On manipule généralement les objets par l’entremise d’une variable.

Au moment d’utiliser les classes dans la partie principale du code on obtient quelque chose qui ressemble à :

Produit unProduit = new Produit()

Créé un objet assigné à la variable unProduit, de type Produit. unProduit a alors accès à tous les champs et toutes les méthodes définies et codées dans la classe Client.

unProduit.nom = Ordinateur

Définit que le champ Nom de l’objet unProduit est Ordinateur

unProduit.prix = 1000

Définit le champ prix de l’objet unProduit à $1000.

unProduit.AfficherPrix()

Produira l’affichage du prix de unProduit. C’est une méthode (action).

Dans la même partie du code principal on pourrait manipuler une deuxième instance de Produit :

unAutreProduit.nom = Tablette

unAutreProduit.prix = 400

unAutreProduit.AfficherPrix()

On a alors 2 objets : unProduit et unAutreProduit, tous les 2 des instances de la classe Produit.

Les **champs** et les **méthodes** d’une classes sont les **membres** de cette classe.

### L01B\_Classes et objets en Java-1

Le laboratoire L01B permet de voir comment les concepts de POO sont appliqués en Java.

## La documentation du code en Java

Les notions vues dans vos cours précédents à propos de l’importance de la documentation du code s’appliquent aussi à vos programmes en Java.

La documentation est essentielle et fait partie des barèmes de correction de vos travaux et de vos évaluations.

Un commentaire sur une seule ligne commence avec 2 barres obliques : **//**

Un commentaire sur plusieurs lignes est encadré par : **/\* \*/**

# JAVA - Les Packages en Java

Comme chaque classe = 1 fichiers, une application Java est composée de plusieurs fichiers.

Les Packages permettent de consolider plusieurs fichiers en 1 seul fichier, un peu comme un fichier d’archives compressé (format .zip).

Les packages Java sont en format **JAR** (Java ARchive) et ont l’extension **.jar**.

Si on exécute une application à partir de fichiers **.class**, tous les fichiers doivent être présents dans le même dossier (où référencés d’une façon ou d’une autre).

Avec un fichier JAR, tous les fichiers sont présents dans un seul fichier.

La JVM peut exécuter un fichier de type JAR.

Une application peut être composée de plusieurs packages.

Les Packages regroupent ensemble des classes qui servent à une fonctionnalité commune.

C’est le développeur qui décide comment son application est divisée en Package.

**Exemples :**

* Une application simple peut n’avoir qu’un seul Package
* Une application de gestion d’un concessionnaire automobile pourrait avoir un Package Autos pour la gestion de son parc de véhicules et un Package Employes pour la gestion de son personnel.
* Pour la même application de gestion d’un concessionnaire automobile, on pourrait décider de diviser en plusieurs Packages comme : Modeles, InventaireAuto, Ventes, Employes etc.
* On choisit de faire des Packages séparés lorsque les fonctionnalités sont vraiment différentes ou que l’application a une certaine taille/complexité ou pour pouvoir partager ou distribuer des parties de l’application facilement etc.

Les Packages présentent plusieurs avantages dont :

* Une facilité à partager une application ou une partie de l’application entre développeurs
* La possibilité de déployer facilement l’application ou une partie de l’application de développement à test et de test à production.
* Le classement logique des classes en une structure plus facile à gérer pour les développeurs
* La sécurité, en isolant certaines parties du code dans un package seulement
* Gestion des espaces de noms : dans une application sans Packages, chaque classe doit avoir un nom différent, mais le même nom peut se retrouver dans des packages différents.
* Et plus…

## Le mot clé import

Lorsqu’on désire accéder aux membres d’une classe d’un autre package à partir d’une classe donnée, il existe différents moyens :

1. Sans le mot clé import
   1. Utiliser la notation **nomDePackage.Classe** var = new **nomDePackage.Classe**

Exemple

nomDePackage.nomDeClasse var = new nomDePackage.nomDeClasse

1. Avec le mot clé import pour une classe spécifique
   1. Utiliser : **import nomDePackage.Classe;** au début du code, après la ligne du package
   2. Cette classe est alors disponible pour tout le code de la classe qui importe
2. Avec le mot clé import pour toutes les classes d’un package
   1. Utiliser : **import nomDePackage.\*;** au début du code, après la ligne du package
   2. Toutes les classes du package importé sont alors disponible pour le code de la classe qui importe

## Les modules Java

Les modules (JPMS pour Java Platform Modules System) sont une façon d’organiser le code en regroupement de plusieurs package

L’objectif est de rendre le code encore plus réutilisable.

Ils ont été introduits avec Java 9, en 2017.

Un module doit comprendre toutes les ressources utilisables pas ses packages.

Les modules fournissent une couche supplémentaire d’abstraction et de sécurité, permettant de contrôler quelles classes de quels package d’un module sont accessible par les classes des autres packages dans d’autres modules.

Les modules fournissent aussi d’autres avantages et dépassent le cadre de ce cours.

Importer des classes d’un autre module requiert la mise en place de dépendances. Non couvert dans ce cours.

### L01C\_Packages Java

Le laboratoire L01C permet de voir comment créer et utiliser les packages en Java sous IntelliJ IDEA et comment les classes sont ajoutées à un package.

# JAVA - La hiérarchie des classes

En tant que langage OO, Java autorise la création de sous-classes.

Une sous-classe est une classe qui hérite les caractéristiques des membres d’une autre classe.

La classe qui reçoit les membres d’une autre est appelée **sous-classe** ou classe **enfant**.

La classe qui fournit les caractéristiques à une autre classe s’appelle **superclasse** ou classe **parente**.

On explore l’héritage, les sous-classes et les superclasses en détail plus loin dans le cours, mais il est utile d’en connaitre l’existence pour comprendre les concepts qui suivent.

Par exemple on pourrait définir une classe **Personne** qui aurait un champ **nom**.

On pourrait définir une classe **Client** comme une sous classe de **Personne** et lui créer un champ **numeroClient**.

La classe **Client** (sous-classe) aurait alors 2 champs, **nom** et **numeroClient**, le premier étant hérité de la classe **Personne** (superclasse).

Il y a différentes façons de déclarer des sous-classes, en voici un exemple, utilisé dans les prochains exercices.

|  |
| --- |
| public class Personne{  private String nom;  }  class Client extends Personne{  private String numeroClient  }  class Employe extends Personne{  private String numeroAssuranceSociale  } |

# JAVA - Les modificateurs d’accès (Access Control Modifiers)

Un modificateur d’accès définit dans quelle partie du code peut être utilisé une variable, une classe, ou un membre d’une classe (champs méthode etc.).

On peut voir les modificateurs d’accès dans le code vu jusqu’à maintenant. Dans l’exemple ci-dessous, les modificateurs d’accès sont surlignés.

Note : dans l’exemple, tous les modificateurs d’accès sont **public**, ce n’est pas un exemple typique.

Une image contenant texte

Description générée automatiquement

## Modificateur d’accès pour les membres d’une classe

Le modificateur d’accès d’un membre d’une classe peut être :

* **private** (privé) – accessible à l’intérieur de cette classe seulement
* **public** (publique) – accessible à partir d’autres classes (même package ou autres package)
* **default** (par défaut) – accessible à l’intérieur du package seulement
* **protected** (protégé) - accessible dans son package seulement (comme default) ou dans une sous-classe de cette classe d’un autre package.

Dans l’exemple précédent, le champ **prixSuggere** est public.

Il peut être utilisé dans la classe **GestionnaireVehicule** avec la syntaxe suivante :

Une image contenant texte

Description générée automatiquement

Cette syntaxe, sans validation, autoriserait par exemple un prix suggéré négatif, ce qui ne fait pas de sens logiquement.

On fait plutôt une méthode qui attribue une valeur à **prixSuggere** et dans cette méthode on ajoute le code pour interdire des valeurs négatives.

On peut alors rendre l’attribut **prixSuggere** **private**, mais la méthode **setPrixSuggere** **public**.

Le code pourrait ressembler à :

|  |
| --- |
| package vehicule;  public class ItemInventaire {  public String idItemInventaire;  public String idModele;  private double prixSuggere;   public void setPrixSuggere(double ps){  if (ps < 0){  System.*out*.println("Le prix suggéré ne peut pas être négatif");  }  else{  this.prixSuggere = ps;  }  } |

Dans la classe principale, **GestionVehicule**, le champ **prixSuggere**, qui est **private**, n’est plus accessible :

Une image contenant texte

Description générée automatiquement

Notre code force maintenant l’utilisation d’une autre syntaxe, qui fait appel à la méthode plus robuste **setprixSuggere** :

|  |
| --- |
| iv2.idItemInventaire = "aut320057"; iv2.idModele = "husan04"; iv1.setPrixSuggere(38000); |

Rappel:

**private**: ne peut être utilisé que dans cette classe

**public** : peut être utilisé dans une autre classe

Note : on attribue souvent **private** aux champs et **public** aux méthodes, c’est une des meilleures pratiques, sauf s’il y a un besoin pour faire autrement.

Si on ne met pas de modificateur d’accès pour un champ ou une méthode, le modificateur d’accès **default** est alors appliqué. Le modificateur default est aussi appelé **package-private** et signifie que ce membre de la classe peut être utilisé par cette classe et d’autres classes, mais à l’intérieur du package seulement, pas dans un autre package.

Le modificateur d’accès **protected** signifie que le membre est accessible dans son package seulement (comme défaut) ou dans une sous-classe de cette classe d’un autre package. On voit les sous-classes plus loin.

## Modificateur d’accès pour une classe

Seuls 2 des modificateurs d’accès peuvent être appliqués à une classe :

* default – accessible dans son package seulement
* public – accessible dans tous les package

Une classe ne peut pas être private ou protected.

**Exemple**

Une application a 3 package :

|  |  |
| --- | --- |
| **Package** | **Classes** |
| **gestionclient**  Classes pour la gestion des comptes clients | CompteClient  TypeClient |
| **gestionproduit**  Inclus les classes pour la gestion des produits à vendre | Produit  CategorieProduit  Fournisseur |
| **gestionvente**  Inclus les classes pour gérer les ventes, commandes, facturation etc | Commande  Vente  Facture |

Une classe **Produit** dans le package **gestionproduit** n’aura pas besoin de la classe **Facture** du package **gestionvente**. La classe **Facture** pourrait être de type **default**.

Mais la classe **Facture** a besoin des informations de produits et de clients. Les classes **Client** et **Produit** seraient donc de type **public** pour pouvoir être accédées dans le package **gestionvente**.

Il ne peut y avoir qu’une seule classe **public** dans un fichier java et cette classe doit porter exactement le même nom que le fichier. C’est principalement le cas de la classe qui contient la méthode main.

## Meilleures pratiques pour les modificateurs d’accès

Les règles de base sont les suivantes :

* Utiliser toujours le modificateur d’accès le plus restrictif pour répondre aux besoins du code
* Utiliser les champs **public** pour les constantes seulement (pour les classes, selon les besoins)

# Méthodes d’accès et de mutation

Si on suit les meilleures pratiques, les champs seront **private** et il faudra des méthodes **public** pour y accéder.

Une méthode qui permet de **récupérer/lire** la valeur d’un champ s’appelle : **méthode d’accès** (getter/accessor)

Une méthode qui permet de **modifier** la valeur d ’un champ s’appelle : **méthode de mutation** (setter/mutator)

Ces 2 méthodes standards n’ont typiquement pas d’autres fonctionnalités, sauf la validation de champ.

Sans être une obligation les conventions de nom suivantes sont universellement reconnues :

Méthode d’accès : **get**NomDeChamp

Méthode de mutation : **set**NomDeChamp

Exemple :

|  |
| --- |
| **public class Produit{  private String nomProduit; *//Les champs sont private* private int qteInventaire;   *//Constructeur* public Produit() {  }    *//Méthode d'accès pour nomProduit - public* public String getNomProduit() {  return this.nomProduit;  }   *//Méthode d'accès pour qteInventaire - public* public int getqteInventaire() {  return this.qteInventaire;  }   *//Méthode de mutation pour nomProduit - exemple avec validation* public boolean setNomProduit(String pNomProduit) {  if (pNomProduit.isEmpty()) {  return false;  }  else {  this.nomProduit = pNomProduit;  return true;  }  }    *//Méthode de mutation pour qteInventaire - exemple sans validation* public void setQteInventaire(String pQteInventaire) {   this.nomProduit = pNomProduit;  } }** |

Dans les prochains laboratoires, dans une classe avec des champs, écrivez **get** ou **set** dans l’éditeur IntelliJ IDEA pour voir ce qu’offre l’autocomplete.

### D01A\_Démo\_Modificateurs d’accès

Ce laboratoire de démonstration est fait en groupe et constitue un point de départ pour le laboratoire L01D qui suit..

### L01D\_Modificateurs d’accès

Le laboratoire L01D test vos connaissances à propos des modificateurs d’accès à l’aide d’une mise en situation.

### L01E\_Exercices sur les Modificateurs d’accès

Le laboratoire L01E test vos connaissances à propos des modificateurs d’accès à l’aide d’exercices.

### TRAVAIL PRATIQUE 1

Le TP1 vous demande de mettre en pratique les éléments appris jusqu’à maintenant.