

Java : Partie 3

Programmation Orientée

Objets(POO)

Juin 2013

Auteur : Chouaïb LAGHLAM

Concepteur Développeur Informatique
Module : Programmation Java
Programmation Orientée Objets

P00 : Sommaire

Préalabe.....	4
Qu'est qu'un objet ?	4
Qu'est-ce qu'une classe ?	5
Attributs	5
Méthodes	6
Création d'un classe.....	6
Important pour une classe	7
Utilisation d'un objet	8
Déclaration d'un objet	8
Création d'un objet.....	8
Utilisation d'un objet	9
Encapsulation	9
La visibilité des attributs	13
Getteurs / Setteurs ou (Accesseurs / Mutateurs).....	14
La visibilité des méthodes	14
Constructeur (s).....	14
Déclarer un ou plusieurs constructeurs	17
Destructeur	18
Quand l'objet est il détruit ?	18
L'opérateur This.....	20
Membre statique.....	21
Membre ?	21
Membre Statique.....	21
Dans un code, je crée trois fruits :	22
Maintenant si je modifie le code qui crée les trois fruits :	23
Deux utilisations différentes des membres statiques.....	23
Déclaration d'un membre statique.....	23
Utilisation d'un membre statique.....	24
Tableaux d'objets ou Collections.....	24

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Agrégation ou Membre Objet	25
Utilisation d'un membre Objet.....	26
Exercices récapitulatifs	28
Héritage	28
Prenons un autre exemple et montrons comment fonctionne l'héritage en Java.....	29
Classe abstraite.....	30
Méthodes abstraites	30
Surcharge	34
Redéfinition	34
Héritage multiple.....	34
Héritage et protection	34
Polymorphisme	36
Polymorphisme de surcharge.....	36
Polymorphisme d'héritage.....	37
this / Super	37
Exercices récapitulatifs	37
Classification, regroupement et opérations associées	37
Finale (classe, attribut, méthode).....	37
Abstraire (classe, méthode)	38
Classe métier, classe technique.....	38
Classe générique : les collections	38
Interface (classe purement abstraite).....	38
Casting.....	38
Origine d'un objet.....	38
Objet dynamique	38
Bibliothèque, package.....	38

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Préalabe

Avant de commencer à lire cette 3^{ème} partie, vous êtes supposé avoir les connaissances de :

- ➔ La 1^{ère} partie qui permet d'installer l'environnement de développement : Eclipse,
- ➔ La 2^{ème} partie qui initie à la programmation de base en Java.
- ➔ Merci de créez dans votre Workspace «**Java_010_EspaceTravailCours**», le projet :
 - De type « **Java Project** »,
 - Nommez le «**prj_Java_020_POO**»,
 - Créez un package « **pack_essais**» sous le dossier **src**,
 - Créez un package « **pack_POO_Exos**» sous le dossier **src**,

Qu'est qu'un objet ?

Vous êtes supposé avoir suivi une journée d'introduction aux concepts objets en programmation.

Une application en Java n'est pas un grand fichier de centaines ou de milliers de lignes de code comme ça pouvait l'être dans les langages C ou Cobol.

Une application Java est décomposée en objets qui communiquent entre eux pour réaliser les fonctionnalités de cette application.

Prenons des exemples dans la vie courante :

- Une **voiture** est composée d'objets tels que «**Moteur**», «**Volant**», «**Roue**», «**Coffre**»,
Le volant communique avec la roue pour faire tourner la voiture, ...
- Une **machine à laver le linge** est composée d'objets tels que «**Tambour**», «**panneau de commande**», «**pompe**»,
Le panneau de commande communique avec le tambour pour le faire tourner ou pas

En informatique, c'est pareil, on apprend à décomposer une application en objets logiciels : actuellement la méthode UML permet d'apprendre à décomposer en objets.

Un objet contient toujours deux parties :

- Les **attributs ou propriétés** : ce sont les informations qui décrivent l'objet,
- Les **méthodes ou opérations** : ce sont les différents codes qui permettent de faire faire quelque chose à l'objet.

Reprenons l'exemple de la voiture :

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

L'objet « Moteur » contiendra :

- les attributs : n° de série, poids, date fabrication, ...
- les méthodes : démarrer, arrêter,

Un objet existe pendant l'exécution de l'application.

Qu'est-ce qu'une classe ?

Avant de pouvoir utiliser un objet logiciel, faut-il encore savoir comment le créer.

Imaginez que vous et moi possédons la même voiture : même marque, même modèle, même année,

Ce sont deux objets fabriqués de la même façon mais comment ?

Les ingénieurs de la marque ont conçus des plans, des procédures pour fabriquer en plusieurs exemplaires cette voiture.

Eh ben, une classe est :

- ➔ le plan pour fabriquer des objets,
- ➔ ou le schéma de fabrication des objets,
- ➔ ou le modèle de création des objets,

Généralement dans une équipe informatique :

- ➔ un chef de projet conçoit la décomposition de l'application en classes,
- ➔ une personne de l'équipe, crée ces classes en programmation : ici en Java,
- ➔ les autres développeurs, utilisent les classes pour créer des objets et ainsi fabriquer chacun une partie de l'application.

Le langage Java est fourni avec des centaines de classes (donc des objets à créer) qui facilitent la programmation :

- ➔ faciliter la saisie et l'affichage des données,
- ➔ faciliter l'accès aux Bases de données,
- ➔ faciliter l'accès au réseau, à l'internet,
- ➔ ...

A chaque fois que l'on crée une nouvelle classe : on dit aussi que l'on a créé un nouveau Type.

Attributs

Je l'ai dit plus haut, un attribut ou propriété est une information qui décrit un objet.

Pour déclarer un attribut, il faut :

- Donner la **visibilité** de l'attribut (voir plus loin),
- Donner son **type** : que peut-on stocker dedans ? un nombre entier, un décimal, une chaîne de caractères, Une date, ...
- Eventuellement une longueur.

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Méthodes

Nous avons vu dans la partie « les bases de Java », qu'une méthode est un code Java que l'on appelle à partir d'une autre méthode pour que ce code réalise un traitement (un calcul, un affichage, une action, ...) :

Lorsqu'une méthode réalise un traitement général : on dit que c'est une **méthode statique**.

La nouveauté ici est que la méthode va être attachée à un type d'objets donné et elle réalise quelque chose sur l'objet lui-même. On dit que c'est une méthode d'objet.

Toujours dans l'exemple de la voiture :

- Compter le nombre de voitures vendues à un instant donné : une méthode statique,
- Tourner à gauche : une méthode d'objet.

Création d'une classe

Dans Eclipse, une future application Java est :

- ➔ Un **projet** d'un type donné (c'est sous dossier du workspace sous Windows),
- ➔ Ce projet contient un dossier nommé «**src**» (c'est un sous dossier du projet sous Windows),
- ➔ Le dossier « src » doit contenir un ou plusieurs **packages** (des sous-dossiers de src sous Windows),
- ➔ Un package contiendra des classes Java (des fichiers avec l'extension .java)

Exemple de la création d'une classe nommée «Tennisman01» décrivant un futur objet dans un jeu vidéo de Tennis :

- ➔ Les attributs : nom, prénom, taille du tennisman et une info pour savoir s'il est ou pas dans les 10 premiers du classement mondial,
- ➔ Les méthodes qui font faire quelque chose au futur objet : créer, modifier, consulter ou supprimer un tennisman.

```
package pack_essais;

/*
=====
Classe Tennisman
avec attributs publiques
=====
Auteur      : Chouaïb LAGHLAM
Date       : Aout 2013
Modifié le  :
Modifié par :
=====
*/

public class Tennisman01 {
    // attributs ou propriétés
    public String nom;
    public String prenom;
    public int taille;           // en cm
    public boolean istopTen;
    // méthode pour créer un tennisman
    public void creer()
```


Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

- ➔ On exécute une méthode mais jamais une classe,
- ➔ Une méthode qui n'est pas la méthode « main » ne peut d'exécuter qu'en l'appelant d'une autre méthode,
- ➔ une application java doit contenir une et une seule classe avec la méthode «main» : c'est le code qui s'exécute en premier lorsqu'on lance l'application.

Utilisation d'un objet

Un objet existe pendant l'exécution de l'application et il est détruit pendant l'exécution ou à la fin de l'application.

Si on souhaite garder les informations sur un objet pour le réutiliser au prochain lancement de l'application :

- ➔ il faut stocker les infos de l'objet sur le disque (fichier ou bases de données ou),
- ➔ l'objet se détruit avant la fin ou à la fin de l'application,
- ➔ au prochain lancement de l'application, un nouvel objet est créé et il faut lui affecter les infos stockées sur le disque pour qu'il ressemble à l'objet de dernier lancement.

Déclaration d'un objet

Je déclare un objet comme de la même façon que la déclaration d'une variable simple : il suffit de préciser son type, donc la classe concernée.

Exemples :

```
public Tennisman01    t ;  
.....  
public Raquetterq ;
```

Ici «Tennisman01» et «Raquette» sont des classes (on dit aussi des Types), «t» et «rq» sont des noms des futurs Objets à créer.

Déclarer un objet ne le crée pas : donc on ne peut pas commencer à l'utiliser : lire ou modifier ses attributs, appeler ses méthodes.

Lorsque je déclare un objet, le type (la classe) de l'objet doit être localisé (e) : c'est-à-dire que Java doit savoir où se trouve le fichier de la Classe qui déclare le type :

- ➔ Soit la classe se trouve où même endroit (même package) que le code qui déclare l'objet : rien à faire,
- ➔ Soit la classe se trouve dans un autre package : il faut préciser tout en haut du code qui déclare l'objet, une ligne «**import nomPackage.nomClasse**»

Création d'un objet

Deux façons de créer un objet en Java :

- 1) Créer un objet tout neuf en utilisant l'opérateur **new** :

Exemple 1 : je déclare l'objet puis plus loin je le créé.

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

```
public Tennisman01 t ;  
// ....  
t=new Tennisman01() ;
```

Exemple 2 : je déclare et je crée l'objet en même temps.

```
public Tennisman01 t =new Tennisman01() ;
```

- 2) J'appelle une méthode qui renvoie un objet du même type que celui que je cherche à créer (donc pas de new) :

Exemple :

```
// ....
```

```
Facture f=calculerFacture (1234,'12/02/2013') ;
```

Ci-dessus : je déclare un objet f de type Facture. L'appel de la méthode «calculerFacture» renvoie un objet de type Facture et le stocke dans l'objet f.

Utilisation d'un objet

Une fois un objet créé, l'utiliser revient à affecter (valoriser) ou lire ses attributs, appeler ses méthodes.

Il suffit d'utiliser un point entre l'objet et son attribut ou sa méthode.

Exemple :

```
// .....
```

```
Tennisman01 t=new Tennisman01() ;
```

```
// .....
```

```
t.nom="FEDERER" ;
```

```
t.prenom="Roger" ;
```

```
// ...
```

```
t.creer() ;
```

```
// ...
```

Encapsulation

Lorsque quelqu'un crée une classe d'objets, plusieurs personnes peuvent l'utiliser (membres de l'équipe du développement, mais aussi des personnes qui l'ont téléchargé du Net).

Le danger alors est qu'un développeur puisse mettre des valeurs interdites ou incorrectes dans les attributs (les propriétés) d'un objet. Par exemple : quelqu'un déclare un objet Tennisman01 comme vu ci-dessus puis affecte au

nom du tennisman la valeur "%% !Nadal" ou donne une **taille=3 mètres !!!!!**

Pour éviter ce genre de mésaventure, le créateur de la classe, va protéger les attributs par une procédure nommée l'encapsulation ;

- ➔ Il utilise le mot « private » au moment de déclarer chaque attribut : cela empêche un utilisateur de la classe d'affecter directement l'attribut

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Par exemple :

```
// .....  
private String nom ;  
// .....  
Tennisman01 tn=new Tennisman01 () ;  
// ...  
tn.nom='Nadal';           cette ligne sera refusé car l'attribut est privée
```

➔ Il créera, dans la classe, deux méthodes :

- Une pour permettre d'affecter (modifier) l'attribut : on appelle cette méthode «**un setteur**»,
- Une autre pour permettre la lecture de la valeur de l'attribut : on appelle cette méthode «**un getteur**»
- Dans chacune de ces méthodes, il codera les règles pour accepter ou pas la valeur à donner à un attribut Et le règles pour formater une valeur avant de la restituer

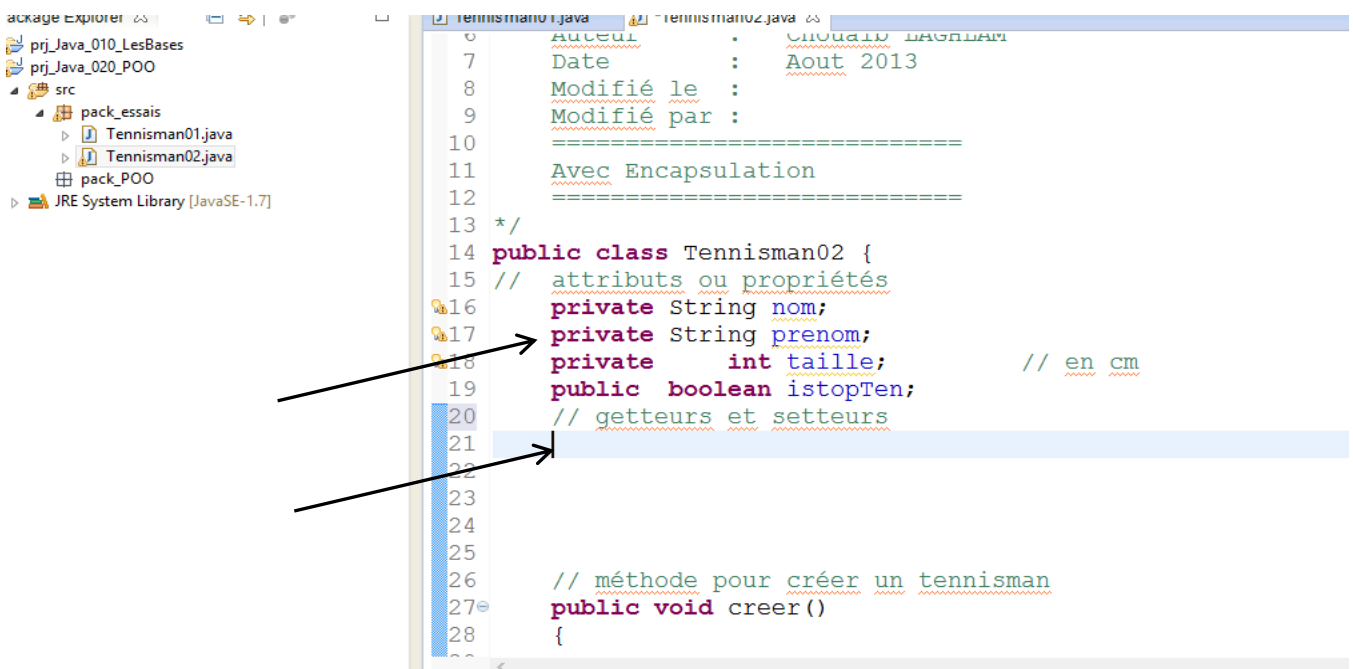
Exemple :

Nous allons reprendre la classe Tennisman01 pour la renommer **Tennisman02** :

Les attributs seront déclarés en private (donc encapsulés),

Pour la taille du tennisman : on refusera toute valeur qui ne sera pas comprise entre 1,55m et 2,20m,

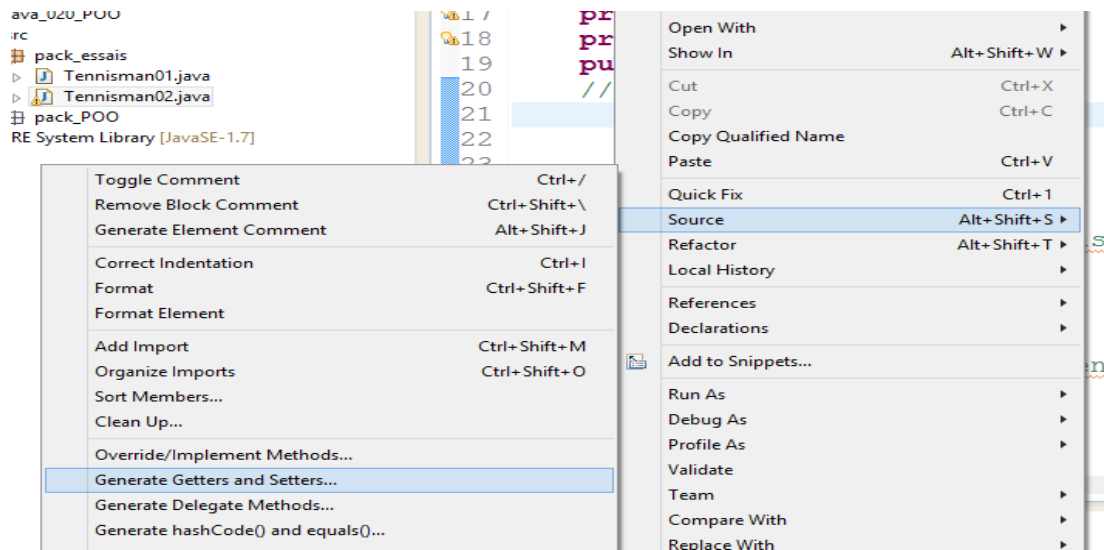
Pour le nom du tennisman : il sera mis en majuscules avant de le donner en lecture.



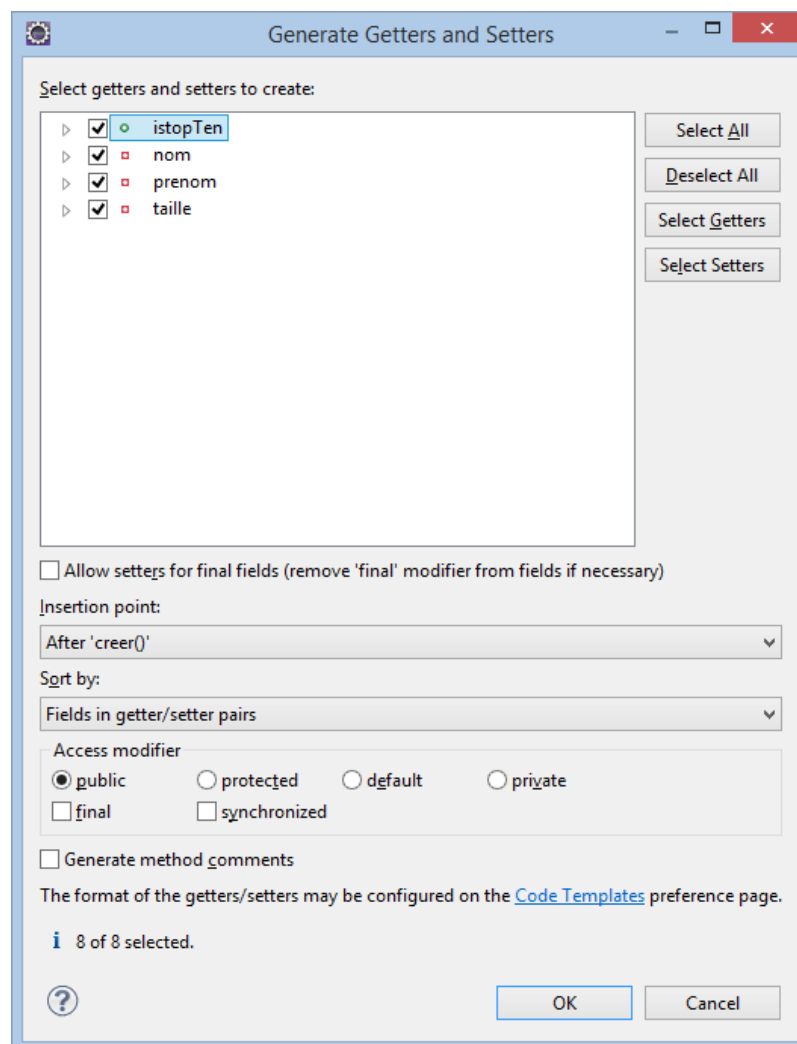
Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets



Je coche tous les attributs :

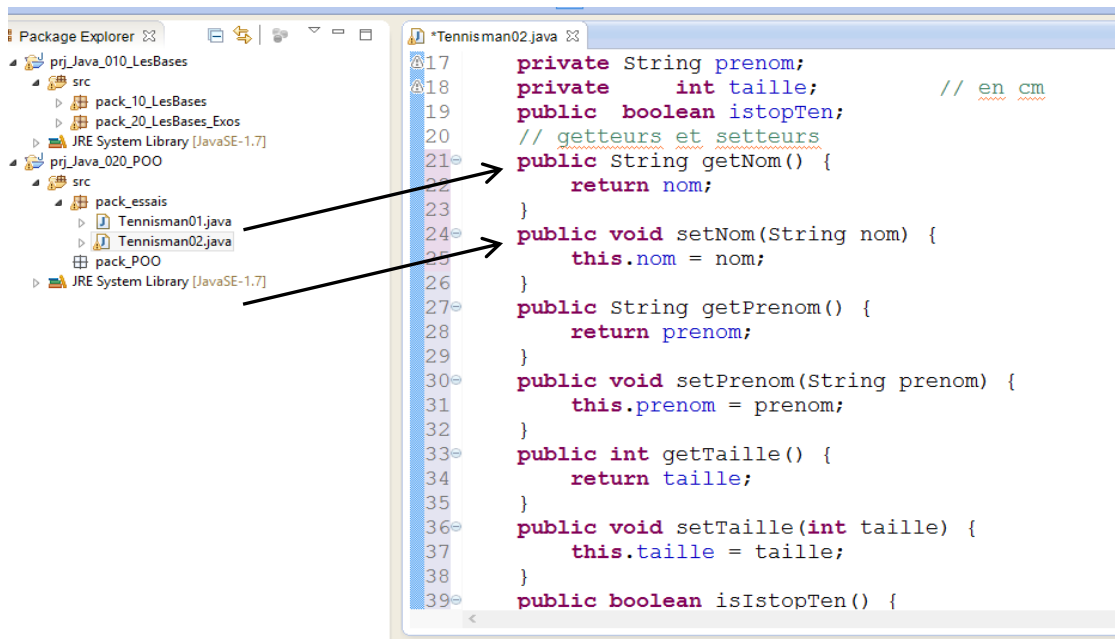


Concepteur Développeur Informatique

Module : Programmation Java

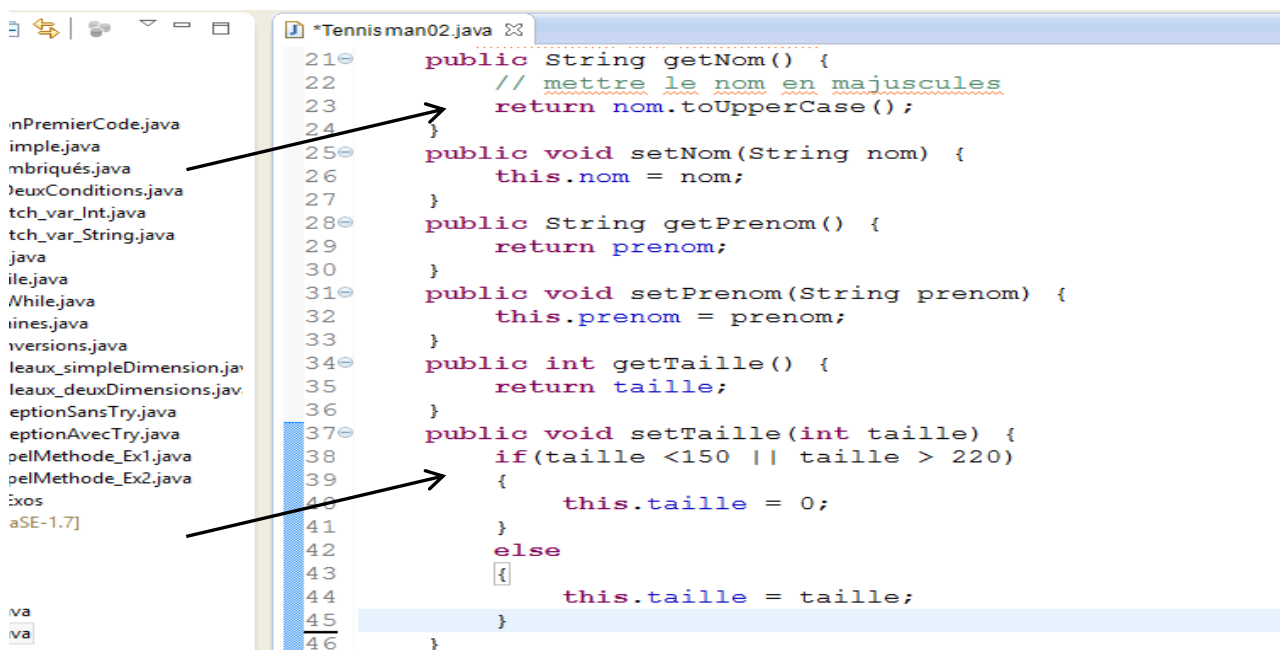
Programmation Orientée Objets

Eclipse génère du code supplémentaire dans ma classe :



Eclipse a créé une méthode «**getNom**» (le getteur) pour récupérer la valeur de l'attribut «**nom**» et la méthode «**setNom**» (le setteur) pour modifier la valeur de l'attribut «**nom**»,
Donc pour chaque attribut un couple de méthodes : un getteur et un setteur.

Voici les méthodes **getNom** et **setTaille** complétées :

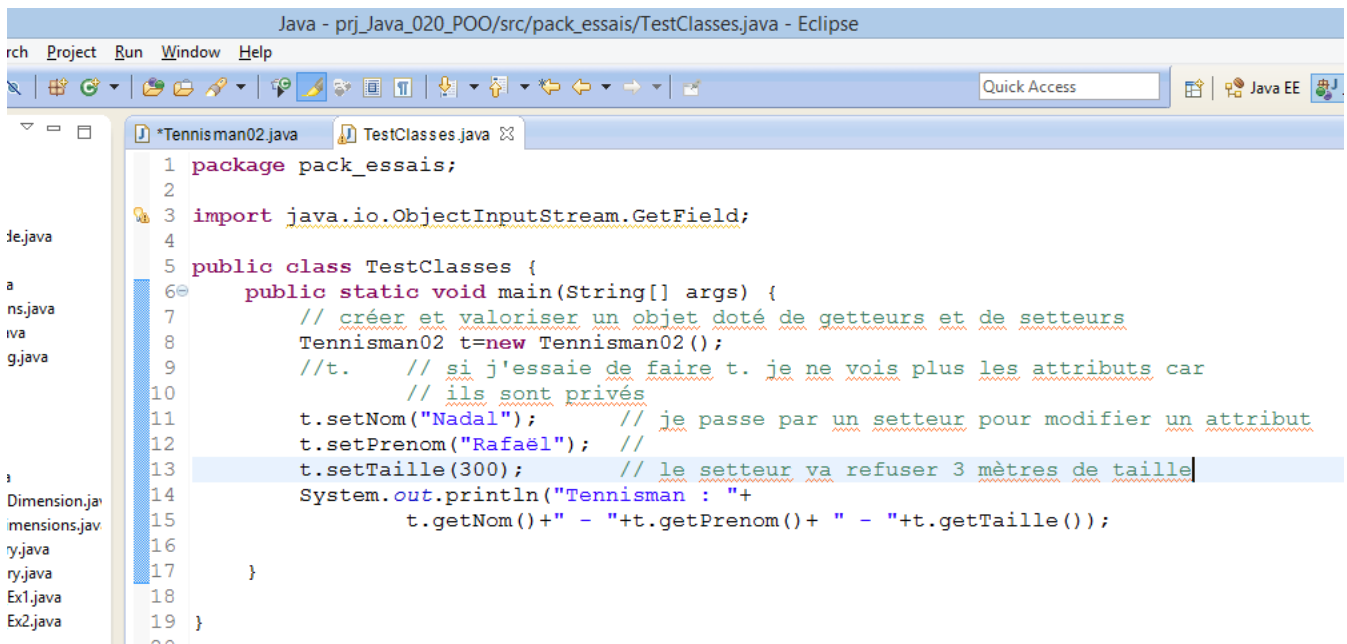


Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

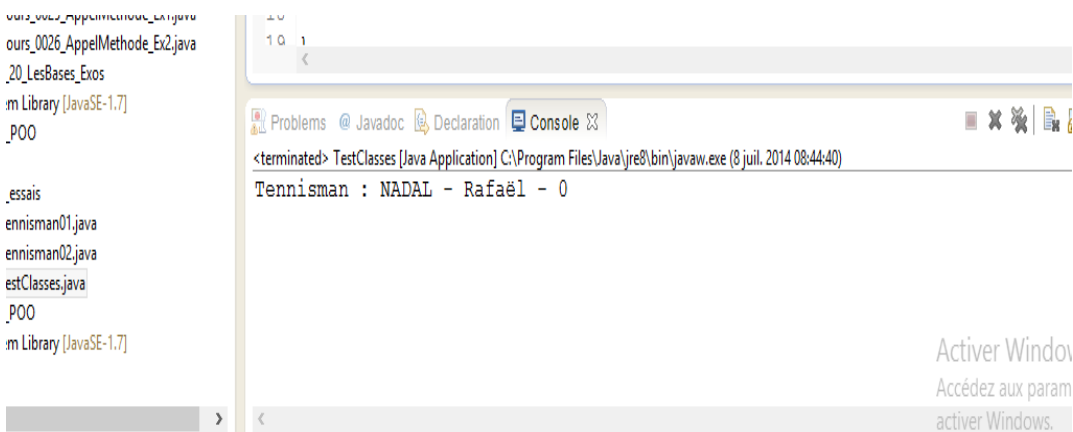
Maintenant si dans un code, je veux utiliser un objet Tennisman02, je procède ainsi :



```
Java - prj_Java_020_POO/src/pack_essais/TestClasses.java - Eclipse
rch Project Run Window Help
Quick Access
Java EE

*Tennisman02.java TestClasses.java
1 package pack_essais;
2
3 import java.io.ObjectInputStream.GetField;
4
5 public class TestClasses {
6     public static void main(String[] args) {
7         // créer et valoriser un objet doté de getteurs et de setteurs
8         Tennisman02 t=new Tennisman02 ();
9         //t. // si j'essaie de faire t. je ne vois plus les attributs car
10        // ils sont privés
11        t.setNom("Nadal"); // je passe par un setteur pour modifier un attribut
12        t.setPrenom("Rafaël"); //
13        t.setTaille(300); // le setteur va refuser 3 mètres de taille
14        System.out.println("Tennisman : "+
15            t.getNom()+" - "+t.getPrenom()+" - "+t.getTaille());
16    }
17
18
19 }
```

A l'exécution, le code ci-dessus donnera :



```

ours_0026_AppelMethode_Ex2.java
_20_LesBases_Exos
m Library [JavaSE-1.7]
_POO
_essais
ennisman01.java
ennisman02.java
estClasses.java
_POO
m Library [JavaSE-1.7]

10 1
<terminated> TestClasses [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (8 juil. 2014 08:44:40)
Tennisman : NADAL - Rafaël - 0

Activer Window
Accédez aux param
activer Windows.
```

Remarquez le nom en majuscule et la taille = 0

La visibilité des attributs

Finalement l'encapsulation se résume à rendre les attributs cachés et créer des méthodes spécialisées qui contrôlent leur modification et leur lecture.

Techniquement, on appelle cela aussi : la visibilité des attributs.

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Un attribut O peut être déclaré :

- ➔ **public** : le développeur peut accéder à l'attribut directement ce qui veut dedans :
O.attribut=..... ;
- ➔ **private** : le développeur doit appeler une méthode particulière (un setteur) pour modifier l'attribut et une autre méthode (un getteur) pour lire la valeur de l'attribut
O.setAttribut=..... ;

Ou **x=O.getAttribut() ;**
- ➔ **protected** : rarement utilisé mais possible, cette déclaration permet :
 - à un code appartenant à une classe fille de la classe contenant l'attribut (voir héritage plus loin) d'accéder directement à la valeur de l'attribut comme s'il était déclaré en public,
 - d'empêcher un code n'appartenant pas à une classe fille de la classe contenant l'attribut d'accéder directement à la valeur de l'attribut comme s'il était déclaré en private,
 -

Getteurs / Setteurs ou (Accesseurs / Mutateurs)

En français, nous avons tenté de nommer les getteurs «**Accesseurs**» et les setteurs «**Mutateurs**».

Mais vous ne trouverez pas beaucoup d'équipes informatiques qui utilisent ces termes.

La visibilité des méthodes

Par défaut les méthodes sont publiques. Donc très souvent, vous aurez le mot « **public** » à gauche du nom d'une méthode.

Parfois, dans une classe, une méthode est uniquement utilisée par les autres méthodes de la même classe qu'elle. Dans ce cas elle est déclarée avec le mot «**private**» : impossible de l'appeler par un code extérieur à la classe.

Si vous souhaitez qu'une méthode soit appelée par les méthodes de la même classe qu'elle + les méthodes des classes filles (qui héritent de la classe de cette méthode), il faut utiliser le terme « **protected** »,

Constructeur (s)

Un développeur peut créer un objet à partir d'une classe mais sans jamais valoriser les attributs ou ne renseigner que certains.

Par exemple :

Créer un objet à partir de la classe `Tennisman02` mais oublier de renseigner le nom, le prénom, la taille,

Ou renseigner que le nom et la taille mais pas le prénom.

Si le créateur de la classe veut obliger le développeur à renseigner certains ou tous les attributs au moment de la création de l'objet (quand on fait `new`), il recourt à la notion du constructeur.

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Un constructeur est une méthode de la classe qui porte le même nom que la classe,

Exemple 1 :

```
Tennisman02 t=new Tennisman02 ();
```

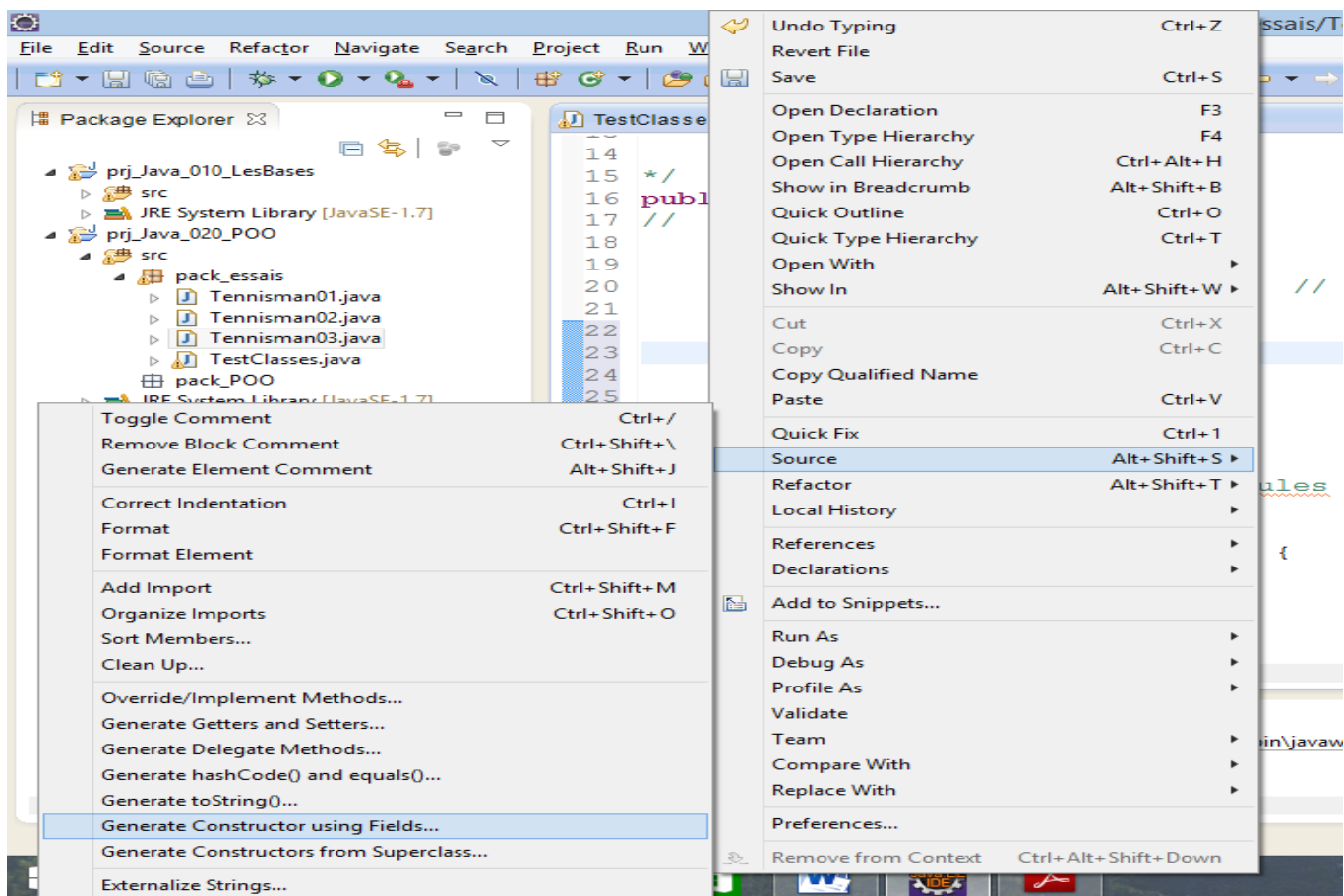
Création de l'objet t à partir de la classe Tennisman02

appel constructeur sans paramètres :
Le constructeur n'existe pas

Exemple 2 :

Création d'une classe **Tennisman03** avec constructeur exigeant le nom et le prénom du joueur :

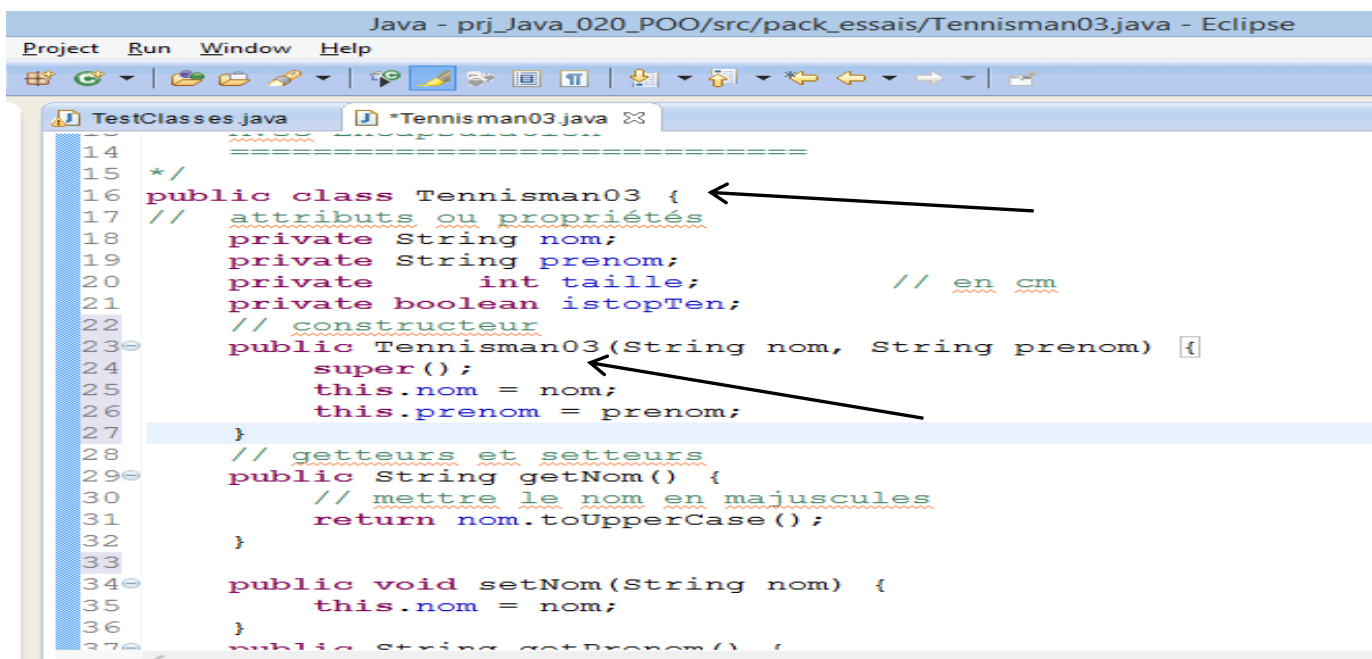
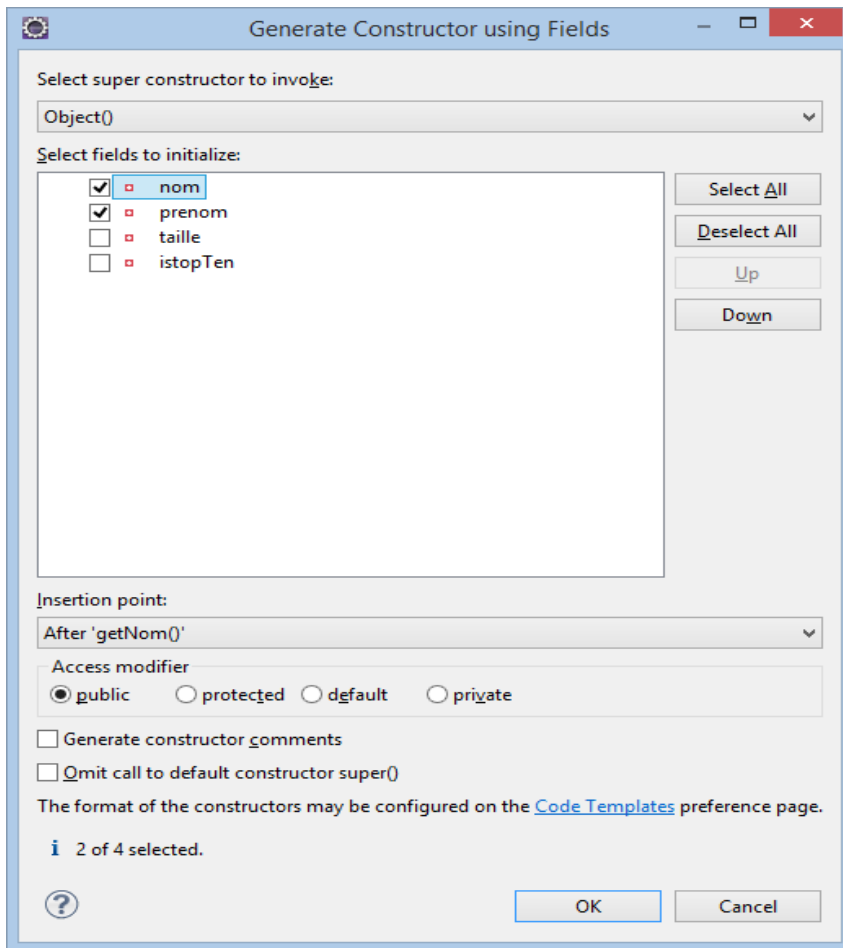
Je crée la classe avec les attributs puis je demande à Eclipse de me générer le code du constructeur :



Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets



Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Dans un code donné, je crée un objet tennisman à partir de la classe **Tennisman03** :

```
24      //
25      // créer et valoriser un objet dont la classe est dotée de constructeur
26      //
27      // Tennisman03 tr=new Tennisman03(); // ligne refusée car le constructeur
28      // exige le nom et le prénom
29      Tennisman03 trn=new Tennisman03("Tsonga","Wilfrid"); // ligne acceptée
30  }
31
```

Déclarer un ou plusieurs constructeurs

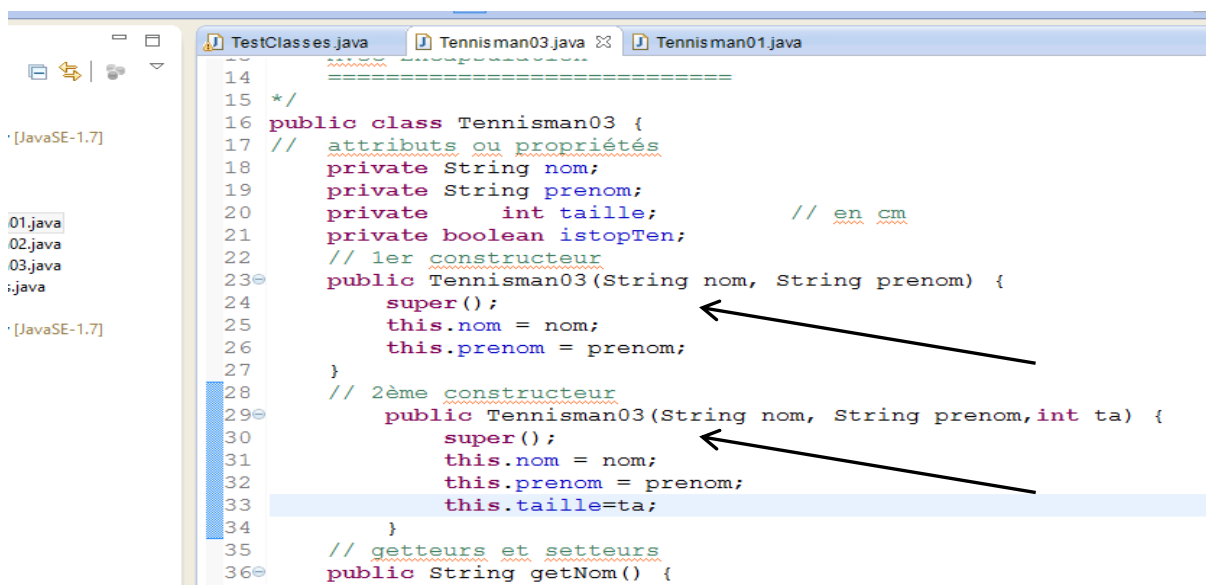
Dans la classe Tennisman03 ci-dessus, à la création de l'objet on ne peut fournir que le nom et le prénom.

Mais si le développeur connaît, au moment de la création de l'objet, le nom, le prénom et la taille aussi ?

Java permet de déclarer plusieurs constructeurs (donc plusieurs méthodes qui portent le nom de la classe), mais qui reçoivent des attributs différents.

Exemple 3 :

- Un constructeur permettant de donner le nom et le prénom à la création,
- Un 2^{ème} constructeur permettant de donner le nom, le prénom et la taille à la création :



```
14      //
15      */
16  public class Tennisman03 {
17      // attributs ou propriétés
18      private String nom;
19      private String prenom;
20      private int taille; // en cm
21      private boolean istopTen;
22      // 1er constructeur
23      public Tennisman03(String nom, String prenom) {
24          super();
25          this.nom = nom;
26          this.prenom = prenom;
27      }
28      // 2ème constructeur
29      public Tennisman03(String nom, String prenom, int ta) {
30          super();
31          this.nom = nom;
32          this.prenom = prenom;
33          this.taille=ta;
34      }
35      // getteurs et setteurs
36      public String getNom() {
```

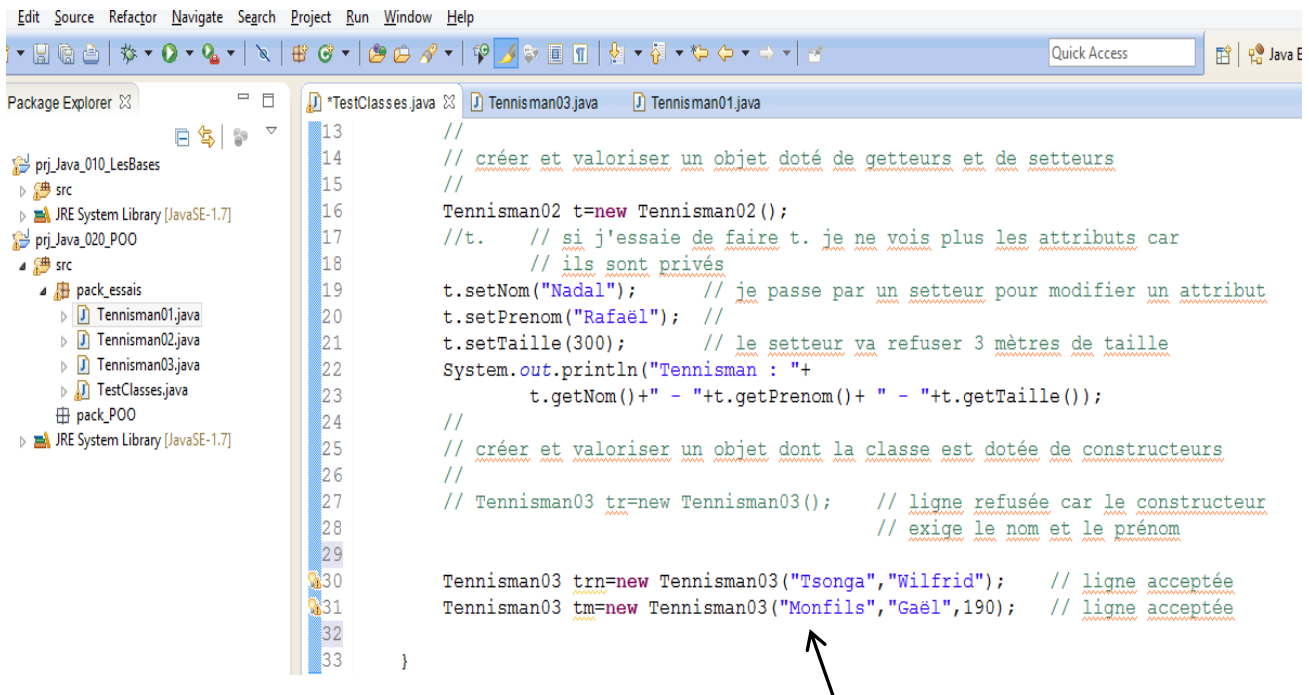
Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Dans un code donné, je crée deux objets tennisman à partir de la classe **Tennisman03** :

- Le 1^{er} en utilisant un constructeur,
- Le 2^{ème} en utilisant un autre constructeur.



```
13 //
14 // créer et valoriser un objet doté de getteurs et de setteurs
15 //
16 Tennisman02 t=new Tennisman02();
17 //t. // si j'essaie de faire t. je ne vois plus les attributs car
18 // ils sont privés
19 t.setNom("Nadal"); // je passe par un setteur pour modifier un attribut
20 t.setPrenom("Rafael"); //
21 t.setTaille(300); // le setteur va refuser 3 mètres de taille
22 System.out.println("Tennisman : "+
23 t.getNom()+" - "+t.getPrenom()+" - "+t.getTaille());
24 //
25 // créer et valoriser un objet dont la classe est dotée de constructeurs
26 //
27 // Tennisman03 tr=new Tennisman03(); // ligne refusée car le constructeur
28 // exige le nom et le prénom
29
30 Tennisman03 trn=new Tennisman03("Tsonga", "Wilfrid"); // ligne acceptée
31 Tennisman03 tm=new Tennisman03("Monfils", "Gaël", 190); // ligne acceptée
32
33 }
```

Destructeur

Un objet créé et utilisé dans une méthode finit par se détruire (disparaître de la mémoire) à un instant t.

S'il y a un traitement à faire lorsque l'objet disparaît : il faut alors créer une méthode qui sera **appelée automatiquement** dès la destruction de l'objet.

Cette méthode s'appelle le «destructeur» et porte un nom particulier «**finalize**»

Par exemple :

Dans le fameux jeu «Pacman», chaque fois que pacman mange une pastille (un objet disparaît), l'écran doit effacer la trace de la pastille : idéalement le code cet effacement sera logé dans le destructeur.

Quand l'objet est-il détruit ?

Dans le langage **C++** (plus ancien que le Java), les développeurs rencontrent des problèmes liés à la saturation de la mémoire :

- ➔ A chaque fois qu'ils créent un objet, ils doivent penser à sa suppression quand il devient inutile pour ne pas encombrer la mémoire : c'est une gestion lourde et une perte de temps.

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

En java, on procède différemment : il y a, pendant l'exécution d'une application Java, un processus automatique nommé «**Garbage Collector**» ou «**ramasse miettes**» en français et qui a pour rôle :

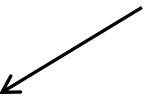
- ➔ Supprimer automatiquement un objet de l'application dès qu'il devienne inaccessible (par exemple : on appelle une méthode qui crée un objet : lorsque le code de cette méthode se termine, l'objet créé n'est plus accessible par les autres méthodes et le Garbage Collector (GC) le détruit dès que possible),
- ➔ Le développeur Java ne s'occupe pas à détruire les objets, ce n'est pas son souci. Il met simplement en place des destructeurs s'il a besoin d'exécuter un traitement quand les objets disparaissent.

Exemple :

Je crée une classe **Tennisman04** et je la dote d'un destructeur,

Dans un code donné, je crée un objet de type Tennisman04 et je force le GC à supprimer immédiatement l'objet :

```
77     }
78
79     // méthode pour modifier un tennisman
80     public void consulter()
81     {
82
83     }
84     // méthode pour modifier un tennisman
85     public void supprimer()
86     {
87
88     }
89     // destructeur
90     public void finalize(){
91         System.out.println("Objet de type Tennisman04 supprimé");
92     }
93
94
95 }
96
```



Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Dans un code donné :

```
Java - prj_Java_020_POO/src/pack_essais/TestClasses.java - Eclipse
Refactor Navigate Search Project Run Window Help
Quick Access

*TestClasses.java
Tennisman04.java

26 //
27 // Tennisman03 tr=new Tennisman03(); // ligne refusée car le constructeur
28 // exige le nom et le prénom
29
30 Tennisman03 trn=new Tennisman03("Tsonga","Wilfrid"); // ligne acceptée
31 Tennisman03 tm=new Tennisman03("Monfils","Gaël",190); // ligne acceptée
32 //
33 // créer un objet et le détruire pour voir le code du destructeur
34 //
35 creerTennisman();
36 System.gc();
37 System.out.println("Fin du programme");
38 }
39 //
40 public static void creerTennisman(){
41     Tennisman04 t3=new Tennisman04("Cornet", "Alizé");
42 }
43 }
44

Problems Javadoc Declaration Console
<terminated> TestClasses [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (8 juil. 2014 14:45:32)
Fin du programme
Objet de type Tennisman04 supprimé
```

Remarque : si vous n'arrivez pas à voir le message du destructeur, ajoutez une instruction après « System.gc() » qui permet de faire une pause et laisser au GC le temps de supprimer l'objet : voir la classe « Thread » pour faire cette pause ;

L'opérateur This

Le mot «**this**» que vous allez rencontrer souvent en java signifie «**l'objet courant**» ou «**l'objet concerné**»,

Par exemple :

Un développeur crée un objet nommé «t» à partir de la classe Tennisman03 ainsi :

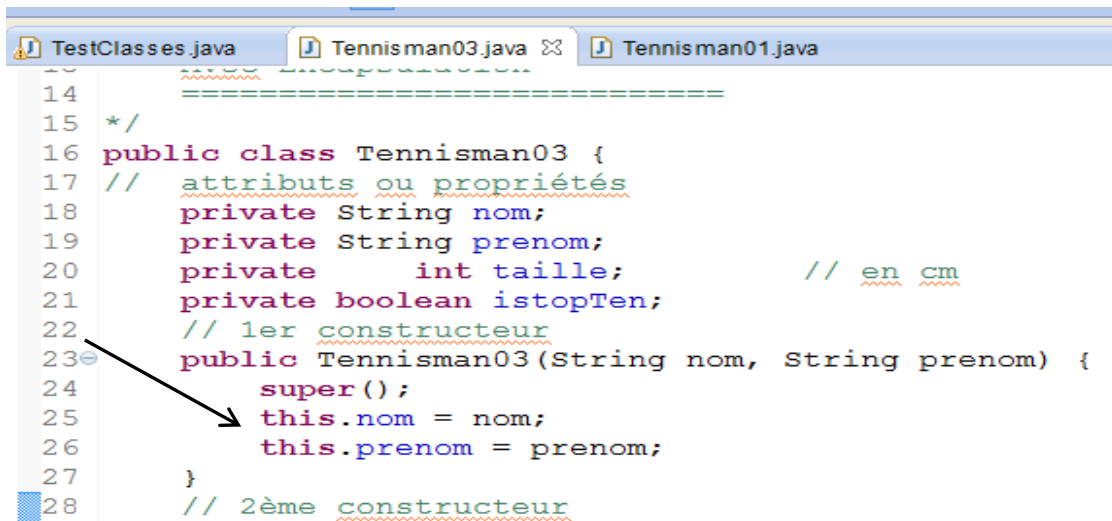
Tennisman03 t=new Tennisman03 ("Djokovic","Novac");

Le constructeur de la classe Tennisman03 est appelé et il utilise le «this» :

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets



```
14  =====
15  */
16  public class Tennisman03 {
17  // attributs ou propriétés
18  private String nom;
19  private String prenom;
20  private int taille;           // en cm
21  private boolean istopTen;
22  // 1er constructeur
23  public Tennisman03(String nom, String prenom) {
24      super();
25      this.nom = nom;
26      this.prenom = prenom;
27  }
28  // 2ème constructeur
```

En utilisant le «this», la méthode constructeur veut dire «je modifie le nom de l'objet courant» (elle ne sait pas que le développeur l'appelle «t») par la valeur passée en paramètre (à savoir «Djokovic»).

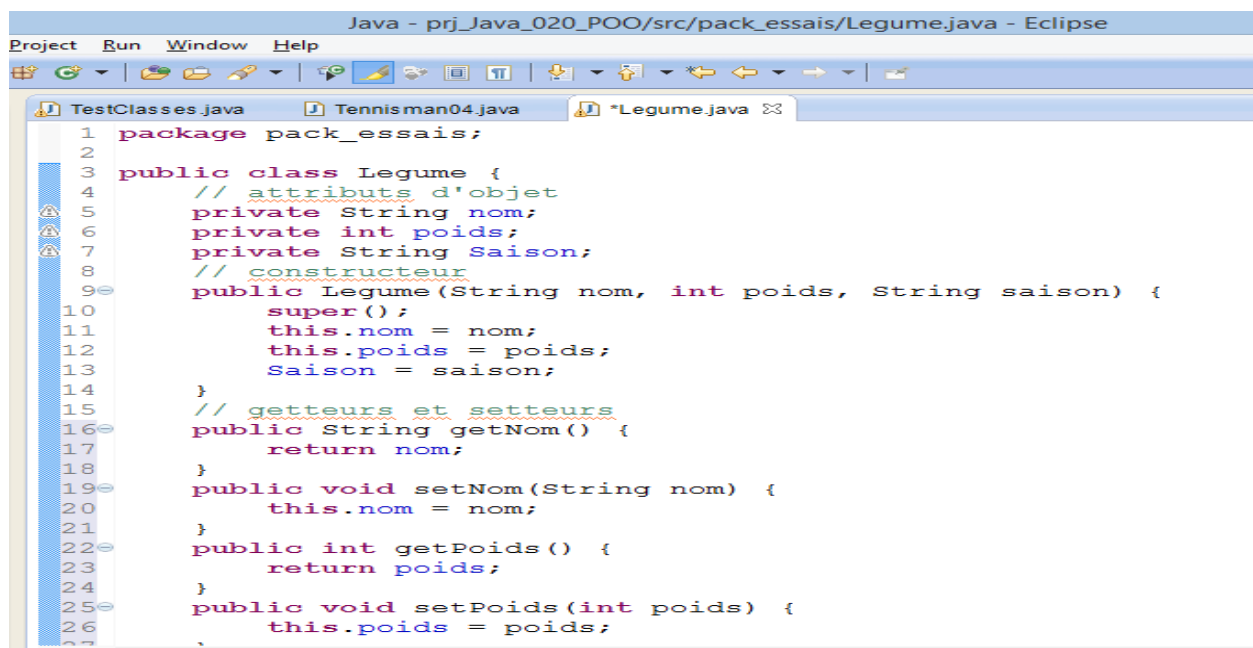
Membre statique

Membre ?

Le mot «Membre» désigne soit l'attribut d'un objet soit la méthode d'un objet. C'est pratique de l'utiliser pour ne pas répéter «attribut ou méthode» à chaque fois.

Membre Statique

Prenons une nouvelle classe nommée «**Legume**» et expliquons avec cette classe la notion de statique,



```
1 package pack_essais;
2
3 public class Legume {
4     // attributs d'objet
5     private String nom;
6     private int poids;
7     private String saison;
8     // constructeur
9     public Legume(String nom, int poids, String saison) {
10         super();
11         this.nom = nom;
12         this.poids = poids;
13         saison = saison;
14     }
15     // getteurs et setteurs
16     public String getNom() {
17         return nom;
18     }
19     public void setNom(String nom) {
20         this.nom = nom;
21     }
22     public int getPoids() {
23         return poids;
24     }
25     public void setPoids(int poids) {
26         this.poids = poids;
27     }
28 }
```

Concepteur Développeur Informatique

Module : Programmation Java

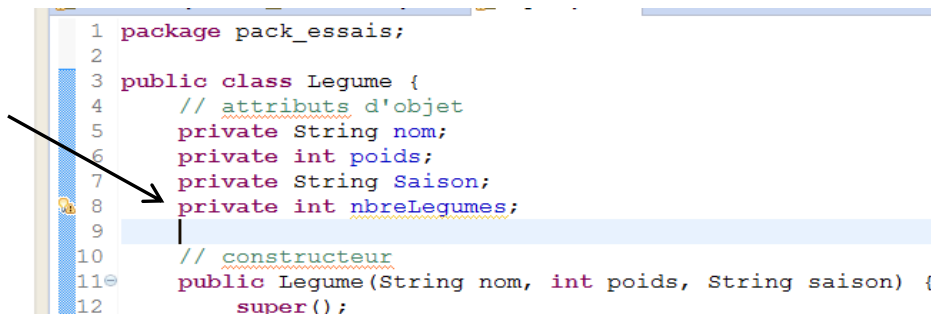
Programmation Orientée Objets

Dans un code, je créé trois fruits :

```
Legume l1=new Legume("Carottes",50,"Tout");  
Legume l2=new Legume ("Artichaut",110,"Juin");  
Legume l3=new Legume ("Chou Blanc",300,"Janvier");
```

J'aimerais qu'à chaque création de légume, je comptabilise le nombre de légumes créés.

Je ne peux ajouter dans la classe, l'attribut suivant :

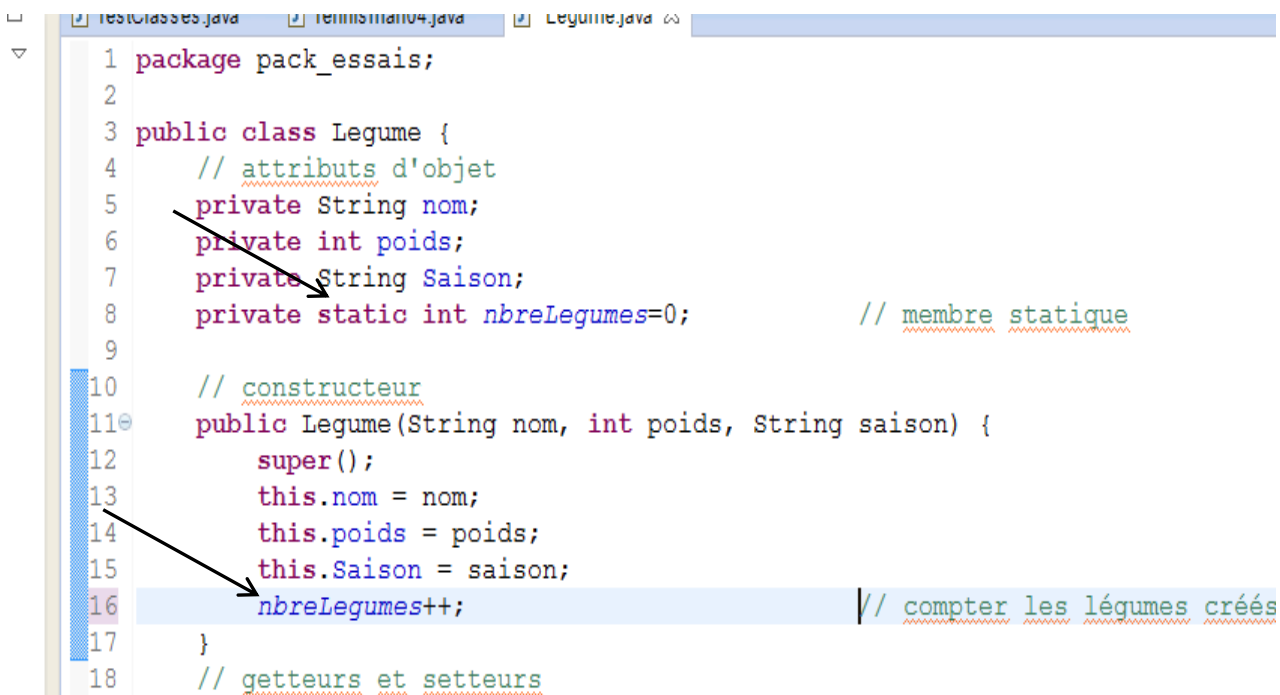


```
1 package pack_essais;  
2  
3 public class Legume {  
4     // attributs d'objet  
5     private String nom;  
6     private int poids;  
7     private String Saison;  
8     private int nbreLegumes;  
9  
10    // constructeur  
11    public Legume(String nom, int poids, String saison) {  
12        super();
```

Car java va créer un attribut pour chaque légume créé. Or je que je veux : c'est un seul attribut pour tous les légumes créés et qui stocke dans mon exemple « 3 » pour trois légumes créés.

Pour cela, je modifie mon code pour :

- Ajouter le mot **static** dans la déclaration de l'attribut «**nbreLegumes**»,
- J'incréméte cet attribut dans le constructeur,
- Créer un getteur qui donne le nombre de légumes créés.



```
1 package pack_essais;  
2  
3 public class Legume {  
4     // attributs d'objet  
5     private String nom;  
6     private int poids;  
7     private String Saison;  
8     private static int nbreLegumes=0;           // membre statique  
9  
10    // constructeur  
11    public Legume(String nom, int poids, String saison) {  
12        super();  
13        this.nom = nom;  
14        this.poids = poids;  
15        this.Saison = saison;  
16        nbreLegumes++;                          // compter les légumes créés  
17    }  
18    // getteurs et setteurs
```

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

```
34 public void setSaison(String saison) {  
35     Saison = saison;  
36 }  
37 public static int getNbreLegumes() {  
38     return nbreLegumes;  
39 }  
40  
41 }  
42
```

Maintenant si je modifie le code qui crée les trois fruits :

```
40 //  
41 // a chaque fois que je crée un légume : le constructeur comptabilise le nbre de  
42 // légumes dans un attribut statique : donc variable globale ou attribut de classe  
43 Legume l1=new Legume("Carottes",50,"Tout");  
44 Legume l2=new Legume("Artichaut",110,"Juin");  
45 Legume l3=new Legume("Chou Blanc",300,"Janvier");  
46 // afficher nbre de légumes créés  
47 System.out.println("Nombre de légumes créés : "+ Legume.getNbreLegumes());  
48 }  
49 //  
50 public static void creerTennisman(){  
51     Tennisman04 t3=new Tennisman04("Cornet", "Alizé");  
--
```

Problems @ Javadoc Declaration Console
<terminated> TestClasses [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (9 juil. 2014 09:06:19)
Nombre de légumes créés : 3

Deux utilisations différentes des membres statiques

On a recours, de deux façons différentes, aux membres statiques :

- Création de classes techniques utilitaires avec des variables globales, des constantes et des fonctions générales
- Ou, comme dans l'exemple précédent, on a besoin d'infos sur plusieurs objets ou traiter plusieurs objets : on utilise alors des attributs statiques et des méthodes statiques.

Déclaration d'un membre statique

Pour rendre un attribut ou une méthode statique, on ajoute à la déclaration le mot «**static**»,

On dit que les membres non statiques sont «**des attributs et des méthodes d'objets**»,

Que les membres statiques sont «**des attributs et des méthodes de classes**»

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Utilisation d'un membre statique

Quand je veux accéder à un attribut ou à une méthode non statique :

J'écris : **nomObjet.nomMembre**

Quand je veux accéder à un attribut ou à une méthode statique :

J'écris : **nomClasse.nomMembre**

Tableaux d'objets ou Collections

Nous avons vu précédemment comment on déclare un tableau contenant des données primitives (entier ou décimal ou chaîne,)

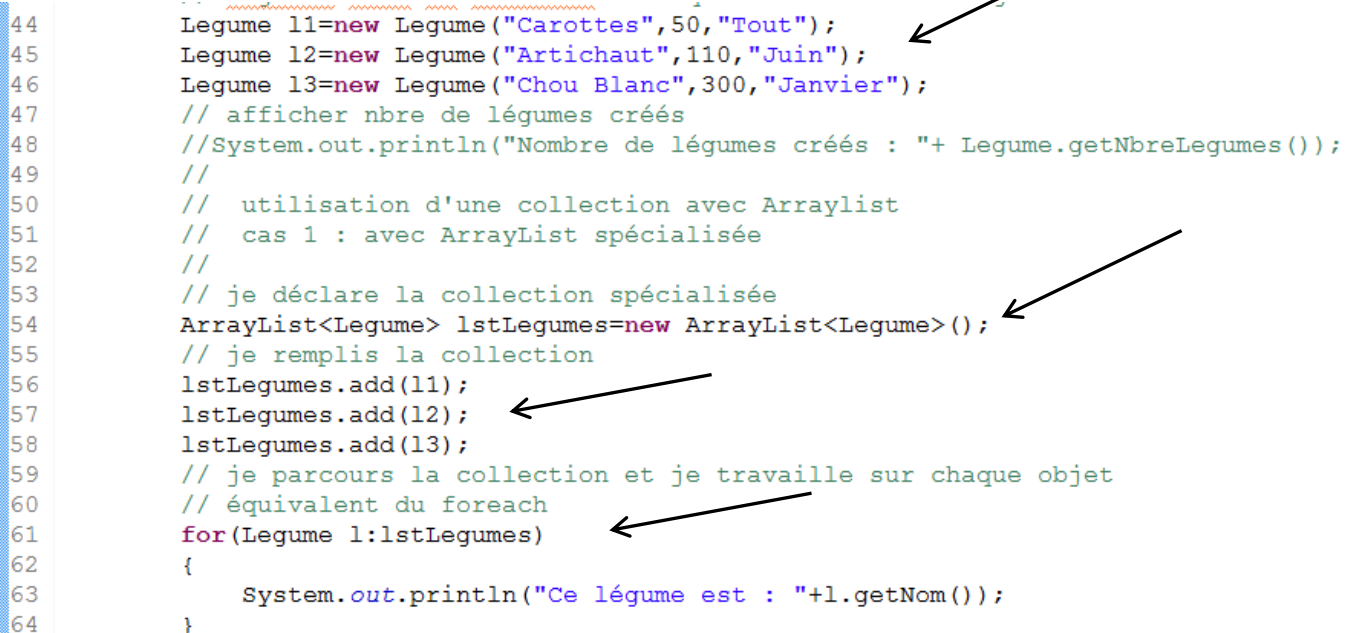
On peut également stocker des objets dans des tableaux. On utilise dans ce cas-là, le mot « **collection** » plutôt que tableau

Plusieurs classes java permettent de créer et d'utiliser des collections d'objets.

Ici je vais vous en présenter deux : la classe «**ArrayList**» et la classe «**Vector**»

Exemple 1 : création de plusieurs objets légumes, leur stockage dans une collection

```
44 Legume l1=new Legume("Carottes",50,"Tout");
45 Legume l2=new Legume("Artichaut",110,"Juin");
46 Legume l3=new Legume("Chou Blanc",300,"Janvier");
47 // afficher nbre de légumes créés
48 //System.out.println("Nombre de légumes créés : "+ Legume.getNbLegumes());
49 //
50 // utilisation d'une collection avec ArrayList
51 // cas 1 : avec ArrayList spécialisée
52 //
53 // je déclare la collection spécialisée
54 ArrayList<Legume> lstLegumes=new ArrayList<Legume>();
55 // je remplis la collection
56 lstLegumes.add(l1);
57 lstLegumes.add(l2);
58 lstLegumes.add(l3);
59 // je parcours la collection et je travaille sur chaque objet
60 // équivalent du foreach
61 for(Legume l:lstLegumes)
62 {
63     System.out.println("Ce légume est : "+l.getNom());
64 }
65
```



Ci-dessus, la collection contient des objets identiques : elle est spécialisée pour contenir des légumes.

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Exemple 2 : création d'une collection d'objet divers et variés

```
68 // utilisation d'une collection avec ArrayList
69 // cas 2 : avec ArrayList non spécialisée
70 //
71 ArrayList lstHeteroclites=new ArrayList();
72 lstHeteroclites.add("hello");
73 lstHeteroclites.add(123);
74 lstHeteroclites.add(true);
75 lstHeteroclites.add(57.30);
76 // parcourir une collection hétéroclite
77 for(int j=0;j<lstHeteroclites.size();j++)
78 {
79     System.out.println("liste hétéroclite contient à la position "+j+
80         " : "+lstHeteroclites.get(j));
81     if(lstHeteroclites.get(j) instanceof String)
82     {
83         System.out.println("en plus c'est une chaîne");
84     }
85 }
```

Problems @ Javadoc Declaration Console

<terminated> TestClasses [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (9 juil. 2014 11:57:59)

```
liste hétéroclite contient à la position 0 : hello
en plus c'est une chaîne
liste hétéroclite contient à la position 1 : 123
liste hétéroclite contient à la position 2 : true
liste hétéroclite contient à la position 3 : 57.3
```

La classe «**Vector**» est plus récente qu'«**ArrayList**» et elle s'utilise comme dans l'exemple 1 pour stocker des objets spécialisés : A préférer.

Agrégation ou Membre Objet

L'un des principes puissants de la POO est la modularité. Autrement dit : un objet peut être composé lui-même d'objets composants. Prenons quelques exemples de la vie courante :

→ **Le corps humain** sera vu comme l'objet principal mais il est composé d'autres objets :

- La tête est un objet lui-même composé de :
 - L'objet crâne,
 - L'objet cerveau,...
- Un bras est un objet lui-même composé de :
 - Plusieurs doigts : chacun d'eux est un objet,
 - L'avant-bras est un objet,
 - Le coude est un objet,
-

→ **Une voiture** sera vue comme l'objet principal mais il est composé d'autres objets :

- Le moteur est un objet lui-même composé de
 - L'objet bloc moteur,
 - Les objets soupapes,

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Utilisation d'un membre Objet

Exemple :

Considérons la modélisation d'un PC portable comportant :

- Un écran,
- Un clavier qui contient lui-même des touches,
- Un processeur,
-

Créons quelques classes pour illustrer l'agrégation :

```
1 package pack_essais;
2
3 public class Ecran {
4     // Attributs
5     private int hauteur;
6     private int largeur;
7     private int diagonale;
8     private String typeEcran; // technologie utilisée dans l'affichage
9 }
10
```

```
1 package pack_essais;
2
3 public class Touche {
4     // Attributs
5     private String codeTouche;
6     private char[] caracteresRepresentes;
7     private int positionVerticale;
8     private int positionHorizontale;
9 }
10
```

```
1 package pack_essais;
2
3 import java.util.ArrayList;
4
5 public class Clavier {
6     // Attributs
7     private boolean estDetachable;
8     private int largeur;
9     private int longueur;
10    private ArrayList<Touche> listeTouches; // collection des touches du clavier
11 }
12
```

Membre Objet = Agrégation

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

```
1 package pack_essais;
2
3 public class Processeur {
4     // Attributs
5     private String marque;
6     private String type;           // intel ou AMD ou ...
7     private String puissance;     // Core i3 ou i5 ou ....
8     private int vitesse;          // en Hz
9 }
10
```

```
1 package pack_essais;
2
3 import java.util.Date;
4
5 public class PcPortable {
6     // Attributs
7     private String marque;
8     private String modele;
9     private double prix;
10    private Ecran ecr;             // agrégation ou Membre objet
11    private Clavier clav;          // agrégation ou Membre objet
12    private Processeur prc;
13    // constructeur
14    public PcPortable(String marque, String modele, double prix,
```

Voici un code pour créer un PC :

```
92    charRep[0] = 'A';
93    Touche t1 = new Touche("A", charRep, 3, 2);
94    charRep[0] = '1';
95    charRep[1] = '&';
96    Touche t2 = new Touche("1", charRep, 2, 3);
97    // je mets les touches clavier dans une collection
98    ArrayList<Touche> lesTouches = new ArrayList<Touche>();
99    lesTouches.add(t1);
100    lesTouches.add(t2);
101    // je crée un clavier
102    Clavier cl = new Clavier(false, 35, 22, lesTouches);
103    // je crée un écran
104    Ecran ec = new Ecran(20, 30, 45, "Plasma");
105    // je crée un processeur
106    Processeur pr = new Processeur("HP", "Intel", "Corei5", 1.8);
107    // je crée un PC
108    PcPortable monPc = new PcPortable("HP", "Ultrabook", 530.00, ec, cl, pr);
109    // j'affiche des infos sur le pc
110    System.out.println("Ce PC a comme processeur : " +
111        monPc.getPrc().getMarque() + " - " + monPc.getPrc().getPuissance());
112
113 }
114 //
115
```

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Exercices récapitulatifs

Voir document exercices POO.

Héritage

Dans la conception d'un jeu vidéo, il est question d'armes avec lesquelles le personnage combatte.

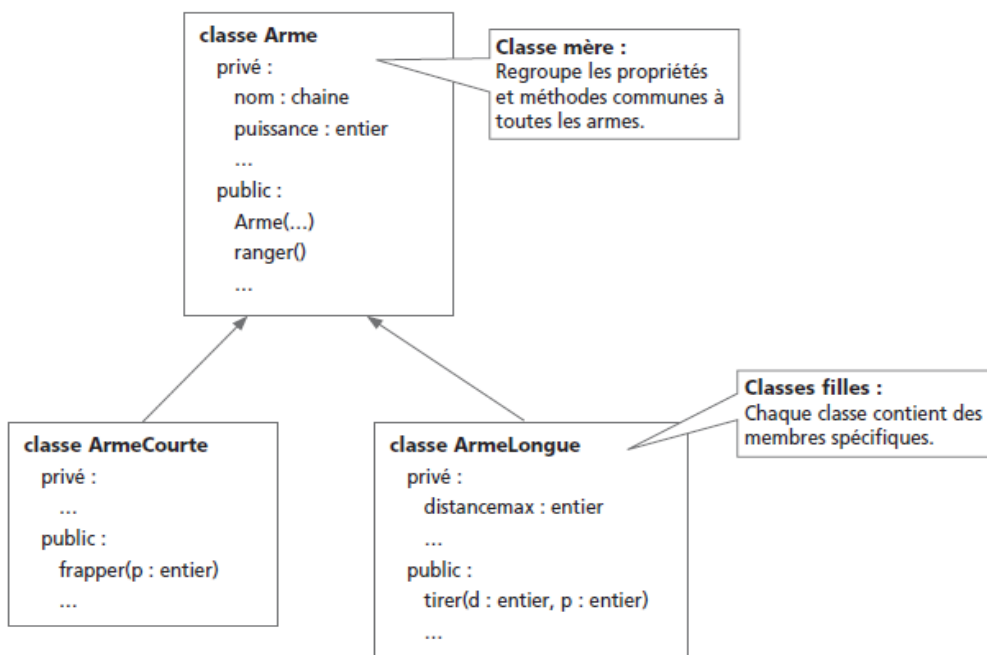
Il y a les armes courtes avec lesquelles il frappe et les armes longues avec lesquelles il tire.

Ces deux armes ont des caractéristiques communes et des caractéristiques spécifiques.

Ce serait dommage d'avoir deux classes complètement distinctes comportant certains membres en double.

Il est possible de créer une classe «**Arme**» qui contiendra les membres communs aux deux types d'armes

Et, ensuite de créer deux classes **filles** «**ArmeCourte**» et «**ArmeLongue**» spécifiques à chaque type d'armes et héritant de la classe **mère** «**Arme**»



Activer Wind

On parle alors, en programmation d'héritage entre les classes :

- ➔ La classe «**Arme**» est la classe **Mère** ou classe de **Base**,
- ➔ La classe «**ArmeCourte**» hérite de la classe «**Arme**» grâce au verbe «**extends**»,
- ➔ La classe «**ArmeCourte**» est la classe **filles** ou classe **dérivée**

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Prenons un autre exemple et montrons comment fonctionne l'héritage en Java :

On participe à la fabrication d'un logiciel pour vétérinaires :

- ➔ Le vétérinaire peut être un vétérinaire de ville ou de campagne ou de réserve sauvage : donc les animaux concernés par les soins peuvent être des animaux domestiques ou de campagne ou sauvages,
- ➔ Les animaux sont tous en commun : une catégorie (chiens, chats, tigres,), un poids,
- ➔ Les animaux sont divisés en animaux domestiques puis en animaux sauvages et en animaux de campagne,
- ➔ Les animaux domestiques sont précisément : des chiens, des chats,

Voici le code de la classe Mère :

```
package pack_essais;
/*
=====
  Classe Mère Abstraite
=====
*/
public abstract class Animal {
    // attributs
    private String categorie;           // chiens, chats, baleines, .....
    private int poids;
    // constructeur
    public Animal(String categorie, int poids) {
        super();
        this.setCategorie(categorie);
        this.setPoids(poids);
    }
    // getteurs + setteurs
    public String getCategorie() {
        return categorie;
    }
    public void setCategorie(String categorie) {
        this.categorie = categorie;
    }
    public int getPoids() {
        return poids;
    }
    public void setPoids(int poids) {
        this.poids = poids;
    }
    // méthodes abstraites : obliger les filles à les implémenter.
    public abstract void afficher();
    public abstract void creer();
    public abstract void modifier();
    public abstract void supprimer();
}
```

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Classe abstraite

Ci-dessus, la classe «**Animal**» a été déclarée avec le mot «**abstract**» et elle va servir pour l'héritage.

Si une classe doit servir pour faire de l'héritage mais on ne doit pas créer des objets (pas d'instances) à partir de cette classe (ici on ne créera pas des objets de type Animal) : alors on ajoute le mot « abstract » à la classe,

Méthodes abstraites

Si le créateur de la classe mère, souhaite imposer aux futures classes filles (qui vont donc hériter de la classe mère), l'obligation de coder certaines méthodes (on dit d'implémenter certaines méthodes), alors il crée des méthodes abstraites dans cette classe mère.

Dans le code ci-dessus, on a la ligne :

```
public abstract void afficher();
```

Vous remarquez qu'il y a juste la déclaration de la méthode « afficher » avec le mot «abstract» mais il n'y a pas de code.

C'est aux classes filles de coder le contenu de cette méthode.

Voici le code d'une classe fille «AnimalDomestique» :

```
ccc package pack_essais;
    /* =====
     *   Classe fille
     *   =====
     */
package pack_essais;
    /* =====
     *   Classe fille qui hérite de Animal
     *   =====
     */
public abstract class AnimalDomestique extends Animal{
    // attributs
    private String nom;
    private String race;
    // constructeur
    public AnimalDomestique(String categorie, int poids, String nom,
                            String race) {
        super(categorie, poids); // je passe la main au constructeur de la classe mère
        this.nom = nom;
        this.race = race;
    }
    // getteurs + setteurs
    public String getNom() {
        return nom;
    }
}
```

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

```
}
public void setNom(String nom) {
    this.nom = nom;
}
public String getRace() {
    return race;
}
public void setRace(String race) {
    this.race = race;
}
// méthodes abstraites : obliger les filles à les implémenter.
public abstract void soigner();
public abstract void tatouer();
public abstract void vacciner();
}
```

La classe fille ci-dessus est elle-même abstraite car on ne souhaite pas créer, dans le futur logiciel, des objets de cette classe (instancier des objets). Cette classe fille va servir elle-même l'héritage à un niveau en dessous.

Cette classe obligera les classes filles (Chien, Chat, ...) à implémenter les méthodes « **soigner** », « **tatouer** » et « **vacciner** » (méthodes abstraites).

Code de la classe Chien :

```
package pack_essais;
/* =====
 * Classe fille de 2ème niveau
 * =====
 */

public class Chien extends AnimalDomestique{
    // attributs
    private boolean estVaccine;
    private boolean estTaoue;
    private boolean estGrosChien;
    private boolean estCompatibleAppartement;
    // constructeur
    public Chien(String categorie, int poids, String nom, String race,
        boolean estVaccine, boolean estTatoue, boolean estGrosChien,
        boolean estCompatibleAppartement) {
        super(categorie, poids, nom, race);
        this.estVaccine = estVaccine;
        this.estTaoue = estTatoue;
        this.estGrosChien = estGrosChien;
        this.estCompatibleAppartement = estCompatibleAppartement;
    }
    // getteurs + setteurs
    public boolean isEstVaccine() {
        return estVaccine;
    }
}
```

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

```
}
public void setEstVaccine(boolean estVaccine) {
    this.estVaccine = estVaccine;
}
public boolean isEstGrosChien() {
    return estGrosChien;
}
public void setEstGrosChien(boolean estGrosChien) {
    this.estGrosChien = estGrosChien;
}
public boolean isEstCompatibleAppartement() {
    return estCompatibleAppartement;
}
public void setEstCompatibleAppartement(boolean estCompatibleAppartement) {
    this.estCompatibleAppartement = estCompatibleAppartement;
}
public boolean isEstTaoue() {
    return estTaoue;
}
public void setEstTaoue(boolean estTaoue) {
    this.estTaoue = estTaoue;
}
// méthodes imposées par la classe mère
@Override
public void soigner() {
    System.out.println("Le chien est soigné");
}
@Override
public void afficher() {
    System.out.println("Je suis un Chien");
}

@Override
public void tatouer() {
    this.estTaoue=true;
    System.out.println("Le Chien est maintenant tatoué");
}
@Override
public void vacciner() {
    this.estVaccine=true;
    System.out.println("Le Chien est maintenant vacciné");
}
// surcharge de la méthode
public void vacciner(double age) {
    if(age>1)
    {
        this.estVaccine=true;
        System.out.println("Le Chien est maintenant vacciné");
    }
}
```


Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

```
        else
        {
            this.estVaccine=false;
            System.out.println("Le Chien est encore jeune pour être vacciné");
        }
    }
    @Override
    public void creer() {
        // TODO Auto-generated method stub
    }
    @Override
    public void modifier() {
        // TODO Auto-generated method stub
    }
    @Override
    public void supprimer() {
        // TODO Auto-generated method stub
    }

    // méthode spécifique à cette classe
    public void promener(int duree)
    {
        System.out.println("Le chien se promène");
    }
    // méthode spécifique à cette classe
    public void aboyer()
    {
        System.out.println("Le chien aboie");
    }
}
```

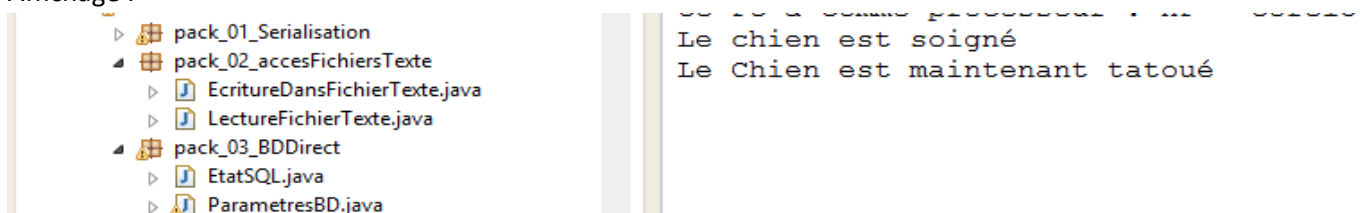
Ci-dessus, la classe «**Chien**» ne contient pas le mot «**abstract**» : on va donc pouvoir créer des objets à partir de cette classe.

Un code qui crée deux chiens logiciellement :

```
Chien ch1=new Chien("Chiens",15,"Horace","Labrador",false,false,false,true);

Chien ch2=new Chien("Chiens",7,"Milou","Caniche",false,false,false,true);
ch1.soigner();
ch2.tatouer();
```

Affichage :



Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Surcharge

Il arrive parfois de vouloir réaliser deux actions fonctionnellement presque identiques dans la même classe.

Par exemple dans la classe «Chien» : on souhaite de deux façons différentes :

- ➔ si on ne connaît pas l'âge du chien : on vaccine d'office d'une certaine façon,
- ➔ si on connaît l'âge du chien : on vaccine à partir d'un certain âge.

Pour résoudre ce genre de problématique, on peut créer deux méthodes de noms différents.

Mais comme, fonctionnellement, elles font la même chose mais différemment, on préfère les nommer du même nom et les différencier par le type et le nombre des paramètres qu'elles recevoient.

On appelle cela la surcharge.

Lorsque deux ou plusieurs méthodes, dans une même classe, portent le même nom : cela s'appelle la surcharge.

Voir ci-dessus, la classe « Chien » dispose de deux méthodes «vacciner»

Redéfinition

Ne pas confondre la surcharge et la redéfinition

Lorsque, dans une classe fille, on souhaite réécrire le code d'une méthode héritée de la classe mère : cela s'appelle «la redéfinition ou **Override**»

Voir ci-dessus :

- ➔ la classe mère « **AnimalDomestique** » déclare la classe « **soigner** »,
- ➔ la classe fille « **Chien** » redéfinit cette méthode «**soigner**»

Héritage multiple

Lorsqu'une classe ne peut hériter que d'une seule classe : on parle d'héritage simple.

Par opposition, lorsqu'une classe peut hériter de plusieurs classes : on parle d'héritage multiple.

Le java ne permet pas l'héritage multiple

Le C++ le permet,

Le C# l'interdit.

Héritage et protection

Imaginez que dans une classe Mère

Les attributs de la classe mère, s'ils sont privés, ne seront accessibles ni par les classes filles ni par un code extérieur aux classes filles : on appelle ce principe : l'encapsulation ou l'accessibilité.

Evidemment, on a la possibilité de créer des méthodes d'accès que l'on appelle les getteurs et les setteurs mais supposons que l'on ne veut pas les implémenter ici.

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Si je mets les attributs de la classe mère en «public», un code de la classe fille peut y accéder mais un code extérieur aux classes filles peut y accéder aussi.

Si on souhaite accéder aux attributs de la classe mère uniquement par les codes des classes filles : il y a un niveau de protection intermédiaire que l'on nomme «**protégé**» ou «**protected**».

Exemple :

- ➔ nous avons une classe mère qui s'appelle « Meuble » avec les attributs en **protected**,
- ➔ nous avons une classe fille qui s'appelle «**TableRectangulaire**» qui peut accéder aux attributs de la classe Mère directement dans la méthode «**afficher**»,
- ➔ un code extérieur : classe «**TestClasses**» avec sa méthode «main» n'arrive pas à accéder aux attributs de la classe Mère

```
package pack_essais;

/*
 *      =====
 *      Illustration de type de visibilité protected
 *      =====
 *      Classe Mère
 *
 */
public class Meuble {
    // attributs privés sans accesseurs :
    // même les classes filles ne peuvent pas y accéder
    //
    // attributs en protected : les méthodes des classes filles peuvent y accéder
    protected String modele;
    protected String nom;
    protected double prix;
    // constructeur
    public Meuble(String modele, String nom, double prix) {
        super();
        this.modele = modele;
        this.nom = nom;
        this.prix = prix;
    }
}
```

```
package pack_essais;

/*
 *      =====
 *      Illustration de type de visibilité protected
 *      =====
 *      Classe Fille
 *      =====
 *
 */
```

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

```
public class TableRectangulaire extends Meuble{
    // attributs
    private int longueur;
    private int largeur;
    private int hauteur;
    // constructeur
    public TableRectangulaire(String modele, String nom, double prix,
        int longueur, int largeur, int hauteur) {
        super(modele, nom, prix);
        this.longueur = longueur;
        this.largeur = largeur;
        this.hauteur = hauteur;
    }
    // méthode
    public void afficher()
    {
        // la ligne ci-dessous est en erreur quand le nom est privé
        // dans la classe mère
        // mais il n'est pas en erreur si je mets l'attribut nom en protected
        System.out.println("Table : "+this.nom);
    }
}
```

Code de la méthode main :

```
// =====
//          Test du niveau de protection "protected"
//          =====
//    table mère : Meuble, table fille : TableRectangulaire
//    si dans la table mère les attributs sont privés : même la classe
//    fille n'arrive pas à y accéder : voir méthode afficher
//    il faut les déclarer en protected et dans ce cas les codes extérieurs au
//    fille n'arrive pas à y accéder
Meuble me=new Meuble("Aurore","Meuble moderne",270.00);
// la ligne en dessous est en erreur parce que nom n'est pas
// accessible à partir de ce code qui est extérieur à la classe
// fille : TableRectangulaire
System.out.println("nom Table : "+me.nom);
```

Polymorphisme

ccc

Polymorphisme de surcharge

Ccc

Concepteur Développeur Informatique
Module : Programmation Java
Programmation Orientée Objets

Polymorphisme d'héritage

this / Super

Exercices récapitulatifs

Voir document exercices POO.

Classification, regroupement et opérations associées

Finale (classe, attribut, méthode)

Concepteur Développeur Informatique

Module : Programmation Java

Programmation Orientée Objets

Abstraire (classe, méthode)

Classe métier, classe technique

Classe générique : les collections

Pourquoi «classe générique» ?

Interface (classe purement abstraite)

Principe

Intérêt

Notion d'implémentation

Casting

Origine d'un objetccc

Objet dynamique

Principe

Intérêt

Principe du « ramasse miettes » ou «Garbage Collector»

Où sont les pointeurs ?

Bibliothèque, package

Intérêt d'une bibliothèque

Utilisation de bibliothèques existantes

Création d'une bibliothèque

Exercices récapitulatifs