République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Mascara
Faculté des Sciences Exactes
Département d'Informatique

Polycopié de Cours en ligne

Programmation Orientée-Objet : Cours et exercices

Cours destiné aux étudiants de la 2ème année Licence Informatique

SALEM Mohamed

20222/2021

Table des matières

Table des figures

Liste des tableaux

Introduction

Résoudre un problème en Informatique, c'est de transformer une suite d'instructions constituant la solution manuelle de ce problème en un programme écrit dans un langage donné afin d'automatiser cette solution. La première étape de ce processus et la plus importante est l'analyse du problème qui consiste à desceller les différents données et traitements faisant partie de la solution du problème.

Plusieurs approches et paradigmes ont vu le jours afin de formaliser cette analyse et de palier les lacunes de l'approche naïve utilisée dés les premiers programmes. Ces approches déferrent selon la nature du problème et sa complexité.(Poo et al., 2007)

La programmation orientée-objet($P.O.O$) est le paradigme de programmation le plus robuste et qui ressemble le plus à la réalité où les problèmes sont composés d'objets et c'est les objets qui ont des données et des traitements(Gilbert, 1996; Budd, 1999). En effet, la $P.O.O$ n'est rien d'autre qu'une nouvelle façon de voir les choses, au lieu de considérer le problème traité comme un ensemble de données manipulées par des fonctions, elle considère les entités constituant le problème comme des objets indépendants encapsulant des attributs et des méthodes.(Bugayenko, 2016)

Le pèsent cours est dédié à l'introduction des principes de base de la $P.O.O$ et de la gestion des exceptions puisque celle-ci sont traitées comme des objets dans la mémoire. Il est destiné principalement aux étudiants en deuxième année Licence informatique à l'Université de Mascara, et en général, à tous les programmeurs et étudiants à la recherche d'approfondir leurs connaissances en paradigmes de programmation Orientée-Objet. le seul pré-requis de ce cours est la maitrise des principes de l'programmation.

Ce cours permettra aux lecteurs de :

a) Comprendre les principes de la $P.O.O$

b) Pouvoir différencier entre la notion de classe et objets.

c) Être capable de détecter les objets d'un problème donnée.

d) Maitriser et utiliser le principe d'héritage pour une une programmation souple et modulaire.

e) Comprendre les exceptions et leurs gestion.

Ce cours est scindé en cinq chapitres, dans le premier, nous avons introduit les différents paradigmes de programmation, leurs avantages et inconvénients tandis que le deuxième chapitre est dédié à présenter les notions de base des objets et classes. Le troisième chapitre met en exergue

le principe d'héritage et celui du polymorphisme et leurs règles d'utilisation. Ces deux notions sont fortement liées aux principes de classes abstraites et interfaces décrits dans le quatrième chapitre. Le dernier chapitre présente la gestion des exceptions dans un programme.

Dans ce cours, nous allons détailler les principes de la *P.O.O* en donnant les exemples en langage *JAVA* comme l'exige le programme du module mais ces principes sont applicables à tous les langages de programmation modernes comme *Python, C#, Ruby, Kotlin, Scala, ...* .

CHAPITRE I

Paradigmes de programmation

I.1 Définition d'un paradigme de programmation

Définition :

Un paradigme de programmation est un style fondamental de programmation qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation. On peut programmer la même chose avec tous les langages. Ils ont tous le même pouvoir d'expressivité (machines de Turing)(Watt, 1990).

Remarque : Les structures de données et les fonctions qui les utilisent sont séparées.Les données sont généralement en début de programme et donc visibles de toutes les fonctions qui suivent(Nell et al., 2013).

Programme = Structure de données + Instructions

I.2 Approches classiques de programmation

I.2.1 Approche naïve

Les premiers programmes informatiques étaient généralement constitués d'une suite d'instructions s'exécutant de façon linéaire (l'exécution commence de la première instruction du fichier source et se poursuivait ligne après ligne jusqu'à la dernière instruction du programme). Cette approche, bien que simple à mettre en ?uvre, a très rapidement montré ses limites. En effet, les programmes monolithiques de ce type (Watt, 1990) :

— Ne se prêtent guère à l'écriture de grosses applications.

— Ne favorisent absolument pas la réutilisation du code.

En conséquence, est apparue une autre approche radicalement différente : l'approche procédurale.

I.2.2 Approche procédurale

Consiste à découper un programme en un ensemble de fonctions (ou procédures) (Figure.I.1). Ces fonctions contiennent un certain nombre d'instructions qui ont pour but de réaliser un traitement particulier. Elle est aussi appelée approche impérative Nous pouvons donner comme exemples de traitements qui peuvent être symbolisés par des fonctions (Watt, 1990) :

— Le calcul de la circonférence d'un cercle.

— L'impression de relevé de notes d'un étudiant.

Exemple de langages de programmation procédurale : *C*, *Pascal*, *Fortran*, etc. Dans le cas de l'approche procédurale, un programme correspond à l'assemblage de plusieurs fonctions qui s'appellent entre elles.(Voir Figure I.2)

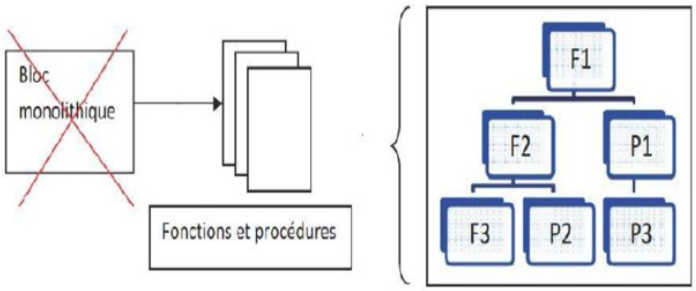

FIGURE I.1 – Approche procédurale.

L'approche procédurale favorise (Watt, 1990) :

— La création d'un code plus modulaire et structuré.

— La possibilité de réutiliser le même code à différents emplacements dans le programme sans avoir à le retaper.

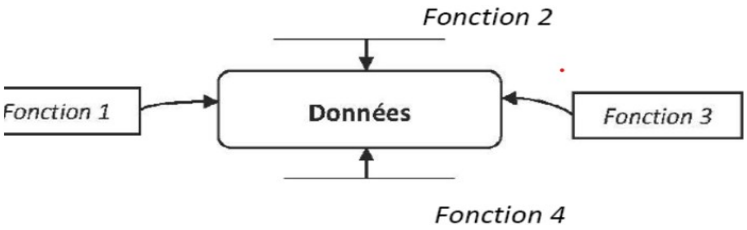

FIGURE I.2 – Approche procédurale est une approche centrée-donnée.

Attention : Malgré ses avantages, l'approche procédurale présente également des inconvénients (Watt, 1990; Permanand, 2013) tel que :

— Les fonctions, procédures accèdent à une zone où sont stockées les données. Il y a donc une dissociation entre les données et les fonctions ce qui pose des difficultés lorsque l'on désire changer les structures de données.

— Dans les langages procéduraux, les procédures s'appellent entre elles et peuvent donc agir sur les mêmes données. Il y a donc un risque de partage de données (écriture en même temps dans le même fichier).

I.2.3 Autres approches classiques

En plus de ces paradigmes, il existe dautres approches selon la nature du problème traité, citons(Bole, 2021; Watt, 1990) :

— La programmation fonctionnelle : Elle adopte une approche beaucoup plus mathématique de la programmation. : *Lisp*.

— La programmation logique : la description d'un programme est sous forme de prédicats. : *Prolog*.

I.3 Approche Orienté-Objet

De ces problèmes est issus une autre manière de programmer c'est la programmation par objet ou bien L'approche orientée objet (Début des années 80). Selon cette approche, un programme est vu comme un ensemble d'entités (ou objets). Au cours de son exécution(Voir Figure.I.3), ces entités collaborent en s'envoyant des messages dans un but commun.(Bugayenko, 2016)
C'est une programmation dans laquelle les programmes sont organisés comme des ensembles d'objets coopérant ensemble.

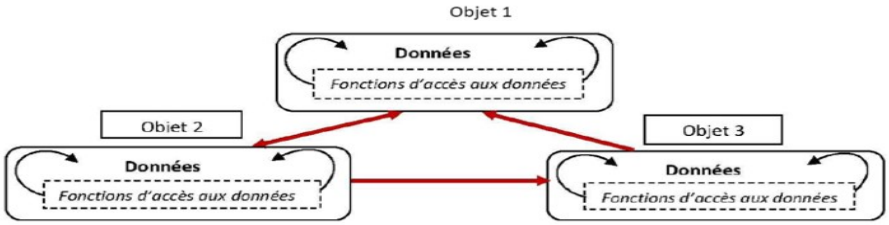

FIGURE I.3 – Approche Orienté-Objet

Nous avons dans ce schéma un lien fort entre les données et les fonctions qui y accèdent. Mais qu'appelle-t-on un objet ? Que représente un objet ?

I.4 Les principes de la P.O.O

La programmation orienté-objet est basée sur un ensemble de principes (Huw, 2020) :

- Encapsulation : Toutes les données sont soit liées à un objet, soit liées à une classe. Chaque objet peut manipuler ces données mais, ne peut accéder aux données d'un autre objet qu'à travers l'échange de message avec l'objet en question
- Séparation et spécialisation : Les classes définies (résultat de découplage) doivent être clairement séparables et ne partagent pas des propriétés similaires
- Abstraction et généralisation : Détail de l'implémentation seront cachés et les propriétés partagées doivent être modélisées dans une même entité (classe).
- Héritage : possibilité d'héritage entre les classes qui partagent des propriétés communes de telle façon qu'une classe est une extension d'une autre classe.

I.5 Exercice

Écrire un programme en C++ qui reçoit les informations de N étudiants en entrée (noms, année et notes). Chaque étudiant a trois notes dans trois modules.

Nous voulons calcules la moyenne selon les cas :

$$Moyenne = \begin{cases} \frac{note1*coef1+note2*coef2+note3*coef3}{coef1+coef2+coef3} & \text{if année} < 3 \\ \\ PFE + \frac{\frac{note1*coef1+note2*coef2+note3*coef3}{coef1+coef2+coef3}}{2} & \text{if année} = 3 \end{cases} \tag{I.1}$$

CHAPITRE II

Classes et Objets

II.1 Objet

II.1.1 Notion d'objet

Définition :

Un objet représente une entité individuelle et identifiable, réelle ou abstraite, avec un rôle bien défini dans le domaine du problème.(Gervais, 2014) Un objet est constitué d'un ensemble de données (ses attributs) et d'un ensemble d'opérations (méthodes) (Figure.II.1)

- Les données (ou champs) qui décrivent sa structure interne sont appelées ses attributs ;
- Les méthodes qu'à travers elles que les objets interagissent entre eux.

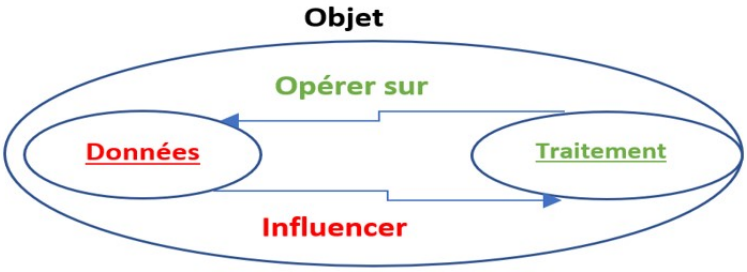

FIGURE II.1 – Représentation d'un objet

Chaque objet peut être caractérisé par une identité, des états significatifs et par un comportement(Figure.II.2).

Objet = État + Comportement + Identité

II.1.2 État

Définition :

L'état d'un objet comprend les propriétés statiques (attributs) et t les valeurs de ces attributs qui peuvent être statiques ou dynamiques.(Gervais, 2014)

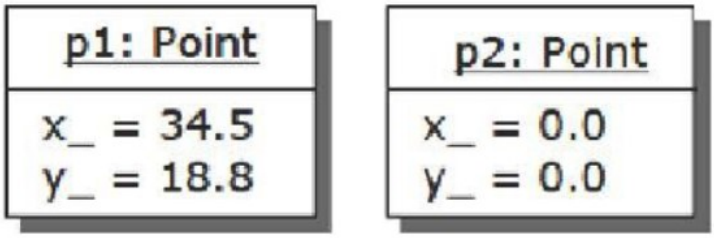

FIGURE II.2 – Exemple d'un objet

II.1.3 Comportement

Définition :

Le comportement d'un objet se défini par l'ensemble des opérations qu'il peut exécuter en réaction aux messages envoyés.(Nell et al., 2013; Gervais, 2014) (un message = demande d'exécution d'une opération) par les autres objets(FigureII.3).

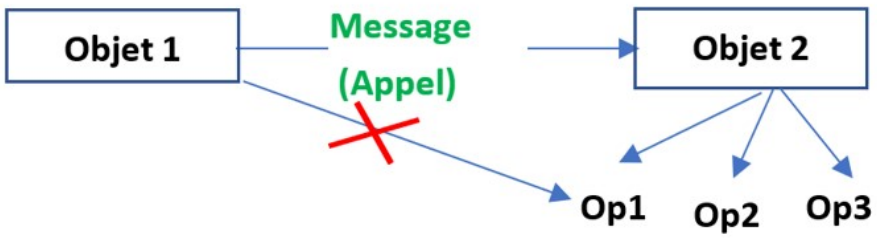

FIGURE II.3 – Communication entre objets

II.1.4 Identité

Définition :

Propriété d'un objet qui permet de le distinguer des autres objets de la même classe.(Gervais, 2014)

II.2 Classes

II.2.1 Définitions

Définition 1 :

La classe est une structure informatique particulière dans le langage objet. Elle décrit la structure interne des données et elle définit les méthodes qui s'appliqueront aux objets de même famille (même classe) ou type.(Poo et al., 2007; Gervais, 2014)

Une classe regroupe un ensemble de données (qui peuvent être des variables primitives ou des objets) et un ensemble de méthodes de traitement de ces données et/ou de données extérieures à la classe. La classe doit comporter obligatoirement une méthode spécifique (constructeur) qui permet de créer des objets (instance de la classe).

Définition 2 :

Une classe est un modèle à partir duquel on peut générer un ensemble d'objets partageant des attributs et des méthodes communes. Les objets appartenant à celle-ci sont les instances de cette classe c.à.d. que L'instanciation est la création d'un objet d'une classe.(Gervais, 2014; Permanand, 2013)

II.2.2 Déclaration d'une classe

Une classe est une structure sauf que :

1. Le mot réservé **class** remplace **struct**.
2. Certains champs sont des fonctions.

En Java, une classe se définit grâce au mot clef **class** suivi du nom de celle-ci. Ce nom commence généralement par une majuscule. A la suite, les données membres et les méthodes sont déclarées entre deux accoladesHuw (2020).

Syntaxe :

```
public class NomClasse {
//données membres
// fonctions membres
}
```

II.3 Attributs

Définition :

Un attribut est une caractéristique, qualité ou trait intrinsèque qui contribue à faire unique un objet.(Gervais, 2014)

Les attributs sont les variables qui constituent les données de la classe, elles sont associées à la classe.
Les données d'une classe sont contenues dans des attributs. Les attributs sont les propriétés d'une classe, ils sont tous visibles à l'intérieur de la classe. Les attributs sont donc des variables relatives à une instance (d'une classe) et qui sont accessibles dans toute la classe.(Bole, 2021)
La syntaxe générale de déclaration des attributs est :

[modificateur de visibilité] type nom_attribut [=expression] ;

Remarque :

- **Les modificateurs de visibilité pour les attributs sont : *public*, *private* et *protected*.**
- **On peut aussi utiliser le modificateur *Final* si l'attribut est une constante.**
- **Les types des attributs peuvent être des type primitif (int, double, ...) ou bien des type références (objet, ...)**

Exemple :

```
class Etudiant{
private String nom ;
private String adresse;
private String prenom ;
final double annee=2020;
}
```

II.4 Méthodes

Définition 1 :

Une méthode est une fonction qui appartient à une classe.(Gervais, 2014)

Elles sont déclarées par (Gilbert, 1996; Budd, 1999) :

```
   modificateurs type_retourné nom_méthode ( arg1, ... )
   {
// Définition des variables locales
// Définition du bloc d'instructions
}
```

- **Le type retourné peut être élémentaire (primitif) ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise *void*.**
- **Les arguments sont passés par valeur.**

Remarque :

- **Toutes les méthodes sauf les *constructeurs* doivent avoir un type de retour.**
- **Les méthodes qui ne renvoient rien, utilisent *void* comme type de retour.**
- **Une méthode peut ne pas avoir besoin d'arguments, ni de variables à déclarer.**
- **Les méthodes agissent sur les attributs définis à l'intérieur de la classe.**
- **Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.(Bole, 2021)**
- **La valeur de retour de la méthode doit être transmise par l'instruction return. Elle indique la valeur que prend la méthode et termine celle-ci : toutes les instructions qui suivent return sont donc ignorées.**

Exemple :

```
public class Etudiant{
private String nom ;
private String adresse;
private String prenom ;
//définition de la méthode afficherEtudiant qui permet d'afficher
    les données d'un étudiant
public void AfficherEtudiant() {
System.out.println("le Nom de l'etudiant est:"+Nom);
System.out.println("le prenom de l'etudiant est:"+Prenom);
System.out.println("l'adresse de l'étudiant est:"+Adresse);
   }
}
```

II.5 Constructeurs

Définition 1 :

La déclaration d'un objet est suivie d'une sorte d'initialisation par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une valeur de départ. Elle n'est systématiquement invoquée que lors de la création d'un objet.(Gervais, 2014)

Le constructeur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé, pas même void, donc il ne peut pas y avoir d'instruction return dans un constructeur. On peut définir plusieurs constructeurs pour une classe selon le type.(Budd, 1999; Poo et al., 2007)

II.5.1 Constructeur simple (Par défaut)

Ce type de constructeur ne nécessite pas de définition explicite : son existence découle automatiquement de la définition de la classe.(Huw, 2020)

```
public class Etudiant{
.....
public Etudiant() { }
}
```

II.5.2 Constructeur avec initialisation

Il existe deux types de constructeurs à initialisation :

— **Initialisation fixe**

Dans ce type de constructeur, les attributs de la classe sont initialisés avec des valeurs constantes données par le programmeur

```
public class Etudiant{
private String Nom;
public Etudiant() {
Nom= 'aymen' ;
   }
}
```

— **Initialisation par paramètres**

Pour spécifier les valeurs de données à initialiser on peut les passer en paramètres au constructeur.(Permanand, 2013; Cooper, 1997)

```
public class Etudiant{
private String Nom;
public Etudiant(String N1 ){
Nom= N1 ; } }
```

II.5.3 Constructeur de recopie

Le constructeur de recopie est utilisé lorsqu'on veut initialiser un objet avec les données d'un autre objet de la même classe.(Bole, 2021)

```
public class Etudiant{
private String Nom;
public Etudiant(Etudiant e ){
Nom= e.Nom; } }
```

II.6 Encapsulation

Définition :

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées private à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés *échanges de message*.(Gervais, 2014)

II.7 Accésseurs

Définition :

Un accésseur est une méthode publique qui donne l'accès à une variable d'instance privée.(Permanand, 2013)

Pour une variable d'instance, il peut ne pas y avoir d'accésseur, un seul accésseur en lecture ou un accésseur en lecture et un autre en écriture. Par convention, les accésseurs en lecture commencent par *get* et les accésseurs en écriture commencent par *set*.

Exemple :

```
public class Etudiant{
private String Nom;
....
public Etudiant( ) { }
public String getNom(){ return Nom ;}
public void setNom(String N1){ Nom=N1;
}}
```

II.8 Instanciation des objets en Java

Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme (Huw, 2020) :

nom_de_classe nom_de_variable ;

Exemple :

```
public static void main(String args[]) {
//déclaration des objets
Etudiant e1;
Etudiant e2;
//Instanciation des objets
e1=new Etudiant() ;
e2=new Etudiant() ;
//déclaration et instanciations des objets
Etudiant E3= new Etudiant() ;}
```

— **L'opérateur new se charge de créer une instance de la classe et de l'associer à la variable. Il est possible de tout réunir en une seule déclaration (comme pour l'objet e3).**

— **Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet. En Java, tous les objets sont instanciés par**

allocation dynamique.

Dans l'exemple, la variable e1 contient une référence sur l'objet instancié (contient l'adresse de l'objet qu'elle désigne :

- Si e2 désigne un objet de type Etudiant, l'instruction *e2 = e1* ne définit pas un nouvel objet mais e1 et e2 désignent tous les deux le même objet.
- L'opérateur new permet d'instancier des objets et d'appeler une méthode particulière de cet objet : le constructeur. Il fait appel à la machine virtuelle pour obtenir l'espace mémoire nécessaire à la représentation de l'objet puis appelle le constructeur pour initialiser l'objet dans l'emplacement obtenu. Il renvoie une valeur qui référence l'objet instancié. Si l'opérateur new n'obtient pas l'allocation mémoire nécessaire, il lève l'exception OutOfMemoryError.
- Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.

Attention : La durée de vie d'un objet passe par trois étapes (Permanand, 2013) :

1. la déclaration de l'objet et l'instanciation par l'opérateur *new*.
2. l'utilisation de l'objet en appelant ces méthodes
3. la suppression de l'objet : elle est automatique en Java grâce à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (garbage collector). Il n'existe pas d'instruction *delete* comme en C++.

II.9 Appel des attributs et méthodes des objets

II.9.1 Pour les attributs

L'échange des données entre les objets (les messages) est réalisé à travers l'exécution des méthodes de chaque objet ou bien à travers l'accès à ses attributs. Pour accéder à la valeur d'un attribut d'un objet, il faut utiliser le modèle de construction de l'objet (la classe) en utilisant la syntaxe suivante (Cooper, 1997) :

```
obj. nom_attribut ;
```

Cette opération n'est autorisée que si l'attribut est déclaré *public*.

II.9.2 Pour les méthodes

Concernant l'appel des méthodes, il est réalisé de la même façon que pour les attributs. Après la création d'un objet, l'appel des méthodes est effectué selon la syntaxe suivante (Permanand, 2013) :

```
obj.nomMethodes(arg1, arg2,...) ;
```

Si la méthode retourne un résultat, il faut que la variable déclarée soit de même type. Il est à remarquer aussi que le nombre des arguments doit correspondre à la définition de la méthode dans la classe correspondante. Si la méthode ne contient aucun argument, les parenthèses sont maintenues à vide.(Budd, 1999)

Objet.NomMethodes() ;

Supposant que l'attribut Nom de la classe Etudiant est déclaré public.

Exemple 1 :

```
public class Principal{
public static void main(String []args) {
e1= new Etudiant() ;
//On donne une valeur à l'attribut Nom
e1.Nom='Ahmed' ;
//On récupère le prenom de l'étudiant e1 dans la variabe locale
   Prenom déclarée dans la //méthode main.
String Prenom= e1.Prenom;
} .....}
```

Les attributs Nom et Prenom de la classe Etudiant sont déclarés private)

Exemple 2 :

```
public class Principal{
..... public static void main(String []args) {
E1= new Etudiant() ;
//On donne une valeur à l'attribut Nom
E1.setNom('Ahmed') ;
//On récupère le prénom de l'étudiant E1 dans la variable locale
   Prenom déclarée dans la méthode main.
String Prenom= E1.getPrenom();
} ..... }
```

II.10 Exercices

II.10.1 Exercice 1

1. **Créer une classe pour représenter les nombres complexes.**
2. **La tester avec une application TestComplexe.**
3. **La classe Complexe contient 2 variables d'instance de type double : partieRelle et artieImaginaire.**
4. **Prévoir 3 constructeurs : sans paramètre (alors le complexe vaut (0,0)), avec 1 paramètre "la partie réelle" (alors le complexe vaut (partieReelle, 0)) et avec deux paramètres.**
5. **Écrire les méthodes accésseurs ainsi que la méthode Afficher.**

6. **Créer la méthode additionner et la méthode soustraire qui créent chacune un nouveau complexe.**

II.10.2 Exercice 2

Le système de gestion d'un lycée à pour but de calculer de manière automatique les moyennes des élèves ainsi que le taux de réussite et cela pour chaque classe du lycée. Un lycée est composé de plusieurs classes. Chaque classe contient plusieurs élèves.

— **Définir les différents objets du système ainsi que leurs attributs.**

II.10.3 Exercice 3

1. **Écrire une classe Date formée de trois attributs entiers : année, mois, jour.**
2. **Définir deux constructeurs, l'un initialise la date au 01/01/2000 et l'autre initialise la date par une valeur passée en entrée.**
3. **Implémenter les méthodes pour :**
   — **Afficher la date en format longue et en format mdy**
   — **Comparer deux dates.**
   — **Donner le mois en lettres.**
   — **Tester si l'année est bissextile.**
   — **Ajouter un nombre de jours a une date**

II.10.4 Exercice 4

Dans cet exercice vous créerez une classe Compte pour gérer le compte bancaire d'un client. On devra retrouver un certain nombre d'informations sur le compte, à savoir :

— **Le numéro du client qui est unique et attribué automatiquement à la création du compte.**
— **Le nom du client qui ne peut changer durant toute l'existence du compte.**
— **Le solde.**

1. **Ecrire la classe Compte en donnant les méthodes usuelles d'utilisateur pour la consultation, le retrait et le dépôt.**
2. **Créer une autre classe Banque pour gérer les comptes de différents clients :**
   (a) **Une Banque gère plusieurs comptes.**
   (b) **Une banque peut croître (ajout d'un compte à la liste de comptes à gérer)**
   (c) **Un client peut quitter la banque (suppression d'un compte à partir de son numéro)**
   (d) **Un agent de banque vérifie des informations des comptes (Obtenir un compte à partir de son numéro).**

CHAPITRE III

Héritage et Polymorphisme

III.1 Héritage

III.1.1 Définition de l'héritage

Soit la classe Etudiant du chapitre précédent :

Exemple :

```
class Etudiant{
private String Nom, prenom ;
private double notes[] ;
public Etudiant(String n, String p){
this.nom=n ;
this.prenom=p ;}
public void afficher (){
System.out.println(this.nom+" "+this.prenom)}
public double calc_moy(){.....}}
```

Un étudiant en troisième année à les mêmes attributs que les autres étudiants (Nom, prénom , année et 3 notes) mais il a en plus une note de projet de fin d'étude et donc la méthode de calcul de moyenne est différente un peu.

Est ce qu'on écrit une nouvelle classe à partir de zéro ?

En POO, on exploite la classe qu'on a déjà et on applique le principe d'héritage.

Définition :

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Il permet d'implémenter le principe d'abstraction par la factorisation des propriétés et des opérations partagées par différents objets.(Gervais, 2014; Nell et al., 2013)

Une classe générale définit alors un ensemble d'attributs qui sont partagés par d'autres classes, dont on dira qu'elles héritent de cette classe. Grâce à l'héritage on peut dériver de nouvelles classes à partir des classes existantes.(Permanand, 2013)

- **La classe de base est appelé classe mère ou super-classe.**
- **La classe dérivée est appelé sous-classe.**
- **Une sous-classe est une extension de sa super-classe, c.-àd. elle comprend toutes ses propriétés et ses méthodes (Voir Figure.III.1).**

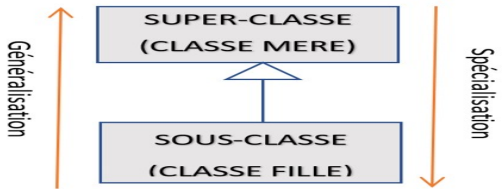

FIGURE III.1 – Super classe et sous-classe

III.1.2 La syntaxe de l'héritage en JAVA

La structure d'une classe héritée est donnée par(Bugayenko, 2016) :

```
public class Nomclass1 extends Nomclass2{
....
}
```

Nous avons créer une nouvelle classe Etudiant3 sur la base de la classe Etudiant en ajoutant un attribut pfe.

Exemple :

```
public class Etudiant3 extends Etudiant{
private double pfe;
}
```

Attention :

En Java, on ne peut hériter que d'une classe à la fois.

III.1.3 Pourquoi hériter ?

On fait recours à l'héritage quand (Bole, 2021; Poo et al., 2007) :

- **Quand il existe des objets différents dans l'ensemble mais qu'ils ont des propriétés ou des méthodes communes.**
- **L'héritage permet de définir les parties communes dans une super-classe et de définir le reste dans des sous-classes.**
- **L'héritage permet de réduire la répétition de données.**
- **L'héritage permet la redéfinition des méthodes.**

III.2 Classe Object

En Java, la racine de l'arbre d'héritage des classes est la classe java.lang.Object.(Gilbert, 1996; Huw, 2020)

- **La classe Object n'a pas de variable d'instance ni de variable de classe**
- **La classe Object fournit plusieurs méthodes qui sont héritées par toutes les classes sans exception**
- **Les plus couramment utilisées sont les méthodes toString et equals.**

III.2.1 Graphe d'héritage

le graphe d'héritage est une hiérarchie de classes, l'une hérite de l'autre jusqu'à la classe Object.(Bole, 2021)

Soit la déclaration :

Exemple :

```
class A {}
class B extends A {}
class C extends B {}
class D extends A {}
```

Le graphe d'héritage de cet exemple est donné par la figure III.2 :

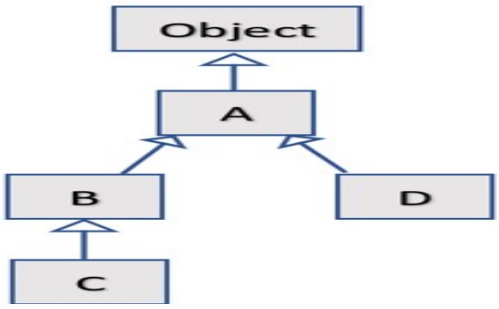

FIGURE III.2 – Super classe et sous-classe

III.2.2 Méthode toString

Elle est parmi les méthodes les plus connues de la classe Object. Elle est déclaré par (Cooper, 1997).

```
public String toString();
```

Pour être utilisée dans les nouvelles classes, la méthode toString() de la classe Object doit donc être redéfinie pour renvoyer une description de l'objet sous la forme d'une chaîne de caractères de l'objet

Exemple :

```
public class Livre {
...
@Override
public String toString() {
return "Livre [titre=" + titre + ",auteur=" + auteur + ",nbPages=" +
    nbPages + "]";
}
}
```

III.2.3 Méthode equals

La méthode equals de Object renvoie true si et seulement si l'objet courant this a *la même valeur* que l'objet obj, c'est-à-dire si this référence le même objet que obj.(Bugayenko, 2016)

```
public boolean equals(Object obj);
```

III.2.4 Méthode getName

La méthode getName() de la classe Class renvoie le nom complet de la classe (avec le nom du paquetage) La méthode getSimpleName() de la classe Class renvoie le nom terminal de la classe (sans le nom du paquetage).(Huw, 2020)

III.2.5 Instanceof

Si x est une instance d'une sous-classe B de A, x instanceof A renvoie true. Pour tester si un objet o est de la même classe que l'objet courant, il ne faut donc pas utiliser instanceof mais le code suivant (Permanand, 2013; Poo et al., 2007) :

```
if ((o != null) &&( o.getClass() == this.getClass())) ....
```

III.3 Accès et visibilité

Toute propriété ou méthode public d'une super-classe fait partie aussi de toutes les sous-classes.(Nell et al., 2013)

- **Les propriétés privées d'une super-classe ne sont pas visibles pour les sous-classes. Donc elle ne sont pas accessibles directement (accès indirect via une méthode public) Voir Table III.1.**
- **En Java on peut déclarer une propriété ou méthode privée mais visible dans les sous-classes. C'est grâce au mot clé protected.(Budd, 1999)**
- **Dans l'exemple précédent les attributs de la classe Etudiant sont toutes privés, Nous devons les déclarer protected pour qu'ils soient visibles dans classes Etudiant3.**
- **Concernant les méthode, une question qui se pose : est ce qu'on peut étendre une méthode déjà définie dans la super-classe ?**

TABLE III.1 – Accessibilité des méthodes en JAVA

	Méthodes		
	classe	classes filles	externes
public	Oui	Oui	Oui
private	Oui	Non	Non
protected	Oui	Oui	Non

III.4 Redéfinition

III.4.1 Redéfinition des méthodes

Définition :

La redéfinition consiste à redéfinir, dans une classe descendante, une méthode implémentée dans la classe mère. Généralement, le code à redéfinir est une extension de la méthode définie dans la classe mère. C'est pour cela, qu'on commence par l'appel de la méthode définit dans la classe mère en utilisant le mot clef super.(Gilbert, 1996; Gervais, 2014)

```
super.Nomdelamethode (param1, param2,...);
```

Pour redéfinir la méthode afficher dans Etudiant3 :

Exemple :

```
    @Override
public void afficher() {
super.afficher() ;
System.out.println(' la note du PFE de l'etudiant est''+this.pfe);
}
```

III.4.2 Redéfinition des constructeurs

Dans le constructeur de la classe fille , on peut appeler le constructeur de la classe mère, on utilise directement super (param1, param2,...).(Bole, 2021; Gilbert, 1996)

Exemple :

```
public Etudiant3(String n, String p, double pfe){
super(n,m) ;
this.pfe=pfe ;}
```

Important :

Si la première instruction d'un constructeur n'est ni super(...), ni this(...), le compilateur ajoute au début un appel implicite super()au constructeur sans paramètre de la classe mère (erreur de compilation s'il n'existe pas !)
Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur.(Bugayenko, 2016; Cooper, 1997)

Remarque :

- **La première instruction d'un constructeur de la classe mère est l'appel à un constructeur de la classe *grand-mère*, et ainsi de suite...**
- **Donc la toute, toute première instruction qui est exécutée par un constructeur est le constructeur (sans paramètre) de la classe Object. C'est le seul qui sait comment créer un nouvel objet en mémoire).**
- **Depuis Java 5 on peut annoter par @Override une méthode qui redéfinit une méthode d'une classe ancêtre.**

III.4.3 Redéfinition et surcharge

Ne pas confondre redéfinition et surcharge des méthodes (Bole, 2021). en effet :

- **Une méthode redéfinit une méthode héritée quand elle a la même signature que l'autre méthode**
- **Une méthode surcharge une méthode (héritée ou définie dans la même classe) quand elle a le même nom, mais pas la même signature, que l'autre méthode.**

III.5 Transtypage

Définition :

Le transtypage (casting) en anglais permet de convertir le type d'un variable, d'un objet ou d'une expression. Il existe deux type de transtypage :

1. **Implicite *(Down-casting)* : est un transtypage descendant qui permet de changer le type d'un objet d'une classe à une classe inférieur.**
2. **Explicite *(Up-casting)* : est un transtypage ascendant qui permet de changer le type d'un objet d'une classe à une classe supérieur.(Budd, 1999; Gervais, 2014)**

Exemple :

```
class Parent extends Personne{
protected Vector enfants;
}
class Homme extends Parent{
Femme epouse;
}
class Femme extends Parent{
Homme epoux;
}
class Enfant extends Personne{
Homme pere;
Femme mere;
}
public class Heritage {
public static void main(String[] args){
Parent p1 = new Homme(...); // Transtypage implicite
Parent p2 = new Femme(...); // Transtypage implicite
Vector v = new Vector();
for(int i=0 ; i<3 ; i++)
v.add(new Enfant(...)); // Object <- Enfant
Homme h = (Homme)p1; // Transtypage explicite
Femme f = (Femme)p2; // Transtypage explicite
/**************************/
h.epouse = f;
f.epoux = h;
h.enfants = f.enfants = v;
}}
```

III.6 Polymorphisme

Définition :

Le polymorphisme est un concept qui permet de manipuler des objets sans en connaître tout à fait le type tout en se basant sur la relation d'héritage. Un objet peut être déclaré en utilisant un *type* générique et ensuite il sera créé à travers une classe descendante.(Huw, 2020; Gervais, 2014)

Exemple :

```
class Enfant extends Personne{
Homme pere;
Femme mere;
...
public String toString(){
return super.toString()+ pere +" "+ mere;
}
}
public class Heritage {
public static void main(String[] args) throws
ParseException {
Enfant e = new Enfant(...);
e.pere = new Homme(...);
e.mere = new Femme(...);
Personne p = e;
   /**************************/
System.out.println(p.toString());
// la méthode toString() invoquée est celle de la
// classe Enfant même si p est déclaré Personne
}}
```

Par l'application d'un transtypage implicite sur un objet appartenant à une super-classe, il devient une instance de la sous-classe et il va changer son comportement.
Le changement du comportement d'un objet veut dire (Nell et al., 2013) :

- **qu'il peut accéder à des méthodes définies seulement dans la sous-classe.**
- **qu'il va utiliser les méthodes redéfinies dans la sousclasse.**

Ce principe est appelé le Polymorphisme. Donc on peut tous simplement définir le polymorphisme comme une adaptation dynamique du comportement dynamique des objets d'une super-classe.
Dynamique : signifie que la détermination de la méthode à invoque se fait au cours d'exécution (Dynamic Binding). (Permanand, 2013)

Remarque : La surcharge est un cas spéciale du polymorphisme qui s'appelle polymorphisme ad hoc.

III.7 Classe, méthode et propriété finales

Définition :

- **Une classe finale est une classe qui n'accepte pas d'extensions (elle ne peut pas avoir des sous-classe).(Bole, 2021)**
- **Une méthode finale est une méthode qui n'accepte pas de redéfinitions dans les sous-classe.(Gervais, 2014)**
- **Une propriété finale n'accepte qu'une seule valeur qui ne sera pas changée (joue le rôle d'un constant).(Bole, 2021)**

En java on utilise le mot-clé final, par exemple :

Exemple :

```
final class Enfant { // classe finale
...
private final int taille = 2; // propriété finale
...
public final Parent[] getParent(){ // méthode finale
Parent[] parents = new Parent[taille];
parents[0] = pere;
parents[1] = mere;
return parents;
}
```

III.8 Exercices

III.8.1 Exercice 1

Une société à un nombre d'employés. Un employé est connu par son nom, type(Vacataire, Titulaire) son matricule (qui l'identifie de façon unique) et son salaire. Le salaire est calculé en multipliant un indice par une certaine valeur qui peut changer en cas d'augmentation générale des salaires, mais qui est la même pour tous les employés. Le matricule a la forme (V0, V1 ,T2)

- **Écrivez la classe des employés avec les informations utiles et des méthodes pour afficher les caractéristiques d'un employé et pour calculer son salaire. Certains employés ont des responsabilités. Ils ont sous leurs ordres d'autres employés.**
- **Écrivez une sous-classe des employés qui représente ces responsables en enregistrant leurs inférieurs hiérarchiques directs dans un tableau.**
- **Écrivez une méthode qui affiche les inférieurs directs (placés directement sous leurs ordres) et une autre qui affiche les employés inférieurs directs ou indirects.**

Les commerciaux ont un salaire composé d'un fixe et d'un montant égal au cinquième de ces ventes.

— **Écrivez une sous-classe des commerciaux qui contient l'information sur leurs ventes du dernier mois, une méthode pour mettre à jour cette information et redéfinissez la méthode de calcul de leurs salaires.**

— **Écrivez une classe représentant tout le personnel de l'entreprise, avec une méthode calculant la somme des salaires à verser et une autre pour ajouter des employés, des responsables et commerciaux et les afficher**

III.8.2 Exercice 2

Examinez attentivement le code Java suivant. Certaines lignes provoquent une erreur.

```
1. class A {
2. protected void affiche()
   {
3. System.out.println(this)
   ;}
4. public void a() {
5. System.out.print("a de A
   dans ");
7. affiche();}
8. public void b() {
9. System.out.print("b de A
   dans ");
10. affiche() ; } }
11. class B extends A{
12. public void b() {
13. super();
14. System.out.print("b de
   B dans ") ;
15. affiche();}
16. public void c() {
17. super.c();
18. System.out.print("c de
   B dans ") ;
19. affiche();} }
```

```
20. public class Exercice
21. {
22. public static void
   main(String[] args) {
23. A a1 = new A();
24. A b1 = new B();
25. B a2 = new A();
26. B b2 = new B();
27. a1.a();
28. b1.a();
29. a2.a();
30. b2.a();
31. a1.b();
32. b1.b();
33. a2.b();
34. b2.b();
35. a1.c();
36. b1.c();
37. a2.c();
38. b2.c();
39. ((B) a1).c();
40. ((B) b1).c();
41. ((B) a2).c();
42. ((B) b2).c();}}
```

— **Donnez les numéros des lignes qui provoquent une erreur et indiquez si l'erreur se produit à l'exécution ou à la compilation.**

— **On supprime toutes les lignes qui provoquent une erreur dans ce programme. Donnez le résultat de son exécution en indiquant à côté de chaque ligne affichée le numéro de ligne de l'appel dans main( ) qui lui correspond.**

III.8.3 Exercice 3

1. **Écrivez une classe CompteDecouvert en se basant sur la classe Compte du chapitre précédent, cette classe représentera un compte ou le client peut re-**

tirer un montant maximum même s'il n'a pas de solde dans sans compte (Le solde devient négatif).

2. **Écrivez une classe CompteInteret à partir de Compte qui représente un compte avec intérêt bancaires où un montant est ajouté chaque année selon un taux au solde (si celui-ci dépasse un montant minimum).**
3. **Écrivez la classe CompteInteretDecouvert**
4. **Pensez à redéfinir ou surcharger les méthodes héritées et d'écrire les nouvelles**
5. **Voici un morceau de code avec des inconnues :**

```
class ExoTD24{
XXX c1 = new YYY;
ZZZ c2 = new TTT;
c1.deposer(50.0);
c2.deposer(50.0);
c2.calculInteret(); }
```

Donnez toutes les combinaisons de classes possibles pour XXX, YYY, ZZZ et TTT (vous ne vous préoccuperez pas des paramètres des constructeurs pour YYY et TTT)

CHAPITRE IV

Classes abstraites et interfaces

IV.1 Classes abstraites

IV.1.1 Définition

Définition :

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 (héritage).(Poo et al., 2007; Gervais, 2014)

Conseil :

On peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour les classes descendantes sous forme de méthodes abstraites dont on est alors sûr qu'elles existeront dans toute classe dérivée mais dont on ne sait pas encore leur implémentation.(Cooper, 1997)

Remarque : Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée et on peut trouver des méthodes dites *abstraites* qui fournissent uniquement la signature (le Nom de méthode, les paramètres de la méthode et le type de retour).

IV.1.2 Syntaxe de déclaration

Le mot clé abstract s'applique aux méthodes et aux classes.

```
public abstract class NomClasseAbstraite {
.....
public abstract typederetour NomMethodeAbstraite(type paraml, type
    param2.)
}
public class NomClasseFille étend NomClasseAbstraite { public
    typederetour NomMethodeAbstraite(type paraml, type param2,...) {
    .......
//On définit le code source de la méthode abstraite dans la classe
    descendante.
}
}
```

IV.1.3 Exemple de classe abstraite

Exemple :

```
public abstract class Etudiant {
.....
public abstract double calculMoyenne(); // méthode déclarée sans
    code source
}
public class Etudiantlet2 extend Etudiant {
 .....
 public double calculMoyenne() {
 return(notel+note2)/2);
 }
 }
  public class Etudiant3 extend Etudiant {
   .....
   public double calculMoyenne(){
    return(( (notel*2)+notePFE)/3);
    }}
```

IV.1.4 Remarques sur les classes abstraites

- Une classe possédant une méthode abstraite est forcément une classe abstraite.(Poo et al., 2007)
- Une classe abstraite peut ne pas contenir de méthode abstraite.(Poo et al., 2007)
- Une méthode abstraite ne doit pas contenir de code source : juste la déclaration de la méthode. Elle correspond à une méthode dont on veut forcer l'implémentation dans une sous classe.(Poo et al., 2007)
- L'abstraction permet une validation du code : une sous classe sans le modificateur abstract et sans définition explicite d'une ou des méthodes abstraites génère une erreur de compilation.(Poo et al., 2007)
- On ne peut utiliser le mot clés *abstract* avec les attributs.(Poo et al., 2007)

IV.2 Les interfaces

IV.2.1 Définition d'une interface

Définition :

Une interface est une classe entièrement abstraite. Elle n'implantant aucune méthode et aucun champ (sauf les constantes).(Permanand, 2013; Gervais, 2014)

Remarque :

- **Une interface définit les entêtes d'un certain nombre de méthodes, ainsi que des constantes**
- **Une classe peut implémenter plusieurs interfaces (alors qu'une classe ne pouvait dériver que d'une seule classe abstraite).**

IV.2.2 Syntaxe d'une interface

Les interfaces se déclarent avec le mot clé *interface*

```
interface I {
void f (int n) ; //abstract public facultatif
void g (); //abstract public facultatif
}
```

IV.2.3 Implémentation d'une interface

Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot clé *implements*.(Nell et al., 2013)

```
interface publique I1 {...}
interface publique 12 {...}
classe A implémente I1, 12 {
...
//A doit définir les méthodes de I1 et I2
}
```

IV.2.4 Exemple d'interface

Exemple :

```
public interface InterfaceEtudiant {
public void AfficherEtudiant();
public void saisir()
public double calculmoyenne();
public void publicMoyenne ();
}
public class Etudiant implements InterfaceEtudiant {
.........
// il faut le code de toutes les méthodes définies dans
   l'interface.
//sinon une erreur sera affichée indiquant qu'il y a une méthode
   non encore implémentée.
}
```

Remarque :

- **Une classe qui implémente une interface doit définir le code source de toutes les méthodes définies dans l'interface. Une erreur sera affichée si le programmeur oublie d'implémenter une des méthodes définies dans l'interface.**
- **Si la classe qui implémente une interface est définie avec *abstract*, Le programmeur peut ignorer certaines méthodes qui doivent être implémentées forcement dans les classes descendantes qui héritent de la classe abstraite. Sinon une erreur sera affichée indiquant qu'une méthode n'est pas encore implémentée.**
- **Toutes les méthodes d'une interface sont abstraites : elles sont implicitement déclarées comme telles même si le programmeur ne l'indique pas.**
- **Les seuls attributs que l'on peut définir dans une interface sont des attributs de classe qui doivent être constantes : elles sont donc implicitement déclarées avec le modificateur *static* et *final* même si le programmeur utilise d'autres modificateurs.**
- **Une interface peut être d'accès public ou package. Si elle est publique, toutes ses méthodes sont implicitement publiques même si elles ne sont pas déclarées avec le modificateur public. Si elle est d'accès package, il s'agit d'une interface d'implémentation pour les autres classes du package et ses méthodes ont le même accès package : elles sont accessibles à toutes les classes du packages.**

IV.3 Choix entre les interfaces et classes abstraites

Avant de choisir d'utiliser une interface ou une classe abstraite, le programmeur doit prendre en considération les remarques et conseils suivants (Poo et al., 2007) :

1. **Créer une classe qui n'étend rien quand elle ne réussit pas le test du EST-UN pour tout autre type.**
2. **Créer une sous-classe uniquement si vous voulez obtenir une version plus spécifique d'une classe, redéfinir ou ajouter des comportements.**
3. **Utiliser une classe abstraite quand vous voulez définir un patron pour un groupe de sous-classes et qu'une partie du code est utilisée par toutes les sous-classes.**
4. **Utiliser une interface pour définir un rôle que d'autres classes puissent jouer, indépendamment de leur place dans la structure d'héritage.**

IV.4 Exercice

On propose de pouvoir comparer des objets de différentes classes au moyen d'une conversion vers les nombres entiers. Pour cela on va utiliser une interface avec la méthode de conversion.

```
Interface Convertible{
int toInt();
}
```

— **Modifiez les classes Compte et Date vues en TD pour qu'elles implémentent cette interface.**

— **Écrivez une classe de test proposant des méthodes statiques pour comparer deux objets convertibles : une pour le test plus grand strict, une pour le test plus petit strict, une pour le test d'égalité, en comparant les entiers obtenus par conversion.**

— **Ajouter à la classe précédente une méthode statique permettant de trier en ordre croissant un tableau d'objets convertibles.**

CHAPITRE V

Gestion des exceptions

V.1 Exemple introductif

Soit le code suivant :

Exemple :

```
public class Div1 {
public static int divint (int x, int y) {
return (x/y);
}
static void main (String [] args) {
int c=0,a=1,b=0;
c= divint(a,b);
System.out.println("res: " + c);
System.exit(0);
}
}
```

Quand on exécute ce code, la machine va se planter, on aura ce message :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Div.divint(Div.java:3)
at Div.main(Div.java:7)
```

Pourquoi ?

On dit qu'une exception division par zéro s'est déclenchée. Ce code s'exécute dans tous les cas de b sauf pour b=0, b=0 est un cas anormal pour ce code
Une erreur s'est produite à la ligne 7 de la méthode main (la méthode divint), plus exactement à la ligne 3 (la division de x/y) ; elle correspond à une division par zéro. Des erreurs peuvent se produire dans d'autres circonstances, comme une ouverture d'un fichier inexistant, l'index d'un tableau qui déborde etc.

V.2 Définitions

Définition :

Une exception est un signal qui indique qu'un événement anormal est survenu dans un programme.(Gervais, 2014)

Définition 2 :

Une exception est un événement exceptionnel risquant de compromettre le bon déroulement du programme. Un débordement de tableau ou de pile, la lecture d'une donnée erronée ou d'une fin de fichier prématurée constituent des exemples d'exceptions.Huw (2020); Gervais (2014)

Remarque : La récupération (le traitement) de l'exception permet au programme de continuer son exécution.

Attention : En plus des exceptions prédéfinis, l'utilisateur peut créer ses propres exceptions(Voir Figure)V.1).

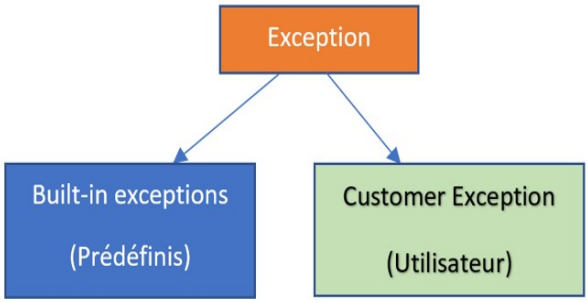

FIGURE V.1 – Types d'exceptions

V.3 Types d'exceptions

En Java, les exceptions sont de véritables objets Ce sont des instances de classes qui héritent de la classe Throwable comme le démontre la Figure V.1)(Gilbert, 1996)

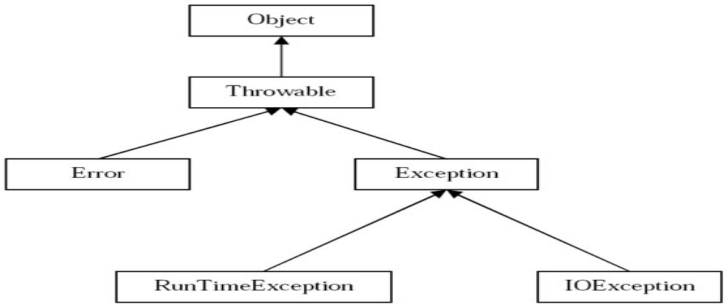

FIGURE V.2 – Hiérarchie des exceptions en Java

- **Throwable est la classe de base, à partir de laquelle vont dériver toutes les exceptions.**
- **Erreur : indications de problèmes irrécupérables dus au matériel ou au système d'exploitation. Elle gère les erreurs liées à la machine virtuelle Elles sont rares et fatales comme *OutOfMemoryError***
- **Exception : erreurs résiduelles dans le programme qui peuvent être traitées par une sortie propre ou une récupération (arithmétique, pointeurs, index, i/o, etc.).**
- **RunTimeException : regroupe les erreurs de base (ArithmeticException etc.)**
- **IOException : regroupe les erreurs entrée/sortie.**

V.4 Quelques exceptions prédéfinies

Il existe une multitude d'exceptions prédéfinis en Java Bole (2021) :

1. **NullPointerException : accès à un champ ou appel de méthode non statique sur un objet valant null. Utilisation de length ou accès à une case d'un tableau valant null.**
2. **ArrayIndexOutOfBoundsException : accès à une case inexistante dans un tableau. ou accès au ieme caractère d'un chêne de caractères de taille inférieure à i. ou création d'un tableau de taille négative.**
3. **NumberFormatException : erreur lors de la conversion d'un chêne de caractères en nombre.**
4. **ArithmeticException : se déclenche quand une condition exceptionnelle est rencontrée dans une expression arithmétique**

V.5 Gestion des exceptions dans un programme

V.5.1 Mécanisme du traitement d'exception

Lorsqu'une exception est levée dans une méthode donnée, les instructions qui suivent le lancement de l'exception, se trouvant dans cette méthode, sont ignorées. L'exception peut-être attrapée par un bloc catch (s'il existe un try) se trouvant dans cette méthode. Si l'exception n'a pas été capturée, le traitement de cette exception remonte vers la méthode appelante, jusqu'à être attrapée ou bien on est arrivé à la fin du programme (ici nous nous trouvons au niveau de la méthode main). (Huw, 2020; Cooper, 1997)

V.5.2 Définir de nouvelle exceptions (Custom Exceptions)

Afin de définir une nouvelle sorte d'exception, on crée une nouvelle classe en utilisant une déclaration de la forme suivante (Bole, 2021; Poo et al., 2007) :

```
class NouvelleException extends Exception {}
```

ou bien par :

```
class NouvelleException extends RuntimeException {}
```

En général, la nouvelle classe appelle le constructeur à un paramètres chêne de caractère de la classe Exception par super(Message)

```
class NouvelleException extends Exception {
public NouvelleException (String Message)
{
super( Message)
}
}
```

V.5.3 Lancer une exception

Lorsque l'on veut lancer une exception, on utilise le mot clé throw suivi de l'exception à lancer, qu'il faut auparavant la créer avec new NomException(), de la même manière que lorsque l'on crée un nouvel objet par un appel à l'un de des constructeurs de sa classe. Ainsi lancer une exception de la classe NomException s'écrit (Bugayenko, 2016) :

```
throw new NomException ();
```

Remarque : Lorsqu'une exception est levée, l'exécution normale du programme s'arrête et on saute toutes les instructions jusqu'à ce que l'exception soit rattrapée ou jusqu'à ce que l'on sorte du programme.
Par exemple, si on considère le code suivant :

Exemple :

```
public class Arret {
public static void main(String [] args) {
int i=0;
System.out.println("OK 1"); // 1
if (i==0) throw new Stop();
System.out.println("OK 2"); // 2
System.out.println("OK 3"); // 3
System.out.println("OK 4"); // 4
}
}
class Stop extends RuntimeException {};
```

l'exécution de la commande java Arret produira l'affichage suivant :

Résultat d'exécution :

```
> java Arret
OK 1
Exception in thread "main" Stop at Arret.main(Arret.java:5)
```

C'est-à-dire que les instructions 2, 3 et 4 n'ont pas été exécutées. Le programme se termine en indiquant que l'exception Stop lancée dans la méthode main à la ligne 5 du fichier Arret.java n'a pas été rattrapée.

V.5.4 Rattraper une exception : le bloc try catch

Le rattrapage d'une exception en Java se fait en utilisant la construction (Budd, 1999; Cooper, 1997) :

```
try {
... // 1
} catch (UneException e) {
... // 2
}
.. // 3
```

Le code 1 est normalement exécuté.
Si une exception est lancée lors de cette exécution, les instructions restantes dans le code 1 sont sautées. Si la classe de l'exception est UneException alors le code 2 est exécuté.
Dans le code 2, on peut faire référence à l'exception en utilisant le nom donné à celle-ci après sa classe (ici le nom est e).
Si la classe de l'exception n'est pas UneException, le code 2 et le code 3 sont sautés
Soit le code suivant :

Exemple :

```
public class Arret {
public static void main(String [] args) {
int i=0;
try {
System.out.println("OK 1"); // 1
if (i==0) throw new Stop();
System.out.println("OK 2"); // 2
} catch (Stop e) {
System.out.println("OK 3"); // 3
}
}
}
class Stop extends RuntimeException {};
class Stop2 extends RuntimeException {};
```

Ce code,produit l'affichage suivant lorsqu'il est exécuté :

Résultat d'exécution :

```
> java Arret
OK 1
OK 3
OK 4
```

En revanche le programme suivant, dans lequel on lance l'exception Stop2, l'affichage devient :

Résultat d'exécution :

```
> java Arret
OK 1
Exception in thread "main" Stop2
at Arret.main(Arret.java:5)
```

V.5.5 Rattraper plusieurs exceptions

Il est possible de rattraper plusieurs types d'exceptions en enchaînant les constructions catch (Permanand, 2013; Bugayenko, 2016) :

Exemple :

```
public class Arret {
public static void main(String [] args) {
int i=0;
try {
System.out.println("OK 1"); // 1
if (i==0) throw new Stop();
System.out.println("OK 2"); // 2
} catch (Stop e) {
System.out.println("OK 3"); // 3
}
catch (Stop2 e) {
System.out.println("OK 3_2"); // 3
}
System.out.println("OK 4_2"); // 4
}
}
class Stop extends RuntimeException {}
class Stop2 extends RuntimeException {}
```

A l'exécution, on obtient :

Résultat d'exécution :

```
> java Arret
OK 1
OK 3_2
OK 4
```

V.5.6 Déclaration throws

Le langage Java, demande à ce que soient déclarées les exceptions et qui peuvent être lancées dans une méthode sans être rattrapées dans cette même méthode. Cette déclaration est de la forme (Budd, 1999) :

```
throws Exception1, Exception2, ...
```

et se place entre les arguments de la méthode et l'accolade ouvrant marquant le début du corps de la méthode. Cette déclaration n'est pas obligatoire pour les exceptions de la catégorie Error ni pour celles de la catégorie RuntimeException (Unchecked exceptions) Soit le code suivant :

Exemple :

```
public class Arret {
static int lance(int x) throws Stop {
if (x < 0) {
throw new Stop();
}
return x;
}
public static void main(String [] args) {
int y = 0;
try {
System.out.println("OK 1");
y = lance(-2);
System.out.println("OK 2");
} catch (Stop e) {
System.out.println("OK throws");
}
System.out.println("y vaut "+y);
}}
class Stop extends RuntimeException {};
```

Le résultat est :

Résultat d'exécution :

```
      > java Arret
OK 1
OK 333
y vaut 0
```

V.6 Exercices

V.6.1 Exercice 1

Considérons la classe *ClasseFact* définie ci-dessous.

Programme Java	Exemples d'exécutions :

`public class ClasseFact { public static int fact(int k) { int f=1; // f=k! for (int i=1; i<=k; i++) f=f*i; return f; } public static void main(String[] args) {...// }`	`> java ClasseFact 4`
	`4! = 24`
	`> java ClasseFact -2`
	`-2! = 1`
	`> java ClasseFact 1.2`
	`Exception in thread "main"`
	`java.lang.NumberFormatException:1.2`
	`> java ClasseFact`
	`Exception in thread "main"`
	`java.lang.ArrayIndexOutOfBoundsException`

- **Écrire la méthode main() qui doit récupérer un entier n sur la ligne de commande, appelle la méthode *fact(n)* et ensuite affiche n !.**
- **Est-il nécessaire de créer un objet de type ClasseFact et appeler la méthode fact()**
- **Modifiez la méthode *fact(n)* pour qu'elle puisse générer une exception de type *ExceptionNombreNegatif* (classe à définir) si on veut calculer la factorielle d'un entier négatif.**
- **Réécrire la méthode main() afin de traiter toutes les exceptions mentionnées sur les exemples d'exécutions ci-dessus.**

V.6.2 Exercice 2

- **Que fait la classe suivante :**
- **Modifier la classe de façon à ce que la méthode rr lance une exception dans le cas où le double qui lui est passé en paramètre est négatif. Cette exception sera attrapée et traitée par la méthode main.**

```
public class CC {
private double e;
public CC(double e) {
this.e = e;
}
public double rr(double n) {
double ss, c, a, b;
a = 0;
b = n;
ss  =(a+b)/2;
while (b - a > e) {
c = ss * ss;
if (c > n) b = ss;
      else a = ss;
ss = ( a + b) / 2;
}
return ss;
}
public double gete () {
return e;
}
public void sete (double e) {
this.e = e;
}
```

V.6.3 Exercice 3

Qu'affiche le programme suivant si l'entier entré est 3 ? Même question avec 0, 1 et 2.

```
class Test10{
static void methode1(int p) throws Exc1, Exc2{
System.out.println("Debut d'execution de methode1");
if (p == 0){
throw new Exc1 ();
}
System.out.println("Milieu d'execution de methode1");
if(p ==1){
throw new Exc2 ();
}
System.out.println("Fin d'execution de methode1");}
static void methode2(int p) throws Exc1, Exc2, Exc3{
System.out.println("Debut d'execution de methode2");
if (p == 2){
throw new Exc3();}
System.out.println("Milieu d'execution de methode2");
methode1(p);
System.out.println("Fin d'execution de methode2");
}
static void methode3(int p) throws Exc1, Exc3{
System.out.println("Debut d'execution de methode3");

try{
methode2(p);
catch(Exc2 ex){
System.out.println("Debut d'execution de methode3");
}
System.out.println("Fin d'execution de methode3");
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Saisissez un entier : ");
```

```
int i = sc.nextInt();
methode3(i);
}}
class Exc1 extends Exception{}
class Exc2 extends Exception{}
class Exc3 extends Exception{}
```

Conclusion

La programmation orienté-objet *P.O.O* est de loin le paradigme qui a marqué le plus la programmeurs dans les derniers décennies. Ce paradigme a complètement changé la donne en permettant d'écrire des programmes tés complexes comme les jeux et les interfaces graphiques.

Dans ce cours, nous avons présenté les notions de bases de la *P.O.O* ainsi que la gestion des exceptions qui est devenue une partie indispensable de n'importe quel programme de nos jours.

Nous avons essayé de s'éloigner des largages de programmation surtout dans les définitions et les principes de base car la *P.O.O* est un manière de pensée d'un programmeur et elle n'est pas lié à un langage donné même si *JAVA* a été utilisé pour les exemples conformément au canevas de la spécialité.

Bibliographie

Bole, J. (2021). ***Introduction to object-oriented programming : ultimate guideline of object-oriented programming language beginner.*** **.**

Budd, T. (1999). ***Understanding Object-Oriented Programming With Java : Updated Edition.*** **Pearson.**

Bugayenko, Y. (2016). ***Elegant Objects 1.0.*** **CreateSpace Independent Publishing Platform.**

Cooper, J. W. (1997). ***Principles of Object-Oriented Programming with Java.*** **Ventana Communications Group, Incorporated, 10th edition.**

Gervais, L. (2014). ***Apprendre la Programmation Orientée Objet avec le langage Java.*** **Eni.**

Gilbert, S. (1996). ***Object-Oriented Programming in Java.*** **Macmillan Publishing Co., Inc.**

Huw, C. (2020). ***The Little Java Book Of Adventure Game Programming : Learn Object Oriented Programming ? advanced coding techniques.*** **Dark Neon.**

Nell, D., Daniel T., J., and Chip, W. (2013). ***Object-Oriented Data Structures Using Java.*** **CreateSpace Independent Publishing Platform.**

Permanand, M. (2013). ***Fundamentals of Object-Oriented Programming in Java.*** **CreateSpace Independent Publishing Platform.**

Poo, D., Kiong, D., and Ashok, S. (2007). ***Object-Oriented Programming and Java.*** **Springer-Verlag, Berlin, Heidelberg.**

Watt, D. A. (1990). ***Programming Language Concepts and Paradigms.*** **Prentice-hall.**