République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Abderrahmane Mira Bejaia
Faculté des Sciences Exactes
Département d'Informatique

Polycopié de cours

Module : Programmation Orientée Objet (POO)

Niveau : 2^{e} année licence

Spécialité : Systèmes Informatiques

Elaboré par Dr. HAMZA Lamia

2022/2023

Table des matières

Table des figures

Introduction générale

Ce polycopié de cours introduit la Programmation Orientée Objet (POO) pour des étudiants informaticiens novices dans ce domaine. Il s'adresse aux étudiants de 2e année licence informatique ayant déjà programmé avec des langages basés sur une pensée fonctionnelle (procédurale), tels que les langages Pascal, C, etc.

Le principal objectif de ce polycopié est de permettre à l'étudiant de découvrir une nouvelle pensée basée sur les objets afin qu'il puisse programmer correctement dans un langage purement orienté objet, tel que Java. Pour cela, tous les codes de ce polycopié sont compilés sous Java et l'étudiant pourra les reprendre et les exécuter directement afin de vérifier les connaissances acquises dans ce cours.

Ce polycopié de cours est organisé en cinq chapitres, à savoir :

Chapitre 1 : intitulé « **Les principes de base de la POO** ». Ce chapitre commence par la présentation de l'historique de la POO tout en clarifiant les raisons du passage de la pensée fonctionnelle vers la pensée objet. Par la suite, il présente les concepts de base de la POO (modularité, réutilisation de codes, classes et communication entre objets). Enfin, il introduit le langage Java.

Chapitre 2 : intitulé « **La programmation objet** ». Le but de ce chapitre est de permettre à l'étudiant de se familiariser avec à la programmation objet en lui introduisant les mécanismes d'instanciation d'objet, d'encapsulation, de communication entre objets, d'implémentation des tableaux objets et des entrées/sorties sous Java.

Chapitre 3 : intitulé « **L'héritage en Java** ». Dans ce chapitre, l'étudiant va apprendre la réutilisabilité du code à travers le mécanisme d'héritage. Il acquerra la maîtrise des concepts liés à l'héritage, tels que : l'héritage simple, le polymorphisme statique, le polymorphisme dynamique, les classes abstraites, les interfaces, etc.

Chapitre 4 : intitulé « **Solutions des exercices du cours** ». Les corrigés des exercices du cours sont mis dans un chapitre à part afin d'inciter l'étudiant à s'autoévaluer.

Chapitre 5 : intitulé « **Tests de connaissances ».** Les exercices de ce chapitre permettront à l'étudiant de tester ses connaissances acquises dans ce polycopié et de consolider sa compréhension. Pour permettre à l'étudiant de s'autotester, un sujet d'examen et son corrigé sont présentés.

Ce document s'achève par une conclusion et des références bibliographiques.

Chapitre 1 : Les principes de bases de la POO

1. Introduction

L'objectif principal de ce chapitre est de permettre à l'étudiant de comprendre les principaux concepts de la pensée orientée objet. Il débute par un historique afin de mettre en évidence : les raisons du passage de l'approche procédurale à l'approche objet d'une part et les principaux concepts de la POO d'autre part. Il se termine par une introduction au langage Java.

2. Concepts fondamentaux de la POO

a.Bref historique de la POO

La programmation objet est introduite par des informaticiens scandinaves. En 1962, les Norvégiens Ole-Johan Dahl et Kristen Nygaard inventent le premier langage orienté objet, SIMULA I. Ce langage évolua en 1967 sous le nom de Simula 67 [1] et s'était le premier langage orienté objet. Les travaux de ces deux informaticiens ont ensuite influencé ceux d'Alan Kay, informaticien américain à l'origine du langage Smalltalk dans les années 70. La programmation objet n'est devenue vraiment populaire que 20 ans plus tard, dans les années 90, avec l'apparition des langages C++ (1985) et Java (1990). Actuellement, il existe plusieurs langages orientés objets, tels que : Java, PHP, Python, etc.

b. Programmation procédurale et programmation objet

Dans l'approche procédurale, le raisonnement se fait comme suit [2] :

- Raisonnement en termes de fonctions du système ;
- Séparation des données et du code de traitement ;
- Décomposition fonctionnelle descendante.

Ce raisonnement *a plusieurs limites, nous citons :*

- Un programme est conçu comme un ensemble de modules fonctionnels (procédures ou fonctions/procédures) qui manipulent des données ;
- Accès libre aux données par n'importe quelle fonction ;
- Difficulté de réutiliser du code déjà écrit et testé.

Les principales questions posées sur l'approche procédurale sont :

- ❖ La séparation (données, fonctions/procédures) est-elle réellement utile ?
- ❖ Pourquoi ne pas considérer que les programmes sont des ensembles d'entités caractérisés par les opérations qu'ils connaissent ?

La réponse à ces questions a fait naissance d'une nouvelle pensée qui est l'orienté objet. Contrairement à l'approche procédurale, l'approche orientée objet repose sur l'observation de la façon dont nous procédons dans notre vie de tous les jours. Ainsi, les langages objets sont fondés sur la connaissance d'une seule catégorie d'entités informatiques : ***les objets***.

L'approche orientée objet se focalise sur [2] :

- Regroupement données-traitements ;
- Diminution de l'écart entre le monde réel et sa représentation informatique ;
- Décomposition par identification des relations entre objets.

c. Réutilisation de code

En terme général, la réutilisation peut se définir comme étant l'usage de concepts précédemment acquis dans une nouvelle situation. En informatique, la réutilisation se résume à l'action de reprendre un code logiciel existant pour le réemployer, éventuellement en l'adaptant, dans un contexte présentant certaines similarités avec le programme d'origine.

Remarque :

La réutilisabilité dans les langages orientés objet peut être prise en considération dès la conception d'une application grâce au mécanisme d'héritage.

Nous allons étudier dans le chapitre 3 l'héritage en Java.

d. Modularité

Avant d'expliquer le concept de modularité, nous devons d'abord clarifier la notion du module en programmation.

Un module est un morceau de code implémentant une fonctionnalité (variables et codes sources) bien définie et dont la description est faite à travers une interface précise de sorte qu'on n'a pas besoin de connaître son code pour utiliser un module [3]. Ainsi, la modularité facilite le développement d'un logiciel, la localisation, la correction des erreurs et la réutilisation des modules.

3. Notions d'objet et de classe

a. Notion d'objet

Un objet est une entité représentée par un état et un ensemble d'opérations qui manipulent cet état.

Un **objet** est caractérisé par [2] :

- un **identificateur** (nom de l'objet) ;
- un **état** (sous forme d'un ensemble d'**attributs**) ;
- un **comportement** (un ensemble d'**opérations**).

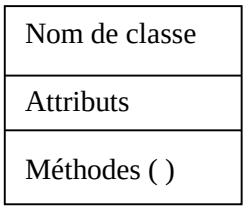
b. Notion de classe

Une classe correspond à la description d'une famille d'objets ayant une même structure et un même comportement.

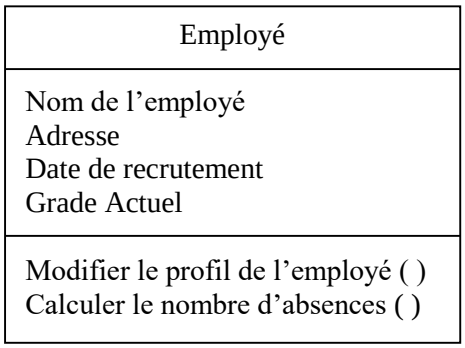
Une classe définie, une partie statique et une partie dynamique [2] :

- la partie statique est représentée par un ensemble **d'attributs** pouvant posséder une valeur (ces attributs caractérisent l'état des objets durant leurs exécutions).
- la partie dynamique est représentée par l'ensemble des **méthodes** qui définissent le comportement commun aux objets de la même classe.

Notation graphique[1] :



Exemple de classe :



Remarque :

Un objet est une instance créée à partir d'une classe. La création d'un objet à partir d'une classe s'appelle l'**instanciation**.

Nous allons étudier dans le chapitre 2 l'instruction permettant d'instancier les objets en Java.

[1]Les notations graphiques utilisées dans ce cours sont inspirées d'UML (Unified Modeling Language).

c. Notion de message

Un message en programmation objet est le seul moyen de communication entre objet. Il transporte l'information nécessaire aux demandes à satisfaire par l'invocation de méthodes d'objet. Notez qu'il est impossible d'accéder directement aux données encapsulées d'un objet, contrairement à l'approche procédurale où l'accès aux données est sans restriction.

4. Introduction à Java

Java (café en argot anglais) est un langage de programmation dont le squelette principal est constitué du langage C++. Son avantage majeur réside dans l'efficacité de son utilisation sur le WEB et la réalisation de jeux fortement basé sur le graphisme. En effet, il permet la création de graphismes élaborés, animés, ainsi que la présentation de textes évolués (hypertexte).

4.1. Historique

Le langage Java trouve ses origines dans les années 1990. A cette époque, quelques ingénieurs de SUN Microsystems commencent à parler d'un projet d'environnement indépendant du hardware pouvant facilement permettre la programmation d'appareils aussi varié que les téléviseurs, les magnétoscopes, etc. James Grosling ingénieur chez SUN Microsystems développe un premier langage, OAK, permettant de programmer dans cet environnement. En 1992, tout est prêt pour envahir le marché avec cette nouvelle technologie, mais la tentative se solde par un échec.

Bill Joy (co-fondateur de SUN Microsystems) sauve malgré tout le projet. Devant la montée en puissance de l'Internet, il lui semble intéressant d'insister sur l'élaboration d'un langage indépendant des plates-formes et des environnements. Dès lors, tout s'accélère. OAK est renommé (en 1995) Java et soumis à la communauté Internet grandissante.

4.2. Java et la portabilité

Dans la plupart des langages, on dit qu'un programme est portable, car un même code source peut être exploité dans des environnements différents. En Java, la portabilité va plus loin. En effet, la compilation d'un code source produit, non pas des instructions machine, mais un code intermédiaire formé de byte-codes [4]. D'une part, ce code est exactement le même, quel que soit le compilateur et l'environnement concernés. D'autre part, ces byte codes sont exécutables dans toute implémentation disposant du logiciel d'interprétation nommé machine virtuelle (JVM *:* Java Virtual Machine).

4.3. Concepts de base

a. accolade, double barre et point-virgule

Le langage Java emploie des accolades ouvrantes et fermentes, afin de délimiter les sections du programme. Il possède trois styles de commentaires :

Commentaires multi-lignes

```
/*

.........................
.........................

*/
```

Commentaires sur une seule ou fraction de ligne

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

Remarque : Il existe une troisième syntaxe de commentaire.

```
/**
*
*
*/
```

Cette syntaxe a une utilité particulière. Elle permettra de générer la documentation d'un programme (on l'appelle « Javadoc » pour « Java Documentation ») [7]. Mais nous n'allons pas l'utiliser dans ce document.

Exemple de commentaire uni-ligne :

```
// Je suis un commentaire
```

Remarque : lorsque vous saisissez une instruction en Java, elle doit être suivie obligatoirement par un point-virgule, ceci afin de marquer la fin et le début de l'instruction suivante.

Exemple : `int valeur = 10 ;`

b. L'instruction class

Tous les programmes Java commencent par cette instruction :

class Nomprogramme

{

…… ;

}

Cette instruction permet de donner un nom au programme, mais elle permet également d'appeler et d'utiliser des classes. Le nom du programme doit correspondre au nom du fichier contenant le code du programme.

Exemple :

```
class Essai
{
// je donne le nom 'Essai' à mon programme
}
```

Le nom du fichier qui contiendra le corps du programme sera sauvegardé sous le nom «`Essai.java`» qui sera ensuite soumis à la compilation.

Remarques :

1) Le terme « class » ne doit pas porter le caractère « e » à la fin
2) Par convention, la première lettre de chaque mot composant le nom de la classe doit être en majuscule.

c. L'instruction main

L'instruction main a le même rôle que la fonction « main() » déjà étudiée en langage C, elle désigne à l'ordinateur que la partie principale du programme commence ici.

Exemple :

```
public static void main (String arg[])
{
 // La partie principale du programme commence ici
}
```

Cette ligne est indispensable après l'instruction class qui va permettre de reconnaître, lors de la compilation, le début des instructions de la partie principale. Elle identifie la partie qui sera traitée la première, car Java est un langage organisé en section.

d. L'instruction System.out.println

Cette instruction commande à l'ordinateur d'afficher une ligne sur le dispositif de sortie du système, en l'occurrence l'écran.

Exemple :

```
System.out.println ("Bonjour") ;
```

Tout ce qui se trouve entre les guillemets est affiché.

En résumé, voici l'exemple d'un programme regroupant les principales notions évoquées dans cette section.

```
class Essai {
public static void main (String arg[]) {
System.out.println ("mon premier programme Java") ;
}
}
```

L'exécution de ce programme affichera :

```
mon premier programme Java
```

4.4. Programmation élémentaire en Java

4.4.1. Type de données

Les données informatiques sont de deux types :

- numériques
- alphanumériques

Quel que soit son type, une variable dans un programme Java se définit de la manière suivante :

- le type de donnée qu'elle contiendra
- son nom

Exemple **:** Déclaration d'un nombre entier

```
int a=10 ;
```

Cette instruction crée une variable numérique appelée `a` et lui affecte la valeur 10.

Il y a quatre sortes de données entières (byte, short, int et long), deux sortes de flottants (float et double), un type qui permet de représenter des données booléennes (boolean), et un type introduit par le mot clé char qui permet la gestion des caractères.

Remarque :

Les noms de variables en Java peuvent comporter des lettres ou des nombres à l'**exception des espaces** et de **la ponctuation**. Elles peuvent commencer par une lettre, par le caractère de soulignement « _ » ou par le signe «$».

4.4.2. Stockage

Dès la définition de la variable, Java permet de lui affecter une valeur initiale en utilisant le signe « = ». À tout moment, dans le programme, on peut modifier la valeur d'une variable toujours en utilisant le signe d'affectation « = ».

4.4.3. Utilisation de chaînes de caractère

Java permet de stocker du texte, c'est-à-dire un ensemble de caractères ou simplement un seul caractère, qui peut être une lettre, un nombre, un signe de ponctuation ou tout autre symbole. Le type de variable employé pour stocker le caractère se définit avec le mot clé « char ».

Exemple : `char touche = 'A' ;`

Cette instruction déclare une variable s'appelle touche pouvant contenir un seul caractère, en l'occurrence `A`. La valeur du caractère doit être placée entre **apostrophes**.

La chaîne de caractères en Java se définit par le mot-clé « String ».

Exemple : `String nomprénom = "Bedjaoui Lya";`

Cette instruction déclare une variable de type String en lui affectant la valeur Bedjaoui Lya. La chaîne de caractères doit être placée entre **guillemets**, qui

n'appartiennent pas à la chaîne de caractères. Le mot-clé « String » commence obligatoirement par une majuscule.

Java donne la possibilité d'insérer plusieurs caractères spéciaux dans une chaîne.

- \' apostrophe
- \\ anti-slash
- \r retour de chariot
- \ » guillemet
- \t tabulation
- \n saut de ligne (newline)

> ***Remarque :***
>
> L'exercice 1 du chapitre 5 présente un bon exemple d'utilisation du caractère spécial \t .

4.4.4. Les opérateurs

Comme tout langage de programmation, Java donne la possibilité d'effectuer des opérations sur des nombres (addition, soustraction, multiplication, division, modulo). Elles sont représentées par les signes suivants :

- signe d'addition « + »
- signe de soustraction « - »
- signe de multiplication « * »
- signe de division « / »
- signe de modulo « % »

Exemple :

```
a = a + 10 ;
a = a - 10 ;
a = a * 2 ;
a = a / 2 ;
a = 15 % 4 ;
```

Java utilise également des opérateurs de comparaison :

inférieur	<
supérieur	>
égalité	= =
inférieur ou égal	<=
supérieur ou égal	>=
inégalité	!=

Java utilise aussi des opérateurs logiques :

OU logique	||
ET logique	&&
NON logique	!

Java utilise des opérateurs d'incrémentation et de décrémentation :

incrémentation	++
décrémentation	--

Remarque :

Un opérateur de type x++ permet de remplacer des notations telles que x=x+1 ou bien **x+=1.**

4.4.5. Les conditions

Comme dans toute programmation, Java donne la possibilité au développeur de faire des choix et de transmettre un message en fonction de conditions prédéfinies.

a. Instruction if, et if-else

L'instruction if est l'instruction de base de toutes les instructions de contrôle. Elle permet d'exécuter un certain morceau de code seulement lorsqu'une condition est évaluée comme elle est vraie. **L'instruction if-else** permet de réaliser l'alternative.

Exemple ***:***

```
if (note >= 10)
System.out.println (" Vous êtes reçu à l'examen ! ") ;
else
System.out.println ("Vous avez échoué à l'examen") ;
```

Remarque :

Il existe également un opérateur particulier propre au langage Java que l'on nomme l'opérateur ternaire. En effet, il s'agit d'afficher ou d'affecter une valeur basée sur la vérification d'une condition.

Exemple ***:***

```
System.out.print ((note >= 10) ? "Vous êtes reçu à l'examen ! " : " Vous avez échoué à l'examen ") ;
```

b. L'instruction Switch/Case

Le **Switch/Case** est une structure conditionnelle en Java qui permet de sélectionner un ensemble d'instructions à exécuter en fonction de la valeur d'une variable. La syntaxe de l'instruction Switch/Case en Java est :

```
switch (expression) {
case constant_1 :
instructions;
break;
case constant_2 :
instructions;
break;
...
case constant_n :
instructions;
break;
default :
instructions;
}
```

4.4.6. Les boucles

Les boucles sont une partie fondamentale de la plupart des langages de programmation. Elles se déclinent sous plusieurs formes suivant l'utilité et l'utilisation dans le code du programme. Dans Java, il existe trois structures itératives, **while**, **do_while** et **for**.

❖ La boucle **while** s'exprime de la manière suivante :

```
while (condition)
{
   instructions ;
}
```

❖ La boucle **do_while** s'exprime de la manière suivante :

```
do{
instructions;
} while (condition);
```

❖ La boucle **for** s'exprime de la manière suivante :

```
for (initialisation; condition_booléenne; incrémentation){
            instructions ;

  }
```

5. Exécution du code Java

Garce à la portabilité de Java (voir la Section 4.2), un code java est compilé dans un fichier exécutable en bytecode java par la JVM (Java Virtual Machine). Sachant qu'actuellement la JVM est intégrée directement dans toutes les plates-formes.

5.1. Exécution en ligne de commande

L'exécution du code Java en ligne de commande nécessite deux étapes :

1. compiler le code en bytecode Java à l'aide de la commande **javac** ;
2. exécuter le code à l'aide de la commande **java**.

5.1.1. Déroulement de l'exécution en ligne de commande

Pour programmer en ligne de commande (sous DOS), suivez les étapes suivantes :

1. Créez un dossier nommé Exercice 1 ;
2. Dans ce dossier, créer un fichier de type « document texte » autrement dit « .txt » et nommer le Essai ;
3. Réécrivez le code de la classe Essai (page 11 de ce chapitre) dans le fichier Essai.txt ;
4. Enregistrer ce fichier sous l'extension « .java »

5. Ouvrer un terminal et placez-vous dans le dossier Exercice1 par l'utilisation de la commande cd, comme illustré dans l'exemple de la Figure 1.1.

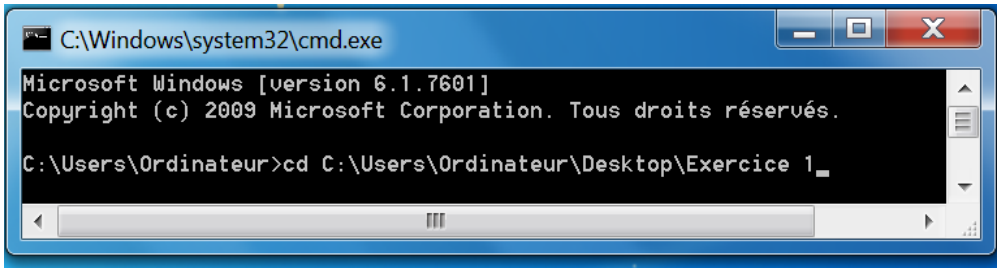


Figure 1.1 : Etape 1 de l'exécution sous DOS

Compiler le fichier Essai.java en exécutant la commande :

`javac Essai.java`.

Si le binaire Java n'est pas présent dans la variable d'environnement « PATH ». Le système vous renvoie l'erreur : « Java n'est pas reconnu en tant que commande interne » tel que montré dans la Figure 1.2.

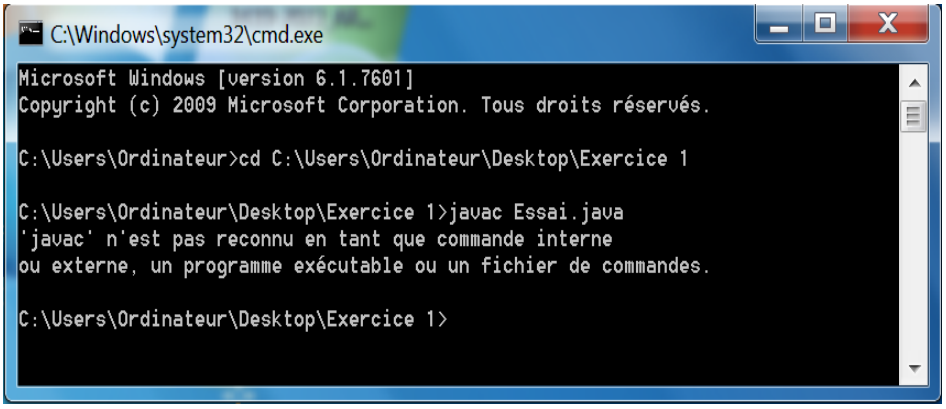


Figure 1.2 : Erreur liée à la variable d'environnement

Pour résoudre cette erreur, il suffit de suivre ces étapes :

1) Faire un clic droit sur l'icône « Ordinateur » puis cliquez sur « Propriétés » puis « Paramètres systèmes avancés » comme illustré dans la Figure 1.3.

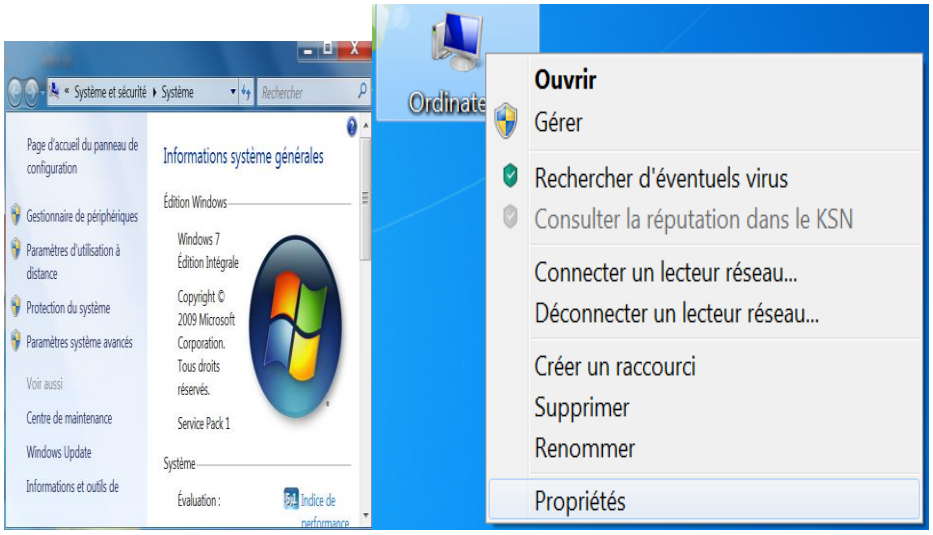


Figure 1.3 : Étape 1 pour résoudre l'erreur de la variable d'environnement

2) Cliquez sur le bouton « Variables d'environnement » qui se positionne en bas de la Fenêtre 1.4.

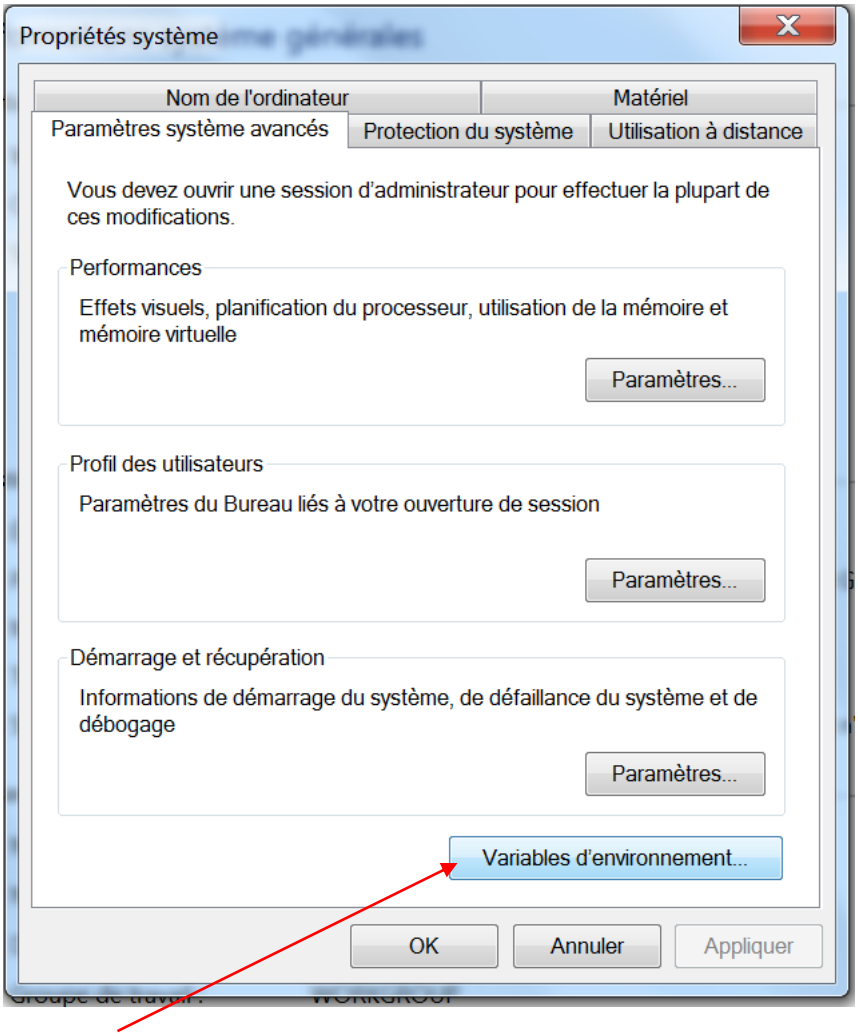


Figure 1.4 : Étape 2 pour résoudre l'erreur de la variable d'environnement

3) La fenêtre 1.5 s'affiche. Dans les « Variables Systèmes », trouvez PATH (voir la flèche rouge de la Figure 1.5).

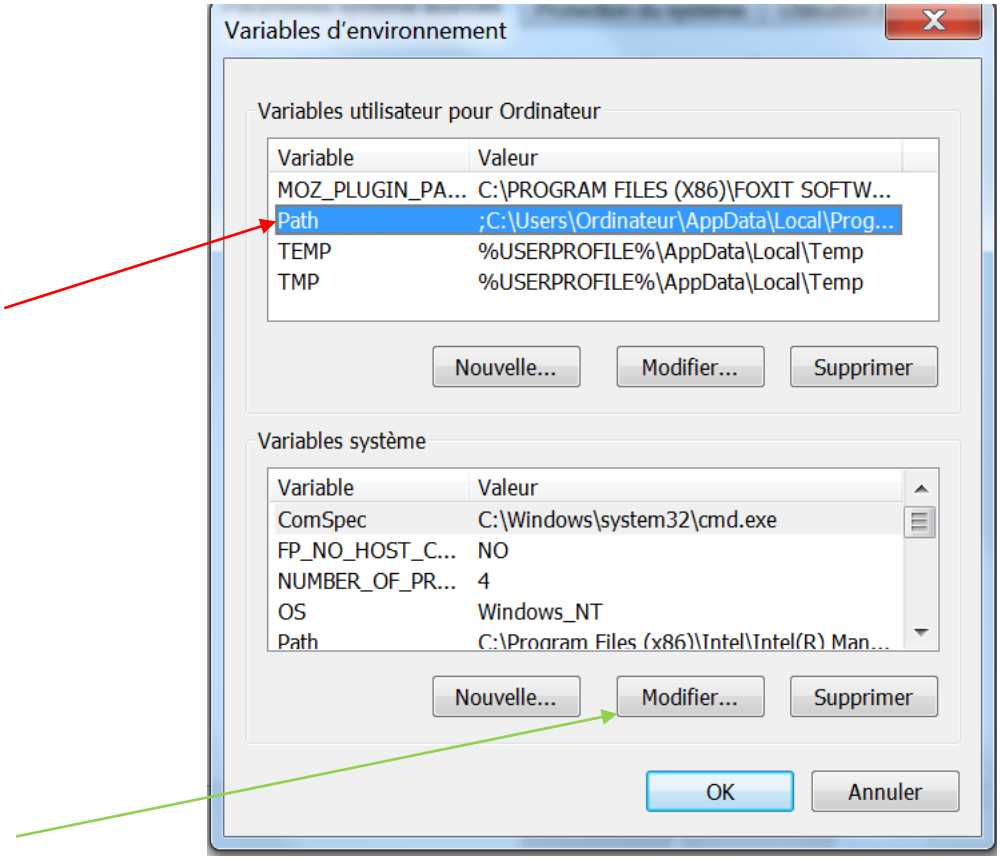


Figure 1.5 : Étape 3 pour résoudre l'erreur de la variable d'environnement

4) Cliquez sur modifier (voir la flèche verte de la Figure 1.5), puis ajouter cette ligne à la fin de la chaîne.

```
; C:\Program Files\Java\jdk1.8.0_191\bin
```

Remarque : « jdk1.8.0_191» est la version de Java installée sur l'ordinateur. Il se peut que vous n'ayez pas la même version, si tel est le cas allez vérifier votre version de JDK en suivant le chemin d'installation (C:\Program Files\Java).

Après cela, exécutez de nouveau la commande :

```
javac Essai.java
```

juste après, faites l'exécution du bytecode Java en lançant la commande `java Essai` et vous aurez le résultat de l'exécution du programme Essai sous DOS, comme illustré par la flèche mentionnée dans la Figure 1.6 :

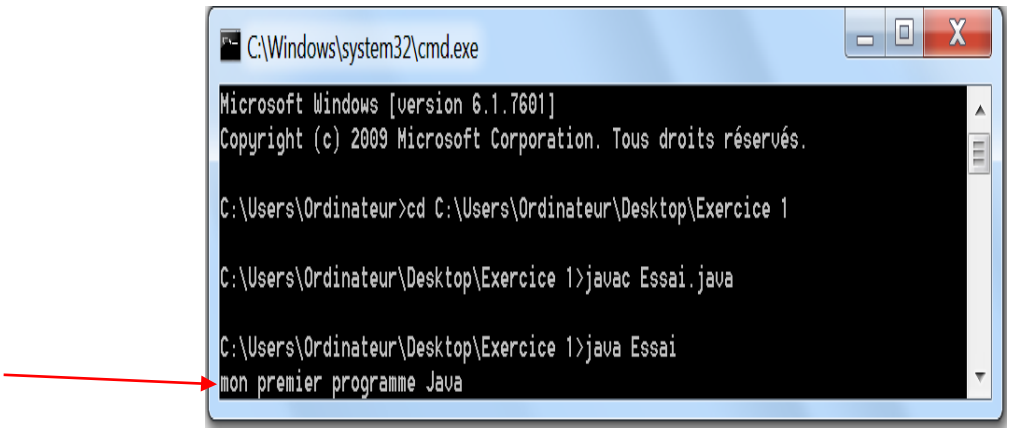


Figure 1.6 : Etape 2 de l'exécution sous DOS

Dans le répertoire « Ecercice 1 » vous allez remarquer la création d'un nouveau fichier nommé Essai.class à l'aide de la commande javac. Ce fichier est la sortie compilée de la classe Essai.java, réellement exécutée par la JVM.

5.1.2. Java sous Linux[2]

Les étudiants intéressés par l'installation de Java sous Linux peuvent suivre les étapes d'installation des liens [9, 10]. La ligne de commande « javac filename.java » est aussi applicable pour la compilation sous linux.

5.2. Exécution dans un environnement de développement (IDE)

Pour pratiquer Java en TP, nous allons utiliser un environnement de développement (Integrated Development Environment : IDE) [7]. Il existe de

[2] Les TPs de ce module se font sous Windows vu que les étudiants de Licence 2 Informatique sont en initiation du système Linux dans le cadre d'un autre module, à savoir le module Système d'Exploitation.

nombreux IDE pour Java. Dans ce cours, nous vous conseillons Eclipse en premier lieu puis, vous pouvez utiliser NetBeans ou un autre IDE.

5.2.1. Lancement de l'IDE Eclipse

Pour pouvoir utiliser Eclipse, vous devez le télécharger via ce lien http://www.eclipse.org/downloads/

Eclipse se lance simplement en double-cliquant sur l'exécutable eclipse.exe. Lors du premier lancement de l'Eclipse il vous demandera de localiser un répertoire particulier nommé workspace, dans lequel Eclipse enregistre vos projets et d'autres répertoires.

- **Création d'un Projet Eclipse**

  Pour pratiquer les exemples et exercices de ce polycopié de cours, nous allons voir en détail la création de projets :

  La manière la plus simple pour créer un premier projet est par le menu « File/New/JavaProject ». Après avoir choisi « Java Project», la fenêtre de création du projet apparaît comme illustré dans la Figure 1.7, dans le champ « Project name » saisissez un nom sans espace, ici le nom saisi est PremierTP. Si vous tapez le raccourci clavier « Alt+Shift+N» et vous choisissez « Java Project» vous aurez le même résultat.

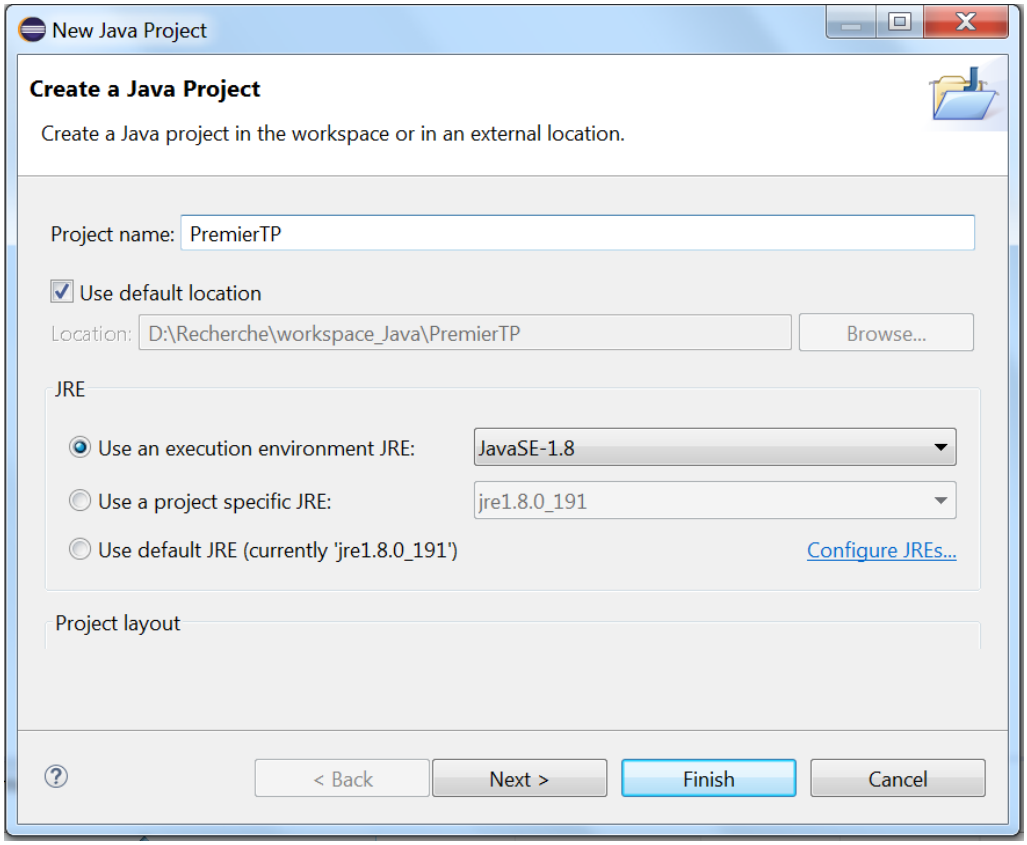


Figure 1.7 : Etape 1 de l'exécution sous Eclipse

Par la suite, il est possible de cliquer sur « Finish » et de finaliser la création du projet. Pour un premier programme, nous n'allons pas spécifier les paramètres des fenêtres d'Eclipse.

- **Ajouter des fichiers au projet crée**

L'ajout de nouveaux fichiers au projet (classes Java, interfaces, etc.) se fait simplement par clic droit sur un élément du projet auquel vous souhaitez ajouter un fichier. La Figure 1.8 suivante montre la première étape pour ajouter une classe au projet « PremierTP ».

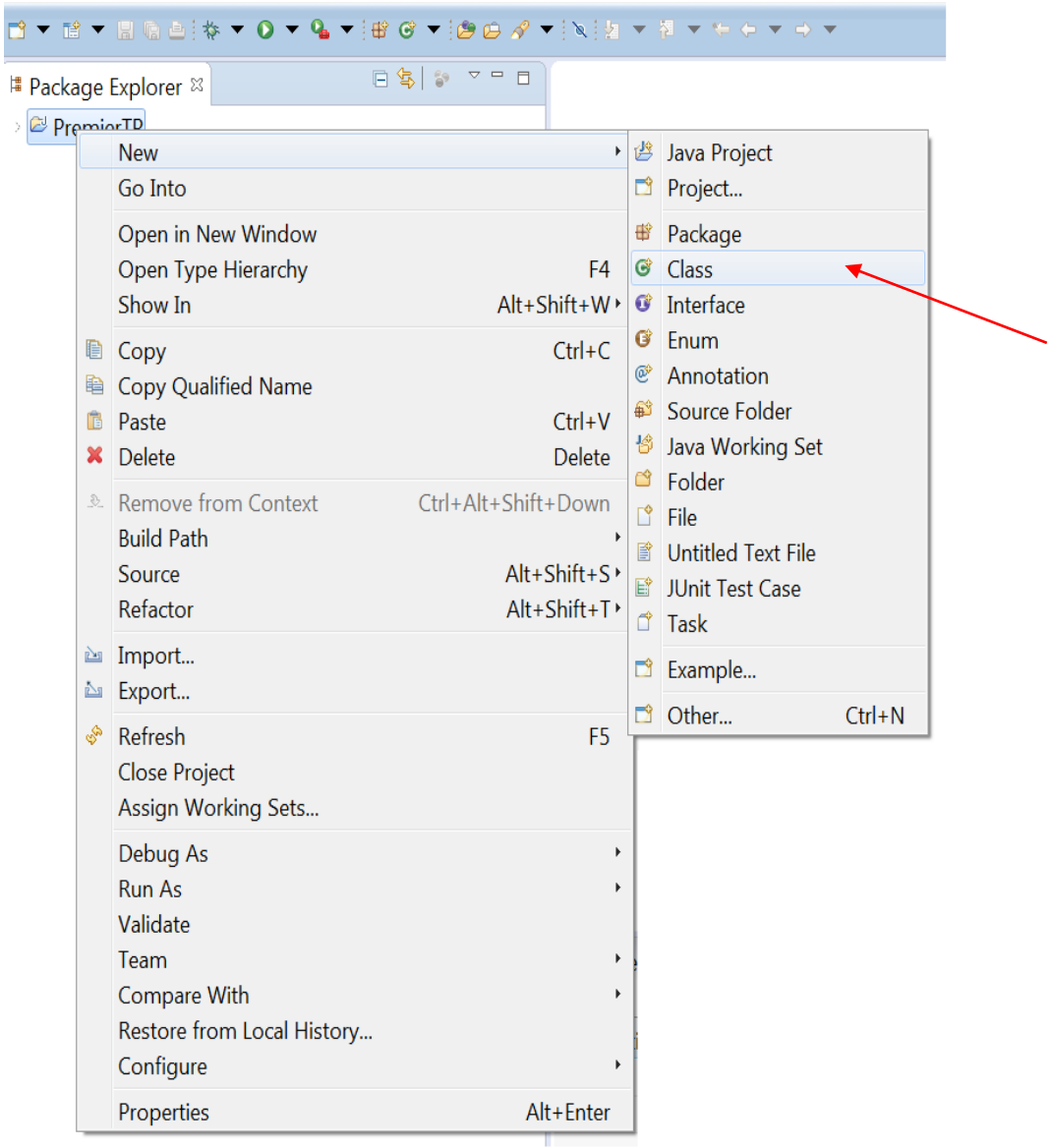


Figure 1.8 : Etape 2 de l'exécution sous Eclipse

Après un clic droit sur le projet et après avoir choisi « New/Class », la fenêtre de création de classe apparaît, tel que présenté par la Figue 1.9.

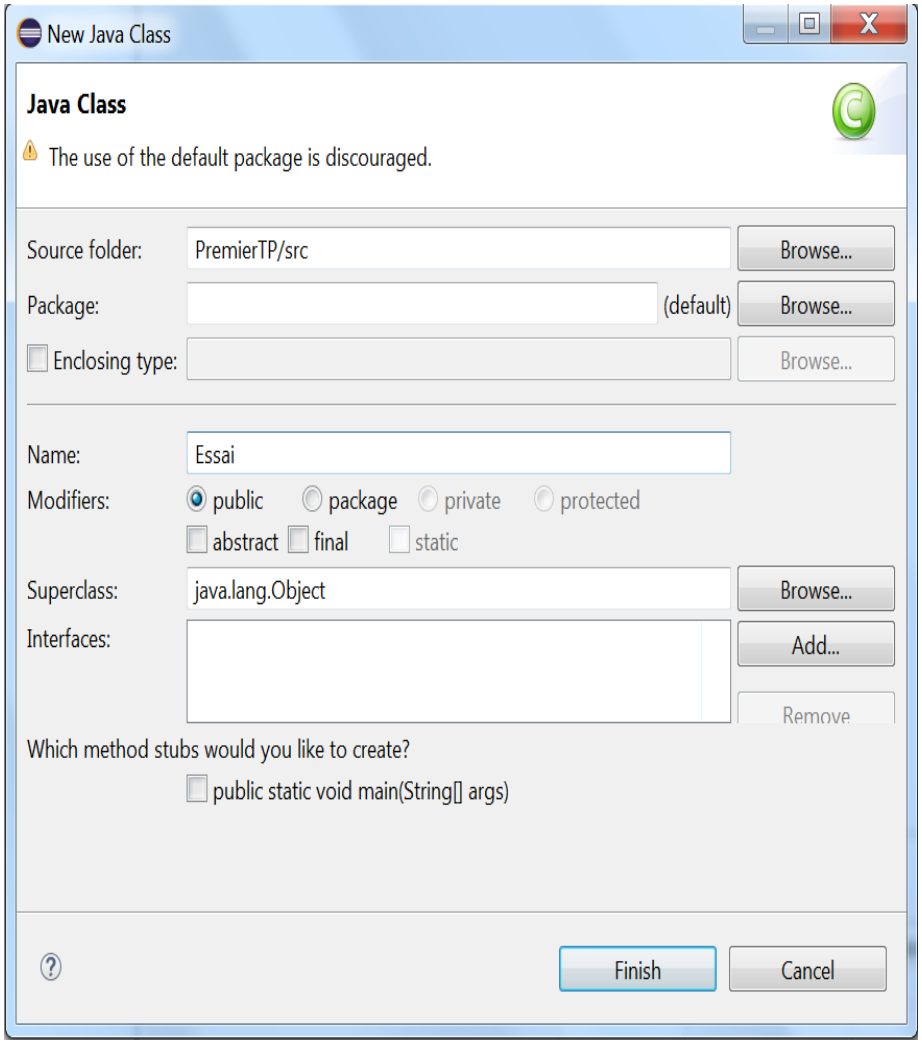


Figure 1.9 : Etape 3 de l'exécution sous Eclipse

Maintenant, il faut spécifier le nom de la classe. Dans notre cas, c'est la classe « Essai ». Une fois le nom saisi, le bouton « Finish » est activé et lorsque l'on clique dessus, le fichier « Essai.java » est créé et ajouté au projet. Il est automatiquement ouvert dans l'éditeur de code et, selon la spécification faite en haut, nous aurons un fichier vide. C'est fait exprès pour vous permettre d'écrire votre premier programme Java à la main.

Selon la flèche en rouge de la Figure 1.10. Pour lancer l'exécution du programme, il suffit de cliquer sur le bouton  dans la barre d'outils,

après avoir sélectionné la classe contenant la méthode « Main », ici c'est le programme « Essai» créé dans le projet « PremierTP ».

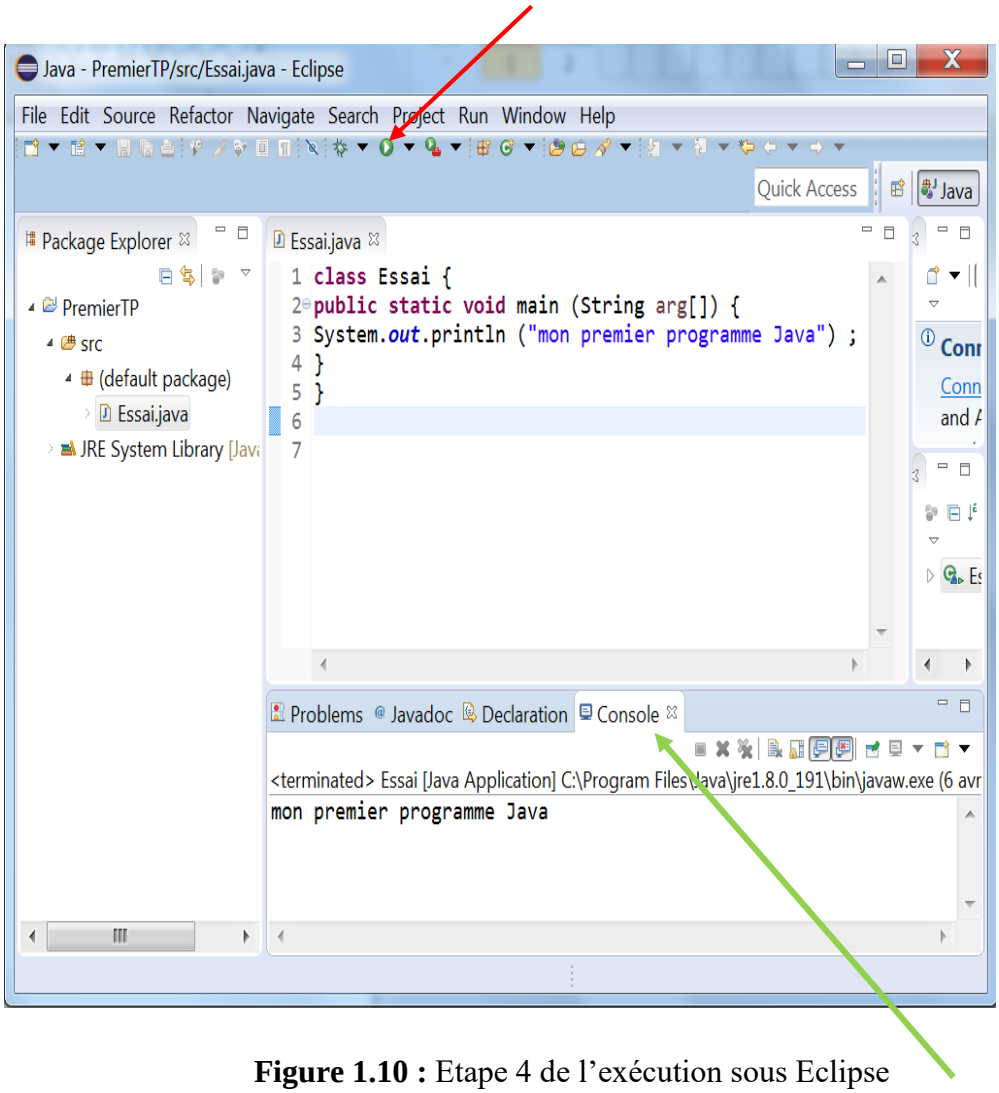


Figure 1.10 : Etape 4 de l'exécution sous Eclipse

Le résultat d'exécution sera affiché dans la partie « console » indiquée en flèche verte dans la Figure 1.10.

Le raccourci clavier CTRL+F11 permet aussi d'exécuter votre code.

Il est à noter que les IDEs Java Eclipse et NetBeans fournissent des fichiers de sortie « .class ».

- **Importation d'un projet Java existant :**
  Vous aurez certainement besoin d'importer des projets Java existants. Pour cela, voici les bonnes étapes à suivre :

- Tapez le raccourci clavier « Alt+Shift+N» et choisissez « Java Project» afin de créer un nouveau projet.
- Faire encore un clic droit sur le nom du projet et choisir l'option « Import … ». Choisir ensuite « General / File System » et appuyer sur Next.
- Choisissez le répertoire du projet que vous voulez importer.
- Dans l'arborescence sous le nom du répertoire, cocher tous les fichiers et faire Finish.

> ***Remarque :***
>
> Il est important de saisir l'intégralité de vos premiers programmes sous Eclipse (et non pas les copier/coller) pour se familiariser mieux avec la syntaxe Java. Une fois maitriser la syntaxe, vous pouvez passer à d'autres IDE, tel que NetBeans.

6. Conclusion

À l'issue de ce chapitre, l'étudiant aura compris qu'il peut programmer une application du monde réel en pensant seulement objet qui a des propriétés (attributs) et peut faire des actions (les méthodes). En outre, il comprendra la syntaxe d'un langage purement orienté objet, Java. Par ailleurs, il pourra exécuter ses programmes sous DOS ou dans un IDE de son choix. Sachant que dans ce cours l'IDE Eclipse est préconisé en premier pour une meilleure maitrise de la syntaxe Java. Une fois la syntaxe du langage Java est acquise l'étudiant pourra basculer vers NetBeans par exemple.

Chapitre 2 : La programmation objet

1. Introduction

La programmation orientée objet exige du développeur d'être rigoureux, mais aussi maitriser les concepts orientés objets (classes, objets, constructeurs, instanciation et encapsulation, etc.). Pour cela, le but de ce chapitre est d'explorer ces concepts à travers la présentation du langage Java.

2. Classe

Un objet (tel que mentionner dans le chapitre 1) est une entité qui contient des données qui définissent son état (on les appelle des attributs) et des fonctions (on les appelle des méthodes). Il est créé selon un modèle qu'on appelle une classe, la déclaration d'une classe nommée C1 en Java se fait comme suit :

```
public class C1{
type1  p1; // attribut p1
type2  p2; // attribut p2
…
type3  m3(…){ // méthode m3
…
}
type4  m4(…){ // méthode m4
…
}
…
}
```

Remarques **:**

1) La syntaxe utilisée pour définir le corps d'une méthode Java est identique à celle utilisée pour les fonctions en langage C.

2) La déclaration NomcClasse nomObjet permet de déclarer un objet sans le créer. Autrement dit, d'allouer une case mémoire destinée à recevoir la référence de l'objet.

Exemple :

```
Etudiant E1 ;
/*Cette instruction déclare juste l'objet E1 du
type Etudiant sans le créer*/
```

3) L'accès à un membre (attribut ou méthode) d'un objet donné se fait à l'aide de la notation à point. Cette notation possède deux parties séparées par un point : à gauche du point, on trouve la référence de l'objet et à sa droite le membre (attribut ou méthode) :

NomDeObjet.NomAttribut

ou

NomDeObjet.NomMethode( )

Exemple d'accès à une méthode d'un objet :

```
Etudiant E1 ;

E1.assisterCours();  /*Supposant que la méthode
assisterCours() est déjà défini dans la classe
Etudiant et que l'objet E1 est déjà instancié (le
mécanisme d'instanciation est détaillé dans la
Section 3 de ce chapitre)*/
```

3. Constructeur

Un constructeur est une méthode qui porte le **nom** de la classe et qui est appelée lors de la **création de l'objet**. On s'en sert généralement pour l'initialiser. C'est une méthode qui peut accepter des arguments mais qui **ne rend aucun résultat**. Son prototype ou sa définition n'est précédés d'**aucun type** (même pas **void**).

Si une classe a un constructeur acceptant **n** arguments *argi*, la déclaration et l'initialisation d'un objet de cette classe pourra se faire sous la forme [5] :

```
classe objet =new classe(arg1,arg2, ... argn);
```

ou

```
classe objet;
...
objet=new classe(arg1,arg2, ... argn);
```

Lorsqu'une classe a un ou plusieurs constructeurs, l'un de ces constructeurs doit être obligatoirement utilisé pour créer un objet de cette classe.

Si une classe C n'a aucun constructeur, elle en a **un par défaut** qui est **le constructeur sans paramètres**. Les attributs de l'objet sont alors initialisés avec des valeurs par défaut.

4. Encapsulation

Pour qu'une classe accède aux attributs ou méthodes d'une autre classe, Java impose d'utiliser le principe d'encapsulation permettant d'assurer une restriction de l'accès externe aux membres d'une classe considérés comme des détails d'implémentation internes. La concrétisation de ce principe est rendue possible par l'utilisation d'un ensemble de modificateurs, nous nous focalisons dans ce chapitre sur deux modificateurs : **public et private.** Ils peuvent être utilisés pour modifier la visibilité (portée) des classes, méthodes,

variables et package[3]. Pour mieux comprendre la notion d'encapsulation, nous nous concentrerons dans ce cours sur les visibilités des classes, des méthodes et des variables.

- **public** : **classes**, **méthodes**, **variables** déclarées « *public* » sont visibles par toutes les autres méthodes que ce soit à l'intérieur ou à l'extérieur du package de définition.

- **private** : c'est le degré de restriction le plus fort ; Elle signifie qu'une **méthode** ne pourra être appelée que depuis l'intérieur de la classe dans laquelle elle se trouve. Il en va de même pour les **variables**.

5. Le mot-clé « this »

Cette variable sert à référencer dans une méthode **l'instance de l'objet en cours d'utilisation**. Autrement dit, «this **»** est un objet qui est égal à l'instance de l'objet dans lequel il est utilisé. Il est préférable de préfixer la variable d'instance par le mot-clé « this **»**.

6. Les méthodes d'accès (getters et setters)

Les getters (ou accesseurs) et setters (ou mutateurs) sont des méthodes **public** afin de permettre l'accès depuis une autre classe.

Les getters fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses champs (généralement privés), sans les modifier. Ils sont du même type que la variable qu'ils doivent retourner. Par contre, les getters permettent de modifier l'état d'un objet (les valeurs de certains de ses champs). Ils sont de type void. Ces méthodes ne retournent aucune valeur parce qu'elles se contentent de les mettre à jour [4].

[3] Un package est un regroupement cohérent de classes par rapport à leurs fonctionnalités.

Remarque :

Eclipse à un mécanisme automatique de création de getter/setter, l'une des méthodes les plus pratiques est :

1. Dans votre code tapez simplement *set* puis ctrl+espace
2. Choisissez dans la liste qui s'ouvre le : *setNomDeVotreAttribut*
3. Si vous avez besoin d'un getter, suivez la même manipulation mais tapez *get* à la place de *set.*

Les accesseurs (getters) commencent par get et les mutateurs (setters) par set, c'est pour cela qu'on parle souvent de *getters* et de *setters*. Mais, on peut écrire manuellement des getters et setters sans débuter par get ou set.

7. Exemple d'une classe Personne

Nous abordons maintenant, par l'exemple, les concepts expliqués ci-dessus :

Prenant le cas d'une classe nommée Personne ayant les attributs prenom, nom, age, une méthode initialise et une méthode identifie.

```
public class Personne{
// attributs
private String prenom;
private String nom;
private int age;
// méthode
public void initialise(String P, String N, int age){
this.prenom=P;
this.nom=N;
this.age=age;
}
// méthode
```

```
public void identifie(){
System.out.println(prenom +","+ nom +","+age);
}
}
```

Nous avons ici la définition d'une classe. Rappelant qu'une classe est un moule à partir duquel sont construits des objets. Les **membres** d'une classe peuvent être des **attributs** ou des **méthodes**.

Supposant qu'on veut instancier une personne (objet) portant le prénom lya, le nom Bedjaoui et l'âge 30.

Un étudiant curieux posera certainement cette question : ***Quel est le rôle de la méthode initialise de la classe Personne ?***

Parce que **nom, prenom** et **age** sont des données **privées** de la classe **Personne**, les instructions suivantes si elles sont écrites dans une classe principale (contenant la fonction main), elles seront **incorrectes** :

```
Personne p1;
p1.prenom ="Lya";
p1.nom ="Bedjaoui";
p1.age =30;
```

Il nous faut initialiser un objet de type *Personne* via **une méthode publique**. C'est le rôle de la méthode *initialise*. On écrira :

```
Personne p1;
p1.initialise("Lya","Bedjaoui",30);
```

L'écriture *p1.initialise* est **légale** car ***initialise* est d'accès public**.

La **méthode initialise** a permet d'accéder aux attributs privés parce qu'elle a **une portée public**. Il est à noter que les méthodes d'accès (getters) et d'altération (setters) suivent le même principe.

Reprenant l'exemple de la classe Personne pour mieux expliquer le concept d'instanciation et d'accès aux membres d'un objet.

La séquence d'instructions suivante si elle est écrite dans une classe principale :

```
Personne p1;   // déclaration d'un objet personne  nommé p1
p1.initialise("Lya","Bedjaoui",30);
```

elle sera incorrecte aussi. L'instruction « Personne p1; » déclare *p1* comme une référence à un objet de type Personne. Cet objet n'existe pas encore et donc *p1* n'est pas initialisé.

Lorsqu'on écrit ensuite

```
p1.initialise("Lya","Bedjaoui",30);
```

on fait appel à la méthode `initialise` de l'objet référencé par *p1*. Or cet objet n'existe pas encore et le compilateur signalera l'erreur. Pour que *p1* référence un objet, il faut d'abord **créer l'objet en l'instanciant** par l'opérateur **new** :

```
Personne p1=new Personne();
```

Cela a pour effet de créer un objet de type Personne non encore initialisé : les attributs `prenom` et `nom` qui sont des références d'objets de type String auront la valeur `null`, et `age` la valeur *0*. Il y a donc une initialisation par défaut parce que nous avons utilisé le constructeur par défaut. Maintenant que *p1* référence un objet, l'instruction d'initialisation de cet objet `p1.initialise("Lya ","Bedjaoui",30);` est valide.

Regardons le code de la méthode *initialise* pour mieux comprendre le mot clé « this »

```
public void initialise(String P, String N, int age){
this.prenom=P;
this.nom=N;
this.age=age;
}
```

L'instruction `this.prenom=P` signifie que l'attribut *prenom* de l'objet courant (**this**) reçoit la valeur *P*. Le mot clé « this » désigne l'objet courant : celui dans lequel se trouve la méthode exécutée.

Regardons maintenant comment se fait l'initialisation de l'objet référencé par *p1* dans le programme appelant (la classe principale) :

```
p1.initialise("Lya","Bedjaoui",30);
```

C'est la méthode *initialise* de l'objet *p1* qui est appelée. Lorsque dans cette méthode, on référence l'objet **this**, on référence en fait l'objet *p1*. La méthode *initialise* aurait aussi pu être écrite comme suit :

```
public void initialise(String P, String N, int age){
prenom=P;
nom=N;
this.age=age;
}
```

> ***Remarque :***
>
> Lorsqu'une méthode d'un objet référence un attribut ***A*** de cet objet, l'écriture ***this.A*** est implicite. On doit l'**utiliser explicitement** lorsqu'il **y a conflit d'identificateurs**. C'est le cas de l'instruction : this.age=age;
>
> où *age* désigne un attribut de l'objet courant ainsi que le paramètre *age* reçu par la méthode. Il faut alors lever l'ambiguïté en désignant l'attribut *age* par ***this.****age*.

Considérons toujours la classe *Personne* et rajoutons-lui la méthode suivante :

```
public void initialise(Personne P){
prenom=P.prenom;
nom=P.nom;
this.age=P.age;
}
```

On a maintenant deux méthodes portant le nom *initialise* : c'est légal tant qu'elles admettent des paramètres différents. Le paramètre de cette méthode est maintenant une référence *P* à une personne. On remarquera que la méthode *initialise* à un accès direct aux attributs de l'objet *P* bien que ceux-ci soient de type *private*. C'est toujours vrai : les méthodes d'un objet *O1* d'une **classe *C*** a toujours accès aux attributs privés des autres objets de la **même classe *C*** [5].

Voici un test de la classe **Personne sans constructeur** :

```
//import Personne
/* Dans ce document, « import Personne » signifie que les
classes Personne et Test doivent être dans le même projet*/
public class Test{
public static void main(String arg[]){
Personne p1=new Personne();
p1.initialise("Lya"," Bedjaoui",30);
System.out.print("p1=");
p1.identifie();
Personne p2=new Personne();
p2.initialise(p1);
System.out.print("p2=");
p2.identifie();
}
}
```

NB : Il faut ajouter la méthode « initialise (Personne P) » à la classe Personne de la page 32 pour que ce code s'exécute.

L'exécution de ce programme donnera le résultat suivant :

```
p1= Lya, Bedjaoui,30
p2= Lya, Bedjaoui,30
```

7.1. Constructeurs de la classe Personne

Créons deux constructeurs à notre classe Personne :

```
public class Personne{
// attributs
private String prenom;
private String nom;
private int age;
// constructeur 1
public Personne(String P, String N, int age){
initialise(P,N,age);
}
// constructeur 2
public Personne(Personne P){
initialise(P);
}
// méthode
public void initialise(String P, String N, int age){
this.prenom=P;
this.nom=N;
this.age=age;
}
public void initialise(Personne P){
this.prenom=P.prenom;
this.nom=P.nom;
this.age=P.age;
}
// méthode
public void identifie(){
System.out.println(prenom+","+nom+","+age);
```

```
}
}
```

Nos deux constructeurs se contentent de faire appel aux méthodes *initialise* correspondantes. On rappelle que lorsque dans un constructeur, on trouve la notation *initialise(P)* par exemple, le compilateur traduit par *this.initialise(P)*. Dans le constructeur, la méthode *initialise* est donc appelée pour travailler sur l'objet référencé par *this*, c'est à dire l'objet courant, celui qui est en cours de construction.

Voici un programme de test de la classe Personne **avec constructeurs** :

```
//import Personne
public class Test1{
public static void main(String arg[]){
Personne p1=new Personne("Lya","Bedjaoui",30);
System.out.print("p1=");
p1.identifie();
Personne p2=new Personne(p1);
System.out.print("p2=");
p2.identifie();
}
}
```

L'exécution de ce programme donnera le résultat suivant :

```
p1= Lya,Bedjaoui,30
p2= Lya,Bedjaoui,30
```

8. Exercice 1

Soit le programme suivant :

```
//import  Personne
public class Test{
public static void main(String arg[]){
// p1
Personne p1=new Personne("Lya","Bedjaoui",30);
System.out.print("p1=");
p1.identifie();
// p2 référence le même objet que p1
Personne p2=p1;
System.out.print("p2=");
p2.identifie();
// p3 référence un objet qui sera une copie de l'objet
référencé par p1
Personne p3=new Personne(p1);
System.out.print("p3=");
p3.identifie();
// On change l'état de l'objet référencé par p1
p1.initialise("Mohamed","Amiri",67);
System.out.print("p1=");
p1.identifie();
// Comme p2=p1, l'objet référencé par p2 a dû changer d'état
System.out.print("p2=");
 p2.identifie();
// Comme p3 ne référence pas le même objet que p1, l'objet
référencé par p3 n'a pas dû changer
System.out.print("p3=");
p3.identifie();
}
}
```

Les résultats d'exécution obtenus sont les suivants :

```
p1= Lya,Bedjaoui,30
p2= Lya,Bedjaoui,30
p3= Lya,Bedjaoui,30
```

```
p1= Mohamed,Amiri,67

p2= Mohamed,Amiri,67

p3= Lya,Bedjaoui,30
```

- **Travail demandé :** Justifier ces résultats.

NB : le commentaire « // import Personne » fait référence à la classe Personne de la page 37.

9. Exercice 2

Ecrivez la classe Personne et une classe Test de votre choix en utilisant les accesseurs (les getters et les setters).

10. Destructeur

Afin de ne pas encombrer la mémoire des objets inutiles, il est indispensable de détruire les objets alloués dont on a plus besoin. Le destructeur est appelé par Java lorsqu'un objet n'est plus accessible. C'est le ramasse-miettes (garbage collector = récupérateur mémoire) qui s'en charge.

Le programmeur peut toutefois définir le corps du destructeur par la méthode finalize () [4].

Exemple :

```
class NomClass {
public finalize () {
...
}
}
```

Remarque :

Le ramasse-miettes est un mécanisme de gestion automatique de la mémoire employé par Java. Lorsqu'il n'existe plus aucune référence sur un objet. Il est donc possible de libérer l'emplacement correspondant, qui pourra être utilisé pour autre chose. Cependant, pour des questions d'efficacité, Java n'impose pas que ce travail de récupération se fasse immédiatement. En fait, on dit que l'objet est devenu candidat au ramasse-miettes.

11. Exercice 3

Ci-après vous trouverez le code de deux classes en JAVA : le code de la classe Vecteur, ainsi que le code de la classe EssaiVecteur. Trouver les erreurs qui se sont glissées dans ces lignes de codes.

NB : Une erreur peut se trouver répétée plusieurs fois. Dans ce cas, signaler chaque ligne où elle se produit. Quel que soit le nombre d'apparitions de l'erreur, ne la comptabiliser qu'une seule fois en tout.

```
01 class Vecteur
02 {
03    real x, y
04
05    public  static constructor(double abscisse, double
ordonnee)
06    {
07      x = abscisse
08      y = ordonnee
09    }
10
11    /* retourne true si le vecteur est égal à faux,
12      retourne false sinon
13     */
14    public static boolean egal( Vecteur v)
15    {
```

```
16      return (x = v.x ) and (y = v.y)
17    }
18
19    /* ajoute le vecteur v au vecteur
20     */
21    public static void ajoute( Vecteur v)
22    {
23      x = x + v.x
24      y = y + v.y
25    }
26 }
27


28 classe EssaiVecteur
29 {
30    public static void main( string arg[] )
31    {
32      Vecteur v1, v2
33      v1 = new Vecteur( 3.1, 5.2 )
34      v2 = new Vecteur( 2.4, 3.5 )
35      ajoute( v1, v2 )
36      Screen.println("Vecteur final = ")
37      Screen.out.println( "abscisse = " + v1.x + "
         ordonnee = " + v1.y  )
38
39 }
```

12. Exercice 4

Qu'affiche le programme suivant ?

```
01 class A {
02 public A(int coeff) {
03 nbre *= coeff;
04 nbre += decal;
```

```
05 }
06 public void affiche() {
07 System.out.println("nbre = " + nbre + " decal = " +
decal);
08 }
09 private int nbre = 20;
10 private int decal;
11 }

12 public class Init{
13 public static void main(String arg[]) {
14 A a = new A(5);
15 a.affiche();
16 }
17 }
```

13. Méthodes et attributs de classe

Remarque :

- En Java une méthode peut être une méthode d'instance, n'agissant que sur un seul objet (instance de la classe) à la fois ou **une méthode statique** (appelé aussi méthode de classe) indépendante de toute instance de la classe (objet).

- Dans tous les exemples de programmes que vous avez vus, la méthode main est qualifiée de **static**. Un attribut ou une méthode statique (i.e. déclaré avec le mot **static**) est aussi dit **attribut de classe** ou **méthode de classe**.

Le **modificateur static** indique, pour une méthode, qu'elle peut être appelée sans instancier sa classe. Pour un attribut, qu'il s'agit d'un attribut de classe,

et que sa valeur est donc partagée entre les différentes instances de sa classe [5].

Nous allons maintenant étudier la signification de ce modificateur par l'exemple :

Supposons qu'on veuille compter le nombre d'objets `personne` créées dans une application. On peut inclure un compteur dans la définition de la classe. Comme c'est un attribut de la classe elle-même et non d'un objet particulier de cette classe, on le déclare différemment avec le mot clé **static** :

```
private  static  long nbPersonnes; // nombre de personnes
créées
```

Pour le référencer, on écrit `Personne.nbPersonnes` pour montrer que c'est un attribut de la classe Personne elle-même.

Ici, nous avons créé un attribut privé auquel on n'aura pas accès directement en-dehors de la classe. On crée donc une méthode publique pour donner accès à l'attribut de classe `nbPersonnes` (qui est simplement un getter). Pour rendre la valeur de `nbPersonnes` la méthode n'a pas besoin d'un objet particulier : en effet `nbPersonnes` n'est pas l'attribut d'un objet particulier, il est l'attribut de toute une classe. Pour cela le getter doit être déclaré aussi **`static`** :

```
public static long getNbPersonnes(){
return nbPersonnes;
}
```

qui de l'extérieur sera appelée avec la syntaxe

`Personne.getNbPersonnes() ;`

Voici un exemple. La classe Personne est légèrement modifiée pour compter le nombre d'objets personnes créées :

```
public class Personne {
// attributs d'objets
private String prenom;
```

```
 private String nom;
 private int age;

// attributs de classe
 private  static  long nbPersonnes;

 // constructeurs
 public Personne(String P, String N, int age){
 initialise(P,N,age);
 nbPersonnes++;
 }
 public Personne(Personne P){
 initialise(P);
 nbPersonnes++;
 }
 // méthodes
 public void initialise(String P, String N, int age){
 this.prenom=P;
 this.nom=N;
 this.age=age;
 }
 public void initialise(Personne P){
 this.prenom=P.prenom;
 this.nom=P.nom;
 this.age=P.age;
 }
 public void identifie(){
 System.out.println(prenom+","+nom+","+age);
 }
 // méthode de classe
 public static long getNbPersonnes(){
 return nbPersonnes;
 }
 }
```

Avec le programme suivant :

```
public class Test{
public static void main(String arg[]){
Personne p1=new Personne ("Lya","Bedjaoui",30);
Personne p2=new Personne (p1);
new Personne (p1);
System.out.println("Nombre de personnes créées :"+
Personne.getNbPersonnes());
} // main
} //Test1
```

On obtient le résultat suivant :

Nombre de personnes créées : 3

14. Exercice 5

1. Écrire le code d'une classe Division qui offre deux méthodes diviser () :
   - La première est une méthode d'instance.
   - La deuxième est une méthode de classe.
2. Écrire le code d'une classe Test qui permet de tester les deux méthodes diviser ().

15. Les tableaux en Java

Comme tous les langages, Java permet de manipuler des tableaux mais nous verrons qu'il fait preuve d'originalité sur ce point [4]. En particulier, les tableaux sont considérés comme des objets et les tableaux à plusieurs indices s'obtiennent par composition de tableaux.

Considérons cette déclaration :

```
int t[];
```

Elle précise que t est destiné à contenir la référence à un tableau d'entiers. Vous constatez qu'aucune dimension ne figure dans cette déclaration et, pour

l'instant, aucune valeur n'a été attribuée à t. Cette déclaration est en fait très proche de celle de la référence à un objet.

On crée un tableau comme on crée un objet, c'est-à-dire en utilisant l'opérateur `new`. On précise à la fois le type des éléments, ainsi que leur nombre (dimension du tableau), comme dans :

```
t = new int[5]; // t fait référence à un tableau de 5 entiers
```

Cette instruction alloue l'emplacement nécessaire à un tableau de 5 éléments de type **`int`** et en place la référence dans t.

Les 5 éléments sont initialisés par défaut à une valeur "nulle" (**0** pour un **`int`**). Cette situation est illustrée par la Figure 2.1 :

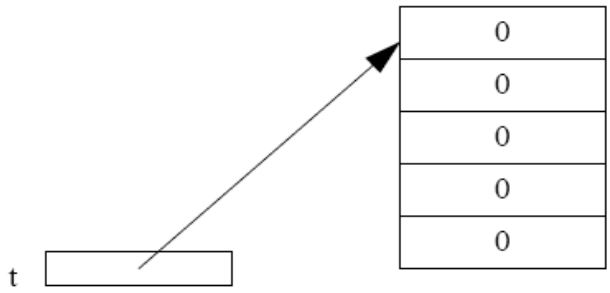


Figure 2.1 : Création d'un tableau en Java

La déclaration d'une référence à un tableau précise donc simplement le type des éléments du tableau. Elle peut prendre deux formes différentes ; par exemple, la déclaration précédente :

```
int t[];
```

peut aussi s'écrire :

```
int [] t;
```

En Java, les éléments d'un tableau peuvent être d'un type primitif ou d'un type objet. Par exemple, si nous avons défini le type classe Etudiant, ces déclarations sont correctes :

```
Etudiant tp[]; // tp est une référence à un tableau d'objets
de type Etudiant

Etudiant a, tp[], b;
/* a et b sont des références à des objets de type Etudiant,
tp est une référence à un tableau d'objets de type Etudiant
*/
```

Remarque : Lors de la déclaration d'une référence de tableau, on peut fournir une liste d'expressions entre accolades, comme dans :

```
int m, p;
...
int t[] = {1, m, m+p, 2*p, 12};
```

Cette instruction crée un tableau de 5 entiers ayant les valeurs des expressions mentionnées et en place la référence dans t. Elle remplace les instructions suivantes :

```
int m, p, t[];

.....

t = new int[5];
t[0] = 1; t[1] = m; t[2] = m+p; t[3] = 2*p; t[4] = 12;
```

15.1. Accès individuel aux éléments d'un tableau

On peut manipuler un élément de tableau comme on le fait en langage C. Rappelant que, le premier élément correspond à l'indice 0 (et non 1).

```
int t[] = new int[5];
.....

t[0] = 20; // place la valeur 20 dans le premier élément du
tableau t
.....

t[2]++; // incrémente de 1 le troisième élément de t
.....
System.out.println(t[4]);
//affiche la valeur du dernier élément de t
```

15.2. Affectation de tableaux

Java permet aussi de manipuler globalement des tableaux, par le biais d'affectations de leurs références.

Considérons ces instructions qui créent deux tableaux d'entiers en plaçant leurs références dans t1 et t2 :

```
int [] t1 = new int[3];
for (int i=0; i<3; i++) t1[i] = i;

int [] t2 = new int[2];
for (int i=0; i<2; i++) t2[i] = 10 + i;
```

La situation peut être schématisée par la Figure 2.2 :

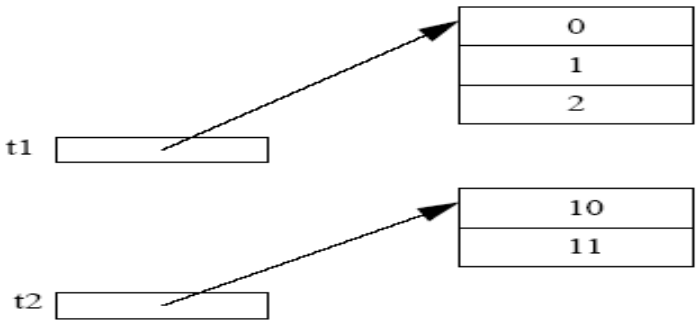


Figure 2.2 : Création des tableaux t1 et t2

Exécutons maintenant l'affectation :

```
t1 = t2;
```

Nous aboutissons à la situation de la Figure 2.3 :

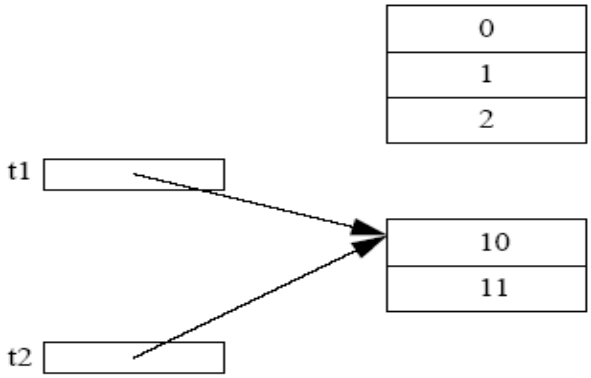


Figure 2.3 : Affectation des tableaux

Dorénavant, t1 et t2 désignent le même tableau. Ainsi, avec :

```
t1[1] = 5;
System.out.println(t2[1]);
```

on obtiendra l'affichage de la valeur 5.

Si l'objet que constitue le tableau de trois entiers anciennement désigné par t1 n'est plus référencé par ailleurs, il deviendra candidat au ramasse-miettes (voir la **Section 10** de ce chapitre).

15.3. La taille d'un tableau : length

Le champ length permet de connaître le nombre d'éléments d'un tableau de référence donnée :

```
int t[] = new int[5];
System.out.println("taille de t : " + t.length);
// affiche 5

t = new int[3];
System.out.println("taille de t : " + t.length);
// affiche 3
```

16. Exercice 6

Cet exercice vous servira de révision sur la programmation objet. Vous allez aussi comprendre le rôle d'une méthode appelée toString().

Qu'affiche le programme suivant ?

```
class Personne {
String nom;
String prenom;
Personne pere;
Personne mere;
Personne(String nom, String prenom, Personne pere, Personne
mere) {
this.nom = nom;
this.prenom = prenom;
this.pere = pere;
this.mere = mere;
}
Personne(String nom, String prenom) {
// appel du constructeur défini ci-dessus avec 4 paramètres
this(nom, prenom, null, null);
}
// la méthode toString()  ci-dessous retourne une chaîne de
caractères formée avec le nom et le prénom
public String toString() {
return nom + " " + prenom;
}
}

public class Test{
public static void main(String arg[]) {
// Tableau de deux personnes
Personne [] tabPers = new Personne[2];
```

```
tabPers[0] = new Personne("nom0", "prenom0");
Personne p0 = tabPers[0];
System.out.println("Personne p0 : " + p0);
tabPers[1] = new Personne("nom1", "prenom1");
Personne p1 = tabPers[1];
System.out.println("Personne p1 : " + p1.toString());
}
}
```

Remarques :

1. Dans une classe, le mot-clé « this » s'il est suivi de parenthèses, this**()** signifie plutôt l'appel à un autre constructeur de la classe. Si on l'utilise, il doit être la première instruction du constructeur.

2. Toute classe possède une méthode spéciale dont la signature est String toString(), qui peut être utilisée pour convertir un objet de cette classe en String. On peut alors l'utiliser pour afficher les informations contenues dans un objet comme ceci :

   Exemple d'appel explicite :

   ```
   System.out.println("Informations de l'objet :" +
   nomObjet.toString());
   ```

   Exemple d'appel implicite

   ```
   System.out.println("Informations de l'objet:" +
   nomObjet);
   ```

   Si on ne définit pas la méthode `toString ()` dans une classe, l'appel de cette méthode donnera des informations comme le nom de la classe et l'**adresse de l'objet en mémoire**.

L'exercice 8 du chapitre 5 présente un autre exemple d'utilisation de la méthode `toString ()`.

17. Entrées/Sorties

17.1. Affichage des informations

Nous avons déjà vu qu'on pouvait afficher par la méthode `println/print`.

En effet, Java prévois toute une classe `String` pour la manipulation des chaines de caractères.

Java prévoit plus d'une dizaine de constructeurs pour la classe String [6], comme par exemple un constructeur auquel il faut passer en paramètre un tableau de caractères.

Exemple :

```
public class Test {
 public static void main(String[] args) {
 char[] hello = {'h','e','l','l', 'o',  'W', 'o', 'r', 'l', 'd',
'!'} ;
 String H = new String (hello);
 System.out.println(H);
 }
}
```

Le résultat d'exécution est :

```
helloWorld!
```

La classe `String` dispose d'un grand nombre de méthodes.

- La méthode `length` permet de déterminer la langueur d'un string, tel qu'illustré dans cet exemple :

Exemple :

```
public class Test {
 public static void main(String[] arg) {
 String ch = "bonjour";
 int n = ch.length() ;
 System.out.println(n);
 }
}
```

Le résultat d'exécution est :

```
7
```

> ***Remarque*** :
> Contrairement à ce qui se passait pour les tableaux où *length* désignait un champ, nous avons bien affaire ici à une méthode. **Les parenthèses à la suite de son nom sont donc indispensables.**

- La concaténation de deux strings peut se faire en utilisant la méthode `concat` (voir exemple a) ou en additionnant les deux strings (voir exemple b).

  a) `String1.concat (string2) ;`

  b) `String helloWorld = « Hello » + « World » ;`

Exemple :

```
public class Test {
public static void main(String[] arg) {
String H = "Hello";
String W = "World";
String N = "!";
String hl = "Hello" + "World";
```

```
System.out.println(hl);
String S = H.concat(N);
System.out.println(S);
}
}
```

Le résultat d'exécution est :

```
HelloWorld
Hello!
```

- la méthode `charAt()` permet de récupérer un caractère à partir de son indice.

Exemple

```
public class Test {
public static void main(String[] arg) {
String nom = "lamia";
System.out.println("le caractère à la position 3 est:" +
nom.charAt(3));
}
}
```

Le résultat d'exécution est :

```
le caractère à la position 3 est:i
```

- Pour extraire un string d'un string, il faut employé la méthode substring qui existe en deux versions :
  a) `substring (index1, index 2)` qui retourne la partie du string comprise entre les positions `index1` et `index2`.
  b) `substring (index1)` qui retourne la partie de string comprise depuis index1 à la fin du string.

Exemple :

```
public class Test {
 public static void main(String[] arg) {
 String helloWorld = "hello World ! " ;
 System.out.println(helloWorld.substring (0, 4));
 System.out.println(helloWorld.substring (6, 10));
 System.out.println(helloWorld.substring (6));
 }
}
```

Le résultat d'exécution est :

```
hell
Worl
World !
```

- La classe String possède d'autres méthodes : la méthode `toLowerCase` (resp, `toUpperCase`) retournant une copie du string convertit en minuscules (resp, en majuscule), La méthodes replace, remplaçant un caractère par un autre à l'intérieur d'un string, etc.

17.2. Saisie à partir du clavier

Java offre plusieurs classes pour la saisie de données. Dans ce cours, nous nous contentons de la classe Java : Scanner

La classe Scanner facilite la lecture dans un flux. Elle est particulièrement utile pour réaliser une lecture de données à partir du clavier.

Pour pouvoir utiliser la classe Scanner il est indispensable d'écrire en début de fichier (avant public class) : `import java.util.Scanner;`

Parce que cette classe est dans ce package. Le mot clé « `import` » est l'équivalent de « `include` » que vous avez l'habitude d'utiliser en langage C.

La saisie se fait en créant d'abord un objet de la classe Scanner par l'instruction

```
Scanner scan=new Scanner (System.in);
```

La méthode next() de cette classe bloque l'exécution jusqu'à la lecture de données et les renvoie sous la forme d'une chaîne de caractères. Il suffit de créer un objet Scanner avec en argument le flux à lire (System.in), puis d'appeler une méthode nextXXX() selon le type primitif XXX à lire. Précisant que la méthode : nextInt() permet de lire un entier, nextFloat() permet de lire un réel, nextDouble() permet de lire un double, next() permet de lire une chaîne de caractères sans caractères blanc, nextLine() permet de lire une chaîne de caractères y compris le caractère blanc entre les mots jusqu'à la fin de la ligne.

Exemple :

```
import java.util.Scanner;

public class LectureEnJava {
public static void main(String[] args) {
Scanner scan = new Scanner(System.in);
int nombre;
System.out.print("Entrez un nombre entier: ");
nombre = scan.nextInt();
System.out.println(nombre);
}
}
```

L'exercice 10 du chapitre 5 présente un bon exemple de lecture et de manipulation de chaine de caractères.

18. Conclusion

À l'issue de ce chapitre, l'étudiant comprend mieux la programmation orientée objet. Il aura la maitrise d'instanciation d'objets, de création des tableaux, de traitement de chaines de caractères, de lecture en Java, etc.

Chapitre 3 : L'héritage en Java

1. Introduction

Ce chapitre a pour objectif d'inculquer aux étudiants un autre concept très important en POO qui est l'héritage. Ce concept a comme principal atout la **réutilisation du code** que l'étudiant découvrira par des exemples présentés dans ce chapitre.

2. Définition

L'héritage permet de définir une nouvelle classe à partir d'une classe existante (qu'on réutilise), à laquelle on ajoute de nouvelles données et de nouvelles méthodes. La nouvelle classe est dite classe dérivée (ou héritée, fille, sous-classe) et la classe d'origine est dite classe de base (ou classe mère, super-classe).
La notation graphique de l'héritage (en langage de modélisation objet UML) entre deux classes est représentée par une flèche à la tête en forme de triangle blanc [2].

Exemple : L'exemple de la Figure 3.1 représente la classe Personne qui hérite de la classe EtreVivant.

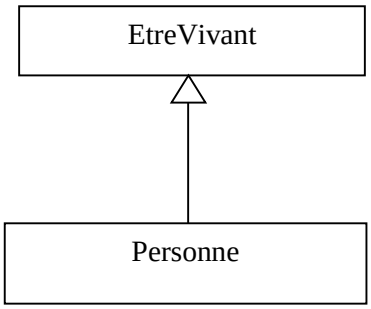


Figure 3.1 : Exemple d'héritage simple

Il existe deux types d'héritage en POO, l'héritage simple et l'héritage multiple :

L'héritage simple c'est lorsqu'une classe donnée ne peut hériter que d'une seule super classe à la fois (voir Figure 3.1). Java fournit le mot-clé `extends` pour faire hériter une classe d'une superclasse.

Par contre, dans l'héritage multiple (voir Figure 3.2) il y a deux ou plusieurs super-classes pour une même sous-classe. **Java ne permet pas l'héritage multiple, mais il permet l'implémentation d'interfaces multiples.**

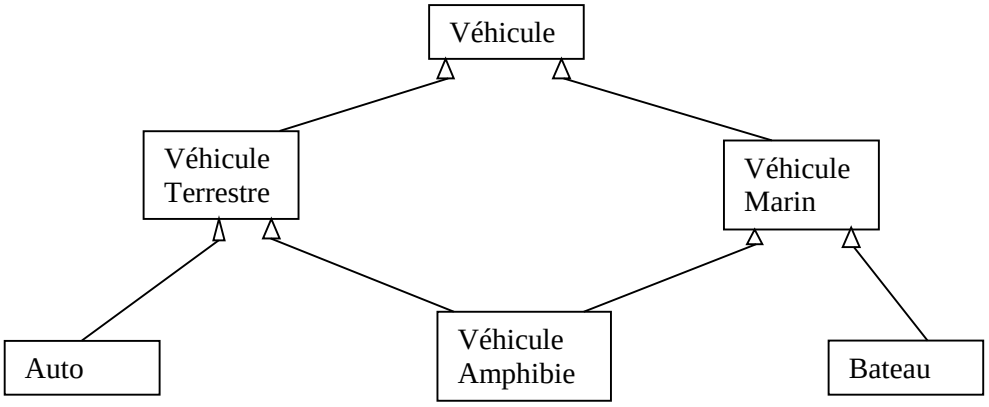


Figure 3.2 : Exemple d'héritage multiple

Dans l'exemple de la figure 3.2, la classe « Véhicule Amphibie » hérite à la fois de la classe « Véhicule Terrestre » et à la fois de la classe « Véhicule Marin ».

3. Encapsulation dans l'héritage

Le concept d'encapsulation a été défini dans la section 4 du chapitre 2 où deux modificateurs ont été présentés (public et private). L'encapsulation est aussi applicable pour l'héritage à travers le modificateur *« protected »*. Précisant que, les méthodes et les variables déclarées *« protected »* ne sont accessibles que par les méthodes de la classe et des sous-classes du package de l'objet de définition.

4. Constructeurs des classes hérités (super())

Le mot clé super est utilisé pour l'appel des constructeurs de la classe mère. La Section 5.1 ci-dessous présente un exemple par l'instruction :

```
super(P,N,age);
```

Il peut que l'on ait besoin d'appeler une méthode de la classe mère. Dans ce cas, le nom de la méthode est préfixé par le mot-clé super, comme suit :

```
super.nomDeMethode ;
```

Remarques :

- La super classe (racine) du langage Java est la classe Object. Si le développeur ne précise pas de classe mère dans la déclaration d'une classe, alors la classe hérite implicitement d'Object. Nous pouvons créer à tout moment une instance de Object par l'instruction.

```
Object monObj = new Object();
```

- L'opérateur « == » en Java sert à comparer les références. Il ne faut jamais l'utiliser pour comparer des objets. La comparaison d'objets se fait grâce à la méthode equals héritée de la classe Object. la méthode equals(Object o), pour une classe X est définit généralement comme suit :

```
public boolean equals(Object o){

if(this==o) return true;
if(!(o instanceof X) return false;
...

 }
```

Noter que, l'opérateur `instanceof` permet de savoir à quelle classe appartient une instance. Il suffit d'écrire :

« objet » `instanceof` « classe ».

5. L'héritage par l'exemple

Supposons qu'on veuille créer une classe Enseignant : un enseignant est une personne particulière. Il a des attributs qu'une autre personne n'aura pas : son grade par exemple. Mais, il a aussi les attributs de toute personne : prénom, nom et âge. Un enseignant fait donc pleinement partie de la classe Personne, mais a des attributs supplémentaires. Le mécanisme d'héritage nous permet justement de reprendre l'acquis de la classe Personne qu'on adapterait au caractère particulier des enseignants. Pour exprimer que la classe Enseignant hérite des propriétés de la classe Personne, on écrira :

```
public class Enseignant extends Personne
```

Personne est appelée la classe mère (ou super classe) et Enseignant la classe fille (ou sous classe). Un objet enseignant a toutes les qualités d'un objet personne : il a les mêmes attributs et les mêmes méthodes. Ces attributs et méthodes de la classe mère ne sont pas répétés dans la définition de la classe fille : on se contente d'indiquer les attributs et méthodes rajoutés par la classe fille, ce qui illustre bien le principe de **réutilisation de code**.

Nous supposons que la classe Personne est définie comme suit :

```
public class Personne{
private String prenom;
private String nom;
private int age;
public Personne(String P, String N, int age){
this.prenom=P;
this.nom=N;
this.age=age;
}
public String identite()
{
return "personne("+prenom+","+nom+","+age+")";
}
```

```
public String getPrenom()
{
return prenom;
}
public String getNom()
{
return nom;
}
public int getAge()
{
return age;
}
public void setPrenom(String P)
{
this.prenom=P;
}
public void setNom(String N)
{
this.nom=N;
}
public void setAge(int age)
{
this.age=age;
}
}
```

La classe Enseignant est définie comme suit :

```
class Enseignant extends Personne{
// attributs
private String grade;
// constructeur
public Enseignant(String P, String N, int age, int grade){
super(P,N,age);
this.grade=grade;
}
```

```
}
```

La méthode identifie a été légèrement modifiée pour rendre une chaîne de caractères identifiant la personne et porte maintenant le nom identite.

Ici la classe Enseignant rajoute aux méthodes et attributs de la classe Personne :

- Un attribut grade qui est le grade auquel appartient l'enseignant dans le corps des enseignants (ex. MAB, MAA, MCB, MCA, PROF, etc.).
- Un nouveau constructeur permettant d'initialiser tous les attributs d'un enseignant.

5.1. Construction d'un objet enseignant

Le constructeur de la classe Enseignant est le suivant :

```
// constructeur
public Enseignant(String P, String N, int age, int grade){
super(P,N,age);
this.grade = grade;
}
```

L'instruction `super(P,N,age)` est un appel au constructeur de la classe mère(la classe Personne). On sait que ce constructeur initialise les champs prenom, nom et age de l'objet Personne. Cela paraît bien compliqué et on pourrait préférer écrire :

```
// constructeur
public Enseignant(String P, String N, int age, int grade)
{
this.prenom=P;
this.nom=N
this.age=age
this.grade=grade;
}
```

C'est impossible. La classe Personne a déclaré privés (private) ses trois champs prenom, nom et age. Seuls des objets de la même classe ont un accès

direct à ces champs. Tous les autres objets, y compris des objets fils comme ici, doivent passer par des méthodes publiques pour y avoir accès. Cela aurait été différent si la classe Personne avait déclaré protégés (**protected**) les trois champs : elle autorisait alors des classes dérivées à avoir un accès direct aux trois champs.

Dans cet exemple, utiliser le constructeur de la classe mère était donc la bonne solution et c'est la méthode habituelle : lors de la construction d'un objet fils, on appelle d'abord le constructeur de l'objet parent puis on complète les initialisations propres à l'objet fils.

Essayons le programme suivant :

```
// import Enseignant
public class Test{
public static void main(String arg[]){
System.out.println(
new Enseignant("Lya","Bejaoui",30,"MAA").identite());
}
}
```

Ce programme permet de créer un objet enseignant (new) et de l'identifier. La classe Enseignant n'a pas de méthode identité, mais sa classe parent en a une qui de plus est publique : elle devient par héritage une méthode publique de la classe Enseignant.

Le résultat de l'exécution sera :

```
personne(Lya,Bejaoui,30)
```

6. Exercice 7

Qu'affiche le programme suivant ?

```
public class Cookie {
protected boolean bite() {
System.out.println("bite");
return true;
```

```
}
}
public class ChocolateChip extends Cookie {
public ChocolateChip() {
System.out.println("ChocolateChip constructor");
}
public static void main(String arg[]) {
ChocolateChip x = new ChocolateChip();
System.out.println(x.bite());
}
}
```

7. Hiérarchie de classes

En Java, le mécanisme d'héritage permet de définir une hiérarchie de classes comprenant toutes les classes [8].

La notation graphique de la hiérarchie des classe Personne et Enseignant est comme suit :

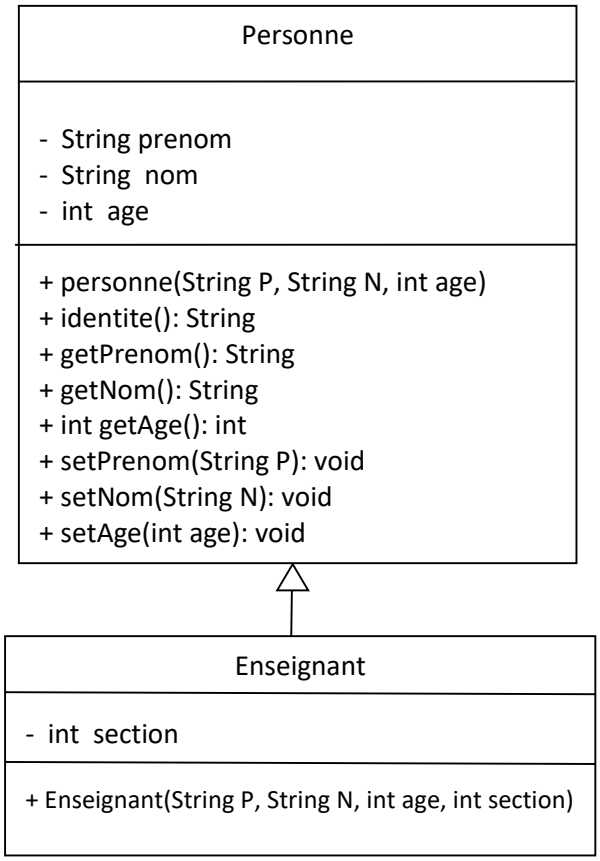


Figure 3.3 : Diagramme de classes

8. Transtypage et héritage

Un transtypage (ou casting) permet de convertir une donnée d'un type de base en un autre. Il s'écrit en faisant précéder la valeur à convertir par le nom du type cible de la conversion, écrit entre parenthèses (Exemple, les lignes 7 et 8 de l'exercice 8 ci-dessous).

Il existe aussi la conversion de classes, dans ce cas le « cast » est le fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou réel de l'objet. En Java, les seuls casts autorisés entre classes sont les casts entre **classe mère** et **classe fille**.

Exemple : Si nous avons défini une classe Animal et que nous définissons une classe Chien qui hérite de cette classe Animal, nous pourrons alors convertir le chien en animal comme suit :

```
Chien dog1 = new Chien();
Animal animal = (Animal)dog1;
```

9. Exercice 8

Cet exercice a pour objectif de rappeler le principe de base du transtypage. Qu'affiche le programme suivant ?

```
01 public class Test
02 {
03 public static void main(String arg[])
04 {
05  double x = 1.23;
06  double y = 3.9999999;
07  int k = (int)x;    //k vaut 1
08  k = (int)y;        //k vaut 3
09  switch(k + 3)
10  {
11  case 6: y = 1;
```

```
12  break;
13  default: y += 1;
14  }
15  System.out.println(y);
16  }
17  }
```

10. Limitation de l'héritage

Bien que l'héritage nous permette de réutiliser le code existant, nous avons parfois besoin de définir des limites d'extensibilité pour diverses raisons ; le mot-clé final nous permet de faire exactement cela [5].

Le mot-clé final signifie pour les classes et les méthodes ce qui suit :

- Les classes déclarées « *final* » ne peuvent pas avoir de sous-classe.
- Les méthodes déclarées « *final* » ne peuvent pas être redéfinies dans une sous-classe.

> ***Remarque*** *:* Une variable déclarée « *final* » est en fait une constante. Par *convention*, les noms des *constantes sont* entièrement en *majuscules.*

11. Polymorphisme

Le polymorphisme désigne la faculté de prendre plusieurs formes. On peut distinguer deux expressions du polymorphisme :

11.1. Surcharge de méthodes (polymorphisme statique)

La surcharge survient lorsque deux méthodes ou plus dans une classe ont le même nom de méthode mais des paramètres différents. Le choix de la forme appropriée est déterminé au moment de la compilation à partir de la déclaration de l'objet polymorphe.

Exemple :

```
class MaClasse
{
   public void maMethode(int i) { ... }
   public void maMethode(float f) { ... }
}
```

Ici, la méthode *maMethode* est **surchargée**.

11.2. Redéfinition de méthodes (polymorphisme dynamique)

La redéfinition signifie avoir deux méthodes avec le même nom et les mêmes paramètres, l'une des méthodes est dans la classe mère et l'autre dans la classe fille. Le choix de la forme appropriée se fonde sur le type de l'objet au moment de l'exécution.

Exemple :

```
public class A{
public void direBonCourage()
{
System.out.println("Bon Courage");
}
}
public class B extends A
{
public void direBonCourage()  // redefinition
{
System.out.println("merci, Bon Courage à toi aussi");
}
public void direBonCourage(String  msg)  // surcharge
{
System.out.println("ok, "+msg+" !");
}
}
```

```
public class Test{
public static void main(String arg[]){
A a = new A();
B b = new B();
A ab = new B();
B c = new B();
a.direBonCourage();
b.direBonCourage();
ab.direBonCourage();
c.direBonCourage("merci");
}
}
```

Ici, la méthode `direBonCourage()` de la classe B est une redéfinition de la méthode `direBonCourage()` de la classe A. En plus, on a une surcharge de la méthode `direBonCourage()` par la méthode direBonCourage(String msg).

Le résultat de ce programme est :

```
Bon Courage
merci, Bon Courage à toi aussi
merci, Bon Courage à toi aussi
ok, merci !
```

12. Exercice 9

Qu'affiche le programme suivant ?

```
public class A {
public void doIt() {
System.out.println("B");
}
}
public class B extends A { }
public class C extends B {
public void doIt() {
```

```
System.out.println("D");
}
}

public class Tester {
public static void main(String arg[]) {
A x = new A();
A y = new C();
B z = new C();
B e = new B();
A f = new B();
x.doIt();
y.doIt();
z.doIt();
e.doIt();
f.doIt();
}
}
```

13. Exercice 10

1. Créez une classe nommé Liquide contenant seulement une méthode imprimer () qui affiche : ”je suis un liquide”
2. Créez deux classes dérivées de la classe Liquide, les classes Cafe et Lait, dont les méthodes imprimer () affichent respectivement ”je suis du Café”, ”je suis du Lait”.
3. Créez une classe Tasse ayant un attribut `x` de la classe Liquide et une méthode AjouterLiquide (Liquide li) et une méthode imprimer ().
4. Dans la classe principale, créer un tableau de Tasses qui contient différents liquides.

14. Classes abstraites

Une classe abstraite est une classe qui ne permet pas d'instancier des objets. Elle ne peut servir que de classe de base pour une dérivation. Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée. Mais, on peut aussi trouver des méthodes dites abstraites, c'est-à-dire dont on ne fournit que la signature et le type de la valeur de retour [4].

> ***Remarque*** : Dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite.

Exemple :

```
abstract class A {
public void f() { ..... }
// f est définie dans A
public abstract void g(int n);
// g n'est pas définie dans A, on n'en a fourni que l'en-tête
}
```

3.1. Quelques règles des classes abstraites

a) Une classe dérivée d'une classe abstraite n'est pas obligée de définir toutes les méthodes abstraites de sa classe de base. Dans ce cas, elle reste simplement abstraite (il est quand même nécessaire de mentionner abstract dans sa déclaration) :

Exemple :

```
abstract class A {
public abstract void f1();
```

```
public abstract  void f2(char c);
.....
}

abstract class B extends A {   // abstract obligatoire ici
public void f1() { ..... }     // définition de f1
.....                      // pas de définition de f2
}
```

Ici, B définit f1, mais pas f2. La classe B reste abstraite.

b) Une méthode abstraite doit obligatoirement être déclarée public.

c) Une classe déclarée final ne peut pas contenir de méthodes abstraites car elle ne peut pas être sous classée.

15. Exercice 11

1. Écrire une classe abstraite « Volaille ». Une volaille est un oiseau élevé dans une ferme, caractérisée par un « numéro » et un « poids ». Ajouter à cette classe un constructeur et une méthode « ChangePoids » qui permet de modifier son poids. Ecrire deux autres méthodes abstraites, l'une retourne son prix, et l'autre si la volaille est assez grosse pour être abattue. En effet, le prix et l'estimation de la grosseur dépend du type de la volaille (Poulet, canard, dinde, etc.).
2. Écrire une classe « Poulet » qui dérive de la classe « Volaille ». Tous les poulets ont le même prix de vente au kilo, et le même poids d'abattage. Cependant, pour des raisons de marché, le prix de vente et le poids d'abattage peuvent changer.
3. Écrire la classe principale qui permettra de tester la classe « Poulet ».

16. Interfaces

Si l'on considère une classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes), on aboutit à la notion d'interface [4]. En

effet, une interface définit les en-têtes d'un certain nombre de méthodes, ainsi que des constantes. Cependant, cette dernière notion se révèle plus riche qu'un simple cas particulier de classe abstraite.

La définition d'une interface se présente comme celle d'une classe. On y utilise simplement le mot-clé interface à la place de class :

Exemple

```
public interface I {
void f(int n);
// en-tête d'une méthode f (public abstract facultatifs)

void g();
// en-tête d'une méthode g (public abstract facultatifs)
}
```

Dans la définition d'une interface, on ne peut trouver que des en-têtes de méthodes ou des constantes. Par essence, les méthodes d'une interface sont abstraites (puisqu'on n'en fournit pas de définition) et publiques. Néanmoins, il n'est pas nécessaire de mentionner les mots-clés « `public` » et « `abstract`» [4]. Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot-clé `implements`, comme dans :

```
class A implements I {
// A doit définir les méthodes f et g prévues dans l'interface
I
}
```

Une même classe peut implémenter plusieurs interfaces :

Exemple :

```
public interface I1 {void f();}
public interface I2 {int h();}
```

```
class A implements I1, I2 {
// A doit obligatoirement définir les méthodes f et h prévues
dans I1 et I2
}
```

17. Exercice 12

Certain animaux peuvent crier, d'autres sont muets. On représentera le fait de crier au moyen d'une méthode affichant à l'écran le cri de l'animal.

1. Ecrire une interface contenant la méthode permettant de crier.
2. Ecrire les classes des chats, des chiens et des lapins (qui sont muets).
3. Ecrire une classe Animaux contenant un tableau pour les animaux qui savent crier, le remplir avec des chiens et des chats, puis faire crier tous ces animaux.

18. Conclusion

À travers ce chapitre, l'étudiant découvre qu'il existe un concept orienté objet permettant de concrétiser la réutilisabilité du code, qui est l'héritage. Ce mécanisme englobe des concepts propres à lui, tels que : Les mots-clés «protected », « final » et « super », la hiérarchie de classes, le polymorphisme statique, le polymorphisme dynamique, les classes abstraites et les interfaces. La résolution des exercices de ce chapitre permettra à l'étudiant de consolider son apprentissage du langage Java.

Chapitre 4 : Solutions des exercices du cours

1. Solution de l'exercice 1 :

Lorsqu'on déclare la variable *p1* par

Personne p1=new Personne("Lya","Bedjaoui",30);

p1 référence l'objet *Personne("Lya","Bedjaoui",30)* mais n'est pas l'objet lui-même. En C, on dirait que c'est un pointeur, c.à.d. l'adresse de l'objet créé. Si on écrit ensuite :

p1=null

Ce n'est pas l'objet *Personne("Lya","Bedjaoui",30)* qui est modifié, c'est la référence *p1* qui change de valeur. L'objet *Personne("Lya","Bedjaoui",30)* sera "perdu" s'il n'est référencé par aucune autre variable.

Lorsqu'on écrit :

Personne p2=p1;

on initialise le pointeur *p2* : il "pointe" sur le même objet (il désigne le même objet) que le pointeur *p1*.

Ainsi si on modifie l'objet pointé (ou référencé) par *p1*, on modifie celui référencé par *p2*.

Lorsqu'on écrit :

Personne p3=new Personne(p1);

il y a création d'un nouvel objet, **copie** de l'objet référencé par *p1*. Ce nouvel objet sera référencé par *p3*. Si on modifie l'objet pointé (ou référencé) par *p1*, on ne modifie en rien celui référencé par *p3*. C'est ce que montrent les résultats obtenus.

2. Solution de l'exercice 2

```
public class Personne{
private String prenom;
private String nom;
private int age;
public Personne(String P, String N, int age){
this.prenom=P;
this.nom=N;
this.age=age;
}
// getters
public String getPrenom()
{
return prenom;
}
public String getNom()
{
return nom;
}
public int getAge()
{
return age;
}
// setters
public void setPrenom(String P)
{
this.prenom=P;
}
public void setNom(String N)
{
this.nom=N;
}
public void setAge(int age)
{
```

```
this.age=age;
}
}
```

La classe principale est à compléter par l'étudiant en créant plusieurs objets personnes et en invoquant les getters et setters de son choix.

3. Solution de l'exercice 3

Les erreurs

1.

Oubli de tous les ";" (lignes 3, 7, 8, 16, 23, 24, 32, 33, 34, 35, 36, 37)

Correction. Rajouter le ";" à la fin de chaque ligne.

2.

ligne 30. Il faut mettre une majuscule à string : Java est sensible à la casse.

3.

Le mot "static" est en trop lignes 05, 14, 21.

Correction. Le supprimer.

4.

Ligne 05. Le mot clé "constructor" n'existe pas en Java.

Correction. Il faut le remplacer par le nom de la classe, soit ici Vecteur

5.

Ligne 28. Le mot "classe" est mal orthographié.

Correction. Écrire "class"

6.

Ligne 16. Les deux "=" (symboles de l'affectation) doivent être remplacés par des "==",.

7.

La bonne traduction du "et" logique n'est pas "and" mais "&&".

8.

Ligne 35, mauvaise utilisation de "ajoute"

Correction. Écrire "v1.ajoute(v2)" (ou "v2.ajoute(v1)", ce qu'a moins de sens ici, puisque c'est v1 qui est affiché)
9.

Ligne 38. Il manque une accolade "}".

10.

Ligne 3. La bonne traduction de "réel" n'est pas "real" mais double".

11.

Ligne 36 et 37. La bonne traduction de "Ecran" n'est pas "screen" mais System.out".

Remarque : On pourra de plus remarquer qu'en ligne 01 il manque le mot public. Ce mot est conseillé mais son oubli n'est pas grave car le programme fonctionne sans ce mot.

En effet, la non écriture de ce mot clé signifie que les variables et les méthodes de cette classe sont accessible dans le même package alors son écriture permet l'accessibilité à tous les packages.

La bon programme est :

```
class Vecteur
{
double  x, y;
public  Vecteur(double abscisse, double ordonnee)
{
x = abscisse;
y = ordonnee;
}
// ajoute le vecteur v au vecteur
public  void ajoute(Vecteur v)
{
x = x + v.x;
y = y + v.y;
}
}

class EssaiVecteur
```

```
{
public static void main(String arg[])
{
Vecteur v1, v2;
v1 = new Vecteur(3.1, 5.2);
v2 = new Vecteur(2.4, 3.5);
v1.ajoute(v2);
System.out.println("Vecteur final = ");
System.out.println("abscisse = " + v1.x + " ordonnee = " +
      v1.y) ;
}
}
```

Le résultat de l'exécution est :

```
Vecteur final =
abscisse = 5.5 ordonnee = 8.7
```

4. Solution de l'exercice 4

Le programme affiche

```
nbre = 100 decal = 0
```

5. Solution de l'exercice 5

```
public class Division {
private double a, b;
Division(double a, double b) {
this.a = a;
this.b = b;
}
public double diviser() {
return (this.a / this.b);
}
public static double diviser(double a, double b) {
return (a / b);
```

```
}
}

public class TestDivision {
public static void main(String[] args) {
Division d1 = new Division(7, 7);
double resultat = d1.diviser();
System.out.println(resultat);
double result = Division.diviser(7, 7);
System.out.println(result);
}
}
```

Le résultat d'exécution sera :

```
1.0
1.0
```

6. Solution de l'exercice 6

Le programme affiche

```
Personne p0 : nom0 prenom0
Personne p1 : nom1 prenom1
```

7. Solution exercice 7

Le programme affiche

```
ChocolateChip constructor
bite
true
```

8. Solution exercice 8

Le programme affiche

```
1.0
```

9. Solution exercice 9

Le programme affiche

B

D

D

B

B

10. Solution de l'exercice 10

```
public class Liquide {
public void imprimer() {
System.out.println("je suis un liquide");
}
}
class Cafe extends Liquide {
public void imprimer() {
System.out.println("je suis du cafe");
}
}
class Lait extends Liquide {
public void imprimer() {
System.out.println("je suis du lait");
}
}

class Tasse {
private Liquide x;
public void AjouterLiquide(Liquide li) {
this.x = li;
}
public void imprimer() {
```

```
x.imprimer();
}
}
public class Main {
public static void main(String[] args) {
Tasse[] lestasses;
lestasses = new Tasse[3];
for(int i=0 ;i<3 ;i++)
lestasses[i]=new Tasse() ;
lestasses[0].AjouterLiquide(new Cafe());
lestasses[1].AjouterLiquide(new Lait());
lestasses[2].AjouterLiquide(new Liquide());
for(int i=0 ;i<3 ;i++)
lestasses[i].imprimer();
}
}
```

Ce programme affiche :

```
je suis du cafe
je suis du lait
je suis un liquide
```

11. Solution de l'exercice 11

```
abstract class Volaille {
double poids;
int identite;
Volaille(double p, int i) {
poids = p;
identite = i;
}
void changePoids(double np) {
poids = np;
}
```

```
abstract double prix();
abstract boolean assezGrosse();
}

class Poulet extends Volaille {
double prixAuKilo = 1.0;
double poidsAbattage = 1.2;
Poulet(double p, int i) {
super(p, i);
}
void changePrix(double x) {
prixAuKilo = x;
}
void changePoidsAbattage(double x) {
poidsAbattage = x;
}
double prix() {
return poids * prixAuKilo;
}
boolean assezGrosse() {
return poids >= poidsAbattage;
}
}

public class Main {
public static void main(String[] args) {
Poulet p1 = new Poulet(3, 1);
Poulet p2 = new Poulet(3.5, 2);
double prix1 = p1.prix();
System.out.println(prix1);
double prix2 = p2.prix();
System.out.println(prix2);
p2.changePoids(0.5);
p2.changePrix(2.5);
```

```
double prix3 = p2.prix();
System.out.println(prix3);
boolean a = p1.assezGrosse();
boolean b = p2.assezGrosse();
System.out.println(a);
System.out.println(b);
}
}
```

Ce programme affiche :

```
3.0
3.5
1.25
true
false
```

12. Solution de l'exercice 12

```
interface Criant{
void crier();
}
class Chat implements Criant{
public void crier(){
System.out.println("maou");
}
}
class Chien implements Criant{
public void crier(){
System.out.println("wouf");
}
}

class Lapin{
```

```
// Cette classe n’implémente pas l’interface parce que le
lapin n’a pas de son
public void froncerDuNez(){
System.out.println("lapin");
/* Cette méthode n’est pas demandée dans l’énoncer de
l’exercice */
}
}

public class Animaux{
public static void main(String[] args){
Criant[] tab = new Criant[4];
tab[0] = new Chat();
tab[1] = new Chien();
tab[2] = new Chat();
tab[3] = new Chien();
for (int i=0; i<4; i++){
tab[i].crier();
}
}
}
```

Ce programme affiche :

```
maou
wouf
maou
wouf
```

Chapitre 5 : Tests de connaissance

3.1. Exercice 1

Soit le programme java suivant

```
class TableauParam {
static void listerTableau(int[] note, int[] coef) {
System.out.println("Note\tCoefficient");
for (int i=0; i < note.length; i++) {
System.out.println(note[i] + "\t\t" + coef[i]);
}
}
static void modifCoeff(int[] coeff, int num, int valeur) {
coeff [num] = valeur;
}
public static void main(String arg[]) {
int[] coef = {1, 3, 2};
int[] note = {10, 12, 8};
listerTableau(note, coef);
modifCoeff(coef, 1, 4);
listerTableau(note, coef);
}
}
```

❖ Quel est le résultat de l'exécution de ce programme ?

3.2. Solution de l'exercice 1

Le programme affiche :

```
Note  Coefficient

10          1

12          3

8           2
```

```
Note  Coefficient
10          1
12          4
8           2
```

3.3. Exercice 2

Quel est le résultat de l'exécution de ce programme.

```
public class nom {
 public String affiche()
 {
 return "family name";
 }
}

public class nomplus extends nom {
 public String afficheplus()
 {
 return"first name";
 }
 public String affiche1()
 {
 return super.affiche();
 }
}

public class plus extends nomplus {
 public String afficheplusplus()
 {
 return "age";
 }
}
```

```
public class Test {
 public static void main(String arg[]) {
 nomplus m=new nomplus();
 System.out.println(m.afficheplus());
 System.out.println(m.affiche1());
 plus p=new plus();
 System.out.println(p.afficheplusplus());
 }
}
```

3.4. Solution de l'exercice 2

Le programme affiche :

```
first name
family name
age
```

3.5. Exercice 3

Soit le code suivant :

```
class POINT
{
real x, y;
constructor(double x1,double y1)
{
x=x1;
y=y1;
}
}

class POINTC extend POINT
{
```

```
constante static integer green=0, white=1;
int c;
POINTC(double x0, double y0, int col)
{
Class_mere(x0, y0);
c=col;
}
}

class COL
{
public static void main (String [] args)
{
Screen.out.println(" la couleur est blanche");
}
}
```

1. Trouver les erreurs qui sont glissées dans ces lignes de codes.
2. Expliquer le rôle de chaque classe.

3.6. Solution de l'exercice 3

Les erreurs sont mises en relief :

```
class POINT
{
real x, y;
constructor(double x1, double y1)
{
x=x1;
y=y1;
}
}
```

```
class POINTC extend POINT
{
constante static integer green=0, white=1;
int c;
POINTC(double x0, double y0, int col)
{
 Class_mere(x0, y0);
 c=col;
 }
 }
class COL
{
public static void main (String arg[])
{
Screen.out.println("la couleur est blanche");
}
}
```

Le code corrigé

```
class POINT {
double x, y;
POINT(double x1, double y1) {
x = x1;
y = y1;
}
}
class POINTC extends POINT
{
final static int green=0, white=1;
int c;
```

```
POINTC(double x0, double y0, int col)
{
super(x0, y0);
c=col;
}
}

class COL
{
public static void main(String arg[])
{
System.out.println("la couleur est blanche");
}
}
```

2. **Le rôle de ces trois classes**

a) la classe POINT, comporte les deux attributs x et y de type double et un constructeur.

b) la classe POINTC **extends** POINT : la classe POINTC hérite de la classe POINT. La conséquence de cette héritage est que les deux attributs x et y de la classe POINT appartiennent aussi à la classe POINTC. On peut si on le veut ajouter des attributs supplémentaires à la classe POINT. C'est ce qui est fait avec l'entier c et les deux constantes de type entiers green et white.

super(x0, y0) : fait appel au constructeur de la super classe ayant deux arguments de type doubles

c) la classe COL permet seulement d'afficher la chaîne « la couleur est blanche » via l'instruction : System.*out*.println("la couleur est blanche") ;

Les deux constructeurs ne sont pas utilisés dans la classe COL, on peut les utilisé dans l'instanciation des objets. Sachant que,

cela n'est pas une erreur parce que le programmeur pourra utilisés les constructeur à tout moment, selon les besoins.

Le mot clé static n'est pas une faute, les constantes peuvent être déclarées static pour désigner des variables globales à toutes les instances d'objet.

3.7. Exercice 4

Qu'affiche le programme suivant ?

```
class A {
int valeur;
A(int v) {
valeur = (byte)v;
}
int getValeur() {
return valeur;
}
}

class Test {
public static void main(String arg[]) {
A[] x = new A[2];
for (int i = 0; i < x.length; i++) {
x[i] = new A(0);
System.out.println(x[i].getValeur());
}
}
}
```

3.8. Solution de l'exercice 4

Le programme affiche :

```
0

0
```

Remarque *:* L'instruction : valeur = (byte)v; Signifie « Transtypage », elle converti v en byte.

3.9. Exercice 5

a) Créer une classe Etudiant qui possède :

- un attribut public de type String nommé nom ;
- un constructeur publique qui a un paramètre de type String servant à initialiser le nom de l'étudiant ;
- une méthode publique sans paramètre et qui ne renvoie rien, nommée `travailler`, qui affiche : Ahmed se met au travail !

b) Créer une classe TestEtudiant contenant une méthode main. Dans cette méthode main :

- créer un objet étudiant ;
- invoquer la méthode `travailler` de l'étudiant créé.

3.10. Solution de l'exercice 5

```
public class Etudiant {
public String nom;
public Etudiant(String nom) {
this.nom = nom;
}
public void travailler() {
System.out.println(nom + " se met au travail !");  }
}

public class TestEtudiant {
public static void main(String arg[]) {
Etudiant etudiant = new Etudiant("Ahmed");
```

```
etudiant.travailler();
}
}
```

Ce programme affiche :

```
Ahmed se met au travail !
```

3.11. Exercice 6

- Définir une classe Livre avec les attributs suivants : Titre, Auteur.
- Définir le constructeur d'initialisation de cette classe.
- Définir à l'aide des getters et des setters les méthodes d'accès aux attributs de cette classe.
- Définir la méthode `afficher` permettant d'afficher les informations du livre en cours.
- Écrire un programme testant la classe Livre.

3.12. Solution de l'exercice 6

```
public class Livre {
private String titre;
private String auteur;
public Livre(String titre, String auteur)
{
this.titre=titre ;
this.auteur=auteur ;
}
public String getTitre() {
return titre;
}
public void setTitre(String titre) {
```

```
this.titre = titre;
}
public String getAuteur() {
return auteur;
}
public void setAuteur(String auteur) {
this.auteur = auteur;
}
public void afficher()
{
System.out.println("Titre: " + titre + ", Auteur: " + auteur );
}
}
public class Test {
public static void main (String arg[]){
Livre L1 = new Livre ("Delannoy","Programmer en Java");
L1.afficher();
}
}
```

Le programme affiche :

```
Titre: Programmer en Java, Auteur: Delannoy
```

3.13. Exercice 7

Partie 1 :

Un employé est caractérisé par son nom, son prénom, son âge et sa date d'entrée en service dans l'entreprise.

Dans un fichier *Salaires.java*, coder une classe *Employe* dotée des attributs nécessaires, d'une méthode *calculerSalaire* (ce calcul dépendra en effet du type de l'employé, le corps de cette méthode sera complété

dans les parties suivantes de l'exercice), d'une méthode *getTitre* retournant la chaîne de caractères "*L'employé*" et une méthode *getNom* retournant une chaîne de caractères obtenue en concaténant la chaîne de caractères "*L'employé*" avec le nom et le prénom. Doter également votre classe d'un constructeur prenant en paramètre l'ensemble des attributs nécessaires.

Partie 2 :

Le calcul du salaire mensuel dépend du type de l'employé. On distingue les types d'employés suivants :

• Ceux affectés à la *Vente*. Leur salaire mensuel est le 20 % du *chiffre d'affaire* qu'ils réalisent mensuellement, plus 400 euros.

• Ceux affectés à la *Représentation*. Leur salaire mensuel est également le 20 % du *chiffre d'affaire* qu'ils réalisent mensuellement, plus 800 euros.

• Ceux affectés à la *Production*. Leur salaire vaut le *nombre d'unités* produites mensuellement multipliées par 5.

• Ceux affectés à la *Manutention*. Leur salaire vaut leur *nombre d'heures* de travail mensuel multipliées par 65 euros.

Coder dans votre fichier *Salaires.java* une hiérarchie de classes pour les employés en respectant les conditions suivantes :

• La super-classe de la hiérarchie doit être la classe *Employe*.

• Les nouvelles classes doivent contenir les attributs qui leur sont spécifiques ainsi que le codage approprié des méthodes *calculerSalaire* et *getTitre*, en changeant le mot "*L'employé*" par la catégorie correspondante.

• Chaque sous classe est dotée d'un constructeur prenant en argument l'ensemble des attributs nécessaires.

Remarque : N'hésitez pas à introduire des classes intermédiaires pour éviter au maximum les redondances d'attributs et de méthodes dans les sous-classes

Partie 3 :

Certains employés des secteurs *production* et *manutention* sont appelés à fabriquer et manipuler des produits dangereux. Après plusieurs négociations syndicales, ces derniers parviennent à obtenir une prime de risque mensuelle.

Complétez votre programme Salaires.java en introduisant deux nouvelles sous-classes d'employés. Ces sous-classes désigneront les employés des secteurs *production* et *manutention* travaillant avec des produits dangereux. Ajouter également à votre programme une interface pour les *employés à risque* permettant de leur associer une *prime mensuelle* fixe de 200.

3.14. Solution de l'exercice 7

```
// La classe Employe
abstract class Employe {
private String nom;
private String prenom;
private int age;
private String date;
public Employe(String prenom, String nom, int age,
String date)
{
this.nom = nom;
this.prenom = prenom;
this.age = age;
this.date = date;
}
```

```
public abstract double calculerSalaire();
public String getTitre()
{
return "L'employé " ;
}
public String getNom()
{
return getTitre() + prenom + " " + nom;
}
}
//La classe Commercial (regroupe Vendeur et Représentant)
abstract class Commercial extends Employe {
private double chiffreAffaire;
public Commercial(String prenom, String nom, int age,
String date, double chiffreAffaire)
{
super(prenom, nom, age, date);
this.chiffreAffaire = chiffreAffaire;
}
public double getChiffreAffaire()
{
return chiffreAffaire;
}

}
//La classe Vendeur
class Vendeur extends Commercial {
private final double POURCENT_VENDEUR = 0.2;
private final int BONUS_VENDEUR = 400;
public Vendeur(String prenom, String nom, int age,
String date, double chiffreAffaire)
{
super(prenom, nom, age, date, chiffreAffaire);
```

```
}
public double calculerSalaire()
{
return (POURCENT_VENDEUR * getChiffreAffaire()) +
BONUS_VENDEUR;
}
public String getTitre()
{
return "Le vendeur ";
}
}
//La classe Représentant
class Representant extends Commercial {
private final double POURCENT_REPRESENTANT = 0.2;
private final int BONUS_REPRESENTANT = 800;
public Representant(String prenom, String nom, int age,
String date, double chiffreAffaire)
{
super(prenom, nom, age, date, chiffreAffaire);
}
public double calculerSalaire() {
return (POURCENT_REPRESENTANT * getChiffreAffaire()) +
BONUS_REPRESENTANT;
}
public String getTitre()
{
return "Le représentant ";
}
}
//La classe Technicien (Production)
class Technicien extends Employe {
private final double FACTEUR_UNITE = 5.0;
private int unites;
```

```
public Technicien(String prenom, String nom, int age,
String date, int unites)
{
super(prenom, nom, age, date);
this.unites = unites;
}
public double calculerSalaire() {
return FACTEUR_UNITE * unites;
}
public String getTitre()
{
return "Le technicien ";
}
}
//La classe Manutentionnaire
class Manutentionnaire extends Employe {
private final double SALAIRE_HORAIRE = 65.0;
private int heures;
public Manutentionnaire(String prenom, String nom, int
age, String date, int heures)
{
super(prenom, nom, age, date);
this.heures = heures;
}
public double calculerSalaire() {
return SALAIRE_HORAIRE * heures;
}
public String getTitre()
{
return "Le manut. " ;
}
}
//L'interface d'employés à risque
```

```
interface ARisque {
int PRIME = 200;
}
//Une première sous-classe d'employé à risque
class TechnARisque extends Technicien implements
ARisque {
public TechnARisque(String prenom, String nom, int age,
String date, int unites)
{
super(prenom, nom, age, date, unites);
}
public double calculerSalaire() {
return super.calculerSalaire() + PRIME;
}
}
//Une autre sous-classe d'employé à risque
class ManutARisque extends Manutentionnaire implements
ARisque {
public ManutARisque(String prenom, String nom, int age,
String date, int heures)
{
super(prenom, nom, age, date, heures);
}
public double calculerSalaire() {
return super.calculerSalaire() + PRIME;
}
}
```

3.15. Exercice 8

1) Quel est le résultat de l'exécution de ce programme.

```
public class Figure {
private int abscisse;
```

```
    private int ordonnee;
    private String couleur;
    public Figure(int abscisse, int ordonnee, String couleur) {
    this.abscisse = abscisse;
    this.ordonnee = ordonnee;
    this.couleur = couleur;
    }
    public String toString() {
    return ("abscisse: " + abscisse + " ordonnee: " + ordonnee
    + " couleur: " + couleur);
    }
    public static void main(String[] argv) {
    Figure f = new Figure(12, 13, "rouge");
    System.out.println(f);
    }
    }
```

3.16. Solution de l'exercice 8

Le programme affiche

```
abscisse: 12 ordonnee: 13 couleur: rouge
```

3.17. Exercice 9

Qu'affiche le programme suivant ?

```
class ClasseA {
public void afficher() {
System.out.println("Bonjour: Je suis ClasseA");}
public void calculer() {
System.out.println("Bonjour : je suis ClasseA: méthode
calculer");
}
}
class ClasseB extends ClasseA {
```

```
public void afficher() {
System.out.println("Bonjour : je suis ClasseB"); }
public void calculer() {
System.out.println("Bonjour : je suis ClasseB: méthode
calculer");
}
}
class ClasseC extends ClasseB {
public void afficher() {
System.out.println("Bonjour: Je suis ClasseC"); }
}
class ClasseD extends ClasseA {
}
class ClasseE extends ClasseD {
public void afficher() {
System.out.println("Bonjour: Je suis ClasseE");
}
}
class ClasseF extends ClasseD {
}
class ClasseG extends ClasseF {
public void afficher() {
System.out.println("Bonjour: Je suis ClasseG");
}
}
class ClasseH extends ClasseG {
public void afficher() {
System.out.println("Bonjour: Je suis ClasseH");
}
}

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

```
ClasseA P1 = new ClasseA();
ClasseA P2 = new ClasseC();
ClasseA P3 = new ClasseG();
ClasseD P4 = new ClasseE();
ClasseF P5= new ClasseH();
ClasseF P6 = new ClasseG();
ClasseA P7 = new ClasseD();
ClasseD P8 = new ClasseF();
ClasseB P9 = new ClasseB();
ClasseA P10 = new ClasseB();
P1.afficher();
P2.afficher();
P3.afficher();
P4.afficher();
P5.afficher();
P6.afficher();
P7.afficher();
P8.afficher();
P9.calculer();
P10.calculer();
}
}
```

3.18. Solution de l'exercice 9

Le programme affiche :

```
Bonjour: Je suis ClasseA
Bonjour: Je suis ClasseC
Bonjour: Je suis ClasseG
Bonjour: Je suis ClasseE
Bonjour: Je suis ClasseH
Bonjour: Je suis ClasseG
Bonjour: Je suis ClasseA
```

```
Bonjour: Je suis ClasseA
Bonjour : je suis ClasseB: méthode calculer
Bonjour : je suis ClasseB: méthode calculer
```

3.19. Exercice 10

Une chaîne de caractères est dite Palindrome si elle peut être lue de gauche à droite ou de droite à gauche telles que : été, laval, AZZA et AZIZA. Écrire un programme Java qui permet de saisir une chaine de caractère et de vérifier si elle est un Palindrome ou non.

3.20. Solution de l'exercice 10

```
public class Test
{
public static void main(String[] args)
{
int i_deb,i_fin;
boolean palindrome=true;
Scanner in= new Scanner(System.in);
System.out.println ("donner un mot :  ");
/* la méthode nextLine() permet de lire une chaîne de caractères y compris le caractère blanc entre les mots jusqu'à la fin de la ligne (voir la page 56 de ce document) */
String mot=in.nextLine();
i_deb=0;
i_fin=mot.length()-1;
while(palindrome && (i_deb<i_fin))
{
if(mot.charAt(i_deb)== mot.charAt(i_fin))
{
i_deb++;
i_fin--;
```

```
}
/* la méthode charAt() permet  de  récupérer un caractère
à partir de son indice (voir la page 54 de ce document)*/

else
palindrome=false;
}
if(palindrome)
System.out.println("C'est un palindrome ");
else
System.out.println("Ce n'est pas un palindrome ");
}
}
```

3.21. Exercice 11

Qu'affiche le programme suivant ?

```
public class Test {
public static void main(String[] args) {
int tab[][]={{2,10,4},{12,15,20},{10,200,100}};
//affichage des éléments de la matrice
for (int i = 0; i < tab.length; i++) {
for (int j = 0; j < tab.length; j++) {
System.out.print(tab[i][j]+"\t");
}
System.out.println();
}
}
}
```

3.22. Solution de l'exercice 11

Le programme affiche :

```
2  10   4
12 15   20
10 200  100
```

3.23. Exercice 12

Qu'affiche le programme suivant ?

```
public class Test {
public static void main(String[] args) {
Object[][]mat
= {{"Java",16.5f,11},{'B',"c++",18.8},{18,100,'A'}};
for (int i = 0; i < mat.length; i++) {
for (int j = 0; j < mat.length; j++) {
System.out.print(mat[i][j]+"\t");
}
 System.out.println();
}
}
}
```

3.24. Solution de l'exercice 12

Le programme affiche :

```
Java   16.5  11
B      c++   18.8
18     100   A
```

Université A/MIRA Bejaia
Département d'Informatique
Module : GL
Documents personnels non autorisés

Épreuve de moyenne durée

Exercice 1:

1. Pourquoi dit-on que Java est un langage portable ?
2. Que signifie le terme « ramasse-miettes» en Java ?

Exercice 2 :

Voici le texte d'une classe représentant de façon sommaire un compte bancaire et les opérations bancaires courantes.

```
class Compte {
int solde = 0 ;
void depot (int montant ) {
solde = solde + montant ;
}
void retrait (int montant ) {
solde = solde - montant ;
}
void virement (int montant , Compte autre ) {
autre.retrait (montant) ;
this.depot ( montant ) ;
}
void afficher ( ) {
System.out.println ( " le solde = " +  solde ) ;
}
} //Fin de la classe Compte.
```

1. Comment fonctionne la méthode *virement ()* ? Combien de comptes fait-elle intervenir ?

2. Créez deux comptes que vous affecterez à deux variables, Biba et Samy. Ecrivez dans une classe principale le code Java correspondant aux opérations suivantes :
   a) Dépôt de 5000 euros sur le premier compte (Biba).
   b) Dépôt de 1000 euros sur le second compte (Samy).
   c) Retrait de 100 euros sur le second compte.
   d) Virement de 750 euros du premier compte vers le second.
   e) Affichage des soldes des deux comptes.

3. Qu'affiche l'exécution ?

Exercice 3 :

Dans un supermarché, un achat est décrit par une référence (String), un prix unitaire (double) et une quantité (double) de l'article acheté. Donner l'implémentation d'une classe Achat comportant une méthode `lire` pour la saisie au clavier. Une méthode `sommeAchat` qui retourne le prix de l'article acheté. Une méthode `sommeBonus` qui réduit le prix de l'article acheté de la moitié du prix unitaire. Dotée cette classe d'une méthode `main` permettant de calculer et d'afficher le prix de l'article acheté avec bonus et sans bonus.

Corrigé de l'EMD

Exercice 1

1. En Java la compilation d'un code source Java produit, non pas des instructions machine, mais un code intermédiaire formé de byte-codes. Ce code est exactement le même quel que soit le compilateur et l'environnement concernés. Ces byte-codes sont exécutables dans toute implémentation disposant de la JVM. Ainsi, grâce à la JVM, on peut dire que Java est un langage portable.

2. Ramasse-miettes est un mécanisme de gestion automatique de la mémoire employé par Java. Lorsqu'il n'existe plus aucune référence sur un objet. Il permet de libérer l'emplacement correspondant.

Exercice 2

1. La méthode `virement` fait intervenir deux objets de type Compte : *this*, l'objet sur lequel la méthode est appelée et *autre*, le paramètre de la méthode.

Le virement s'effectue du paramètre vers this, l'argent est retiré d'un compte et déposé sur l'autre.

2.

```
class Test {
public static void main ( String [ ] args ) {
Compte Biba = new Compte ( ) ;
Compte Samy = new Compte ( ) ;
Biba.depot (5000) ;
Samy.depot (1000) ;
Samy.retrait (100) ;
```

```
Samy.virement (750 , Biba) ;
System.out.print ( "Compte de Biba : " ) ;
Biba.afficher ( ) ;
System.out.print ( "Compte de Samy : " ) ;
Samy.afficher ( ) ;
}
}
```

3. Le résultat d'exécution est :

```
Compte de Biba :  le solde = 4250
Compte de Samy :  le solde = 1650
```

Exercice 3

```
import java.util.Scanner ;
class Achat {
String reference ;
double prixunit;
double quantite;
void lire(){
Scanner e= new Scanner(System.in);
System.out.println("Veuillez sasir la reference");
reference = e.next();
System.out.println("Veuillez sasir le prix unitaire");
prixunit=e.nextDouble();
System.out.println("la quantité achetée est");
quantite=e.nextDouble();
}
double sommeAchat(){return prixunit*quantite ;}
double sommeBonus(){return sommeAchat()-prixunit/2;}
```

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

Achat art= new Achat();

art.lire();

System.out.println("le prix de l'article "+ art.reference +
"  sans bonus est " + art.sommeAchat()+  " avec bonus est "+
art.sommeBonus());

}

}
```

Conclusion générale

À travers ce polycopié de cours, j'ai présenté les connaissances de base qu'un étudiant informaticien doit savoir sur le paradigme orienté objet. Ce document permet à l'étudiant de comprendre tout d'abord les raisons de l'apparition de la pensée objet dans le monde de l'informatique. Précisant que, l'approche procédurale existe toujours mais la tendance est de plus en plus vers l'approche objet vu ses avantages présentés dans ce document

En tant qu'un langage orienté objet, Java possède les meilleurs concepts des langages orienté objet qui le précèdent tels que Eiffel, Small Talk et C++. Dans Java à l'exception des types de données primitifs tout est objet. La syntaxe du langage Java est présentée dans le premier chapitre de ce document, ainsi que, la façon d'exécuter des codes.

La programmation objet en Java est traitée avec suffisamment de détails dans le chapitre 2 ce qui permet à l'étudiant de comprendre les principes de la POO. Commençant par l'instanciation et l'encapsulation puis le référencement et la manipulation des d'objets, suivis par la programmation des tableaux objets, des chaines de caractères et enfin les entrées/sorties en Java.

Le chapitre 3 présente le mécanisme d'héritage en détail afin d'apprendre à l'étudiants comment la réutilisabilité du code est assurée automatiquement en Java, contrairement au langage procédurale.

Le chapitre 4 présente les solutions des exercices du cours et le chapitre 5 propose des exercices de révision solutionnés.

A l'issu de ces cinq chapitres l'étudiant ayant pratiqué tous les codes des exercices sous Eclipse ou NetBeans trouvera le plaisir de programmer en orienté objet et comprendra mieux la différence entre la pensée objet et la

pensée procédurale. De plus, il aura la maitrise des paradigmes objets présentés dans les trois premiers chapitres de ce polycopié (modularité, réutilisabilité, instanciation, encapsulation, héritage, polymorphisme, classes abstraite, Interface, etc.).

Ce document peut être utilisé par les étudiants informaticiens novices, les étudiants de fin de cycle Licence et les étudiants de master (A & 2) préparant des applications sous Java. Précisant que, les codes écris dans ce polycopié sont écrits en langage Java. Ainsi l'étudiant pourra lire et faire des travaux pratiques à la fois.

Références

[1] P. Chatelin & P. E. Kuhn-Mounier. (1990). *Deuxième colloque sur l'histoire de l'informatique en France*, Paris, 24-26 avril 1990. http://jacques-andre.fr/chi/chi90/andre67.html

[2] L. Hamza. (2019). *Génie Logiciel*, Support de cours, Université de Bejaia. https://elearning.univbejaia.dz/course/view.php?id=5958

[3] M. Issoufou Tiado. (2020). *Analyse et Programmation Avancée en C : modularité, récursivité, arbres et fichiers*. avec plus de deux cents exercices : manuel des écoles, instituts et universités 1e et 2e années des filières d'informatique. Collection : Études africaines.

[4] C. Delannoy. (2008). *Programmer en Java*. Editions Eyrolles.

[5] S. Tahé. (2002). Apprentissage du langage JAVA. Université d'Angers, Septembre 98 - Révision juin 2002.

[6] B. Kahloula, *Programmer en Java* (préparation à la certification Java), Pages blues, 2023.

[7] C. Herby. (2018). *Apprenez à programmer en Java*. Eyrolles.

[8] M. Divay. (2006). *La programmation objet en Java, Cours et exercices corrigés.* DUNOD.

[9] https://www.java.com/fr/download/help/linux_x64_install.html

[10] https://help.ubuntu.com/community/Java