Ministère de l'Enseignement supérieur et de la Recherche Scientifique
Université Abderrahmane Mira Bejaia
Faculté des sciences exactes
Département d'informatique

جامعة بجاية
Tasdawit n Bgayet
Université de Béjaïa

Polycopié de cours

Module : **Programmation Orientée Objet (POO)**

A l'usage des étudiants de 2ème année Licence informatique et classes préparatoires

Par Dr Houda EL BOUHISSI épouse BRAHAMI

2019 / 2020

Version 3.0

Table des matières

Introduction

La programmation orientée objet est un concept, et une façon d'organiser ses programmes. Par opposition à un langage orienté objet, un langage dit « procédural » signifie qu'il s'utilise avec des variables simples et des fonctions.

Dans un programme orienté objet, on conçoit un programme à partir d'objets. Par définition, on appelle objet une variable complexe c'est-à-dire qu'elle est elle-même composée de variables et de fonctions, contrairement aux variables « scalaires » que sont les caractères, les entiers, les flottants.

L'intérêt de l'orienté objet est de créer ses objets définis comme on le veut, composés de variables (dites « attributs ») et de fonctions (dites « méthodes ») de notre choix.

Ce document s'adresse principalement aux étudiants de deuxième année de la filière informatique, qui n'ont pas encore eu de cours là-dessus, ainsi qu'aux novices qui ne maitrisent pas ce domaine.

Il se veut clair et simple et chaque point est accompagné d'exemple. Ce document a pour vocation d'initier le lecteur, de l'intéresser et de lui donner envie d'en apprendre plus, peut-être par lui-même,

Ainsi ce document est organisé comme suit :

- **Chapitre 1** : Ce chapitre est une introduction à la programmation orientée objet.
- **Chapitre 2 :** Le deuxième chapitre présente des notions de base du langage java étant donné qu'il représente l'essence de la POO.
- **Chapitre 3 :** Ce chapitre initie le lecteur à l'orienté objet, création de classes, utilisation de méthodes, constructeurs, destructeurs, …etc.
- **Chapitre 4 :** Ce chapitre présente les deux principes fondamentaux de l'orienté objet, à savoir l'héritage et le polymorphisme.
- **Chapitre 5 :** Le cinquième chapitre aborde les classes abstraites et les interfaces et explique le rôle de chaque concept et où et comment l'utiliser.

Aussi, une série d'exercices avec leurs solutions est mise à la disposition du lecteur afin de tester ses connaissances.

Chapitre

1

Introduction à la programmation orientée-Objet

A l'issue de ce chapitre, l'apprenant sera capable de reconnaitre quelques paradigmes de programmation.

1.1. Introduction

Chaque langage de programmation appartient à une "famille" de langages définissant une approche ou une méthodologie générale de programmation appelée « paradigme de programmation ».

Un paradigme de programmation désigne la façon de raisonner et d'implémenter une solution à un problème en programmation. Il existe au moins 20 paradigmes différents.

Dans ce qui suit, nous allons citer les paradigmes les plus connus.

1.2. Paradigmes de programmation

1.2.1. Programmation impérative

La programmation impérative (procédurale) est la plus courante. Un programme impératif n'est rien de plus qu'une suite d'instructions. Il peut donc être assimilé en quelque sorte à une recette de cuisine, il suffit de lire les instructions et les exécuter les unes après les autres.

1.2.2. Programmation fonctionnelle

En opposition avec la programmation impérative, la programmation fonctionnelle rejette la mutation des données et les effets de bord (modification d'un état autre que la valeur de retour de la fonction). Tout calcul est donc fait sous forme d'appel à des fonctions.

1.2.3. Programmation logique

Ce paradigme est très proche des mathématiques, puisqu'il est à l'origine utilisé pour les démonstrations automatiques de théorèmes. Le principe est de définir une liste de faits (axiomes) et de règles de logique qui leur associent des conséquences.

1.2.4. Programmation orientée-objet

Le paradigme de programmation objet est un dérivé de la programmation impérative. Le principe est ici de programmer des objets, qui représentent un concept, un objet physique ou un ensemble de données et les actions qui lui sont rattachées, plutôt que de ne voir ces ensembles uniquement d'un point de vue procédural.

La POO est née suite à la « crise du logiciel » afin de permettre une maintenance des codes plus efficace. De nombreux langages de programmation modernes supportent la POO, comme le langage java.

1.3. Types abstraits de données(TAD)

Un type abstrait de données (quel que soit le langage) c'est le rassemblement de toutes les entités ayant un lien « logique » entre elles. Ce type est clairement identifié par un nom (identificateur). Un TAD est un ensemble d'objets caractérisés par les opérations qui leur sont applicables.

Implicitement il lui est attaché des propriétés sous-jacentes et même explicitement (spécifications) souvent bien décrites.

Exemples:

- Ensemble : ajouter, supprimer, appartient…
- Fenêtre : tracer, agrandir, déplacer …

Ce qui importe, c'est le concept véhiculé par un objet (et non la structure de données sous-jacente).

Chapitre

2

Syntaxe du langage Java

A l'issue de ce chapitre, l'apprenant sera capable de maîtriser la syntaxe du langage java et utiliser les structures alternatives et répétitives.

2.1. Introduction au langage Java

JAVA est un langage de programmation orientée-objet simple et puissant avec une syntaxe qui ressemble beaucoup à celle du langage C++. En revanche, ces deux langages sont très différents dans leur structure (organisation du code et gestion des variables).

Le langage Java est créé par Sun Microsystems, Inc. en 1991.Il a été conçu par James Gosling, PatrickNaughton, Chris Warth, Ed Frank et MikeSheridan chez Sun Microsystems, Inc.

L'avantage principal de Java est sa portabilité, Java a été conçu de manière à «écrire une fois et exécuter partout». Java Virtual Machine(JVM) joue un rôle fondamental pour la réalisation de ce concept. La JVM est l'environnement dans lequel les programmes Java s'exécutent, il s'agit d'un logiciel qui est implémenté au sein des systèmes d'exploitation et matériels. Lorsque le code source (fichiers. Java) est compilé, il est traduit en code octet, puis placé dans des fichiers (. Classe), à ce moment-là la JVM s'occupe de l'exécution de ces bytecode.

Comme tout langage de programmation, un programme java est composé d'un ensemble d'instruction et respecte un certain nombre de règles :

- Les instructions simples se terminent par ";".
- Un bloc d'instruction est délimité par des accolades {} : les instructions à l'intérieur du bloc sont à exécuter séquentiellement.

Par exemple :

```
{instruction 1
    Instruction 2
    Instruction 3
     .....
    Instruction n }
```

Avant d'entamer le développement en Java, il faut préalablement préparer votre machine en installant les logiciels nécessaires. Pour cela, vous devez télécharger les outils suivants :

- Java Developpement Kit (JDK) : environnement de développement Java développé par la société Sun Microsystème
- Java Runtime Environnement (JRE) : outil nécessaire pour l'exécution de tout programme java.
- Un Integrated development environment (IDE) : outil pour écrire un programme java, en général, un éditeur de texte peut suffire mais ils existent des IDE spécialement conçu pour java et qui sont très sophistiqués, on peut en citer les meilleurs : Eclipse, Netbeans, – Jdevelopper, – JBuilder… –et bien sûr d'autres…)**.**

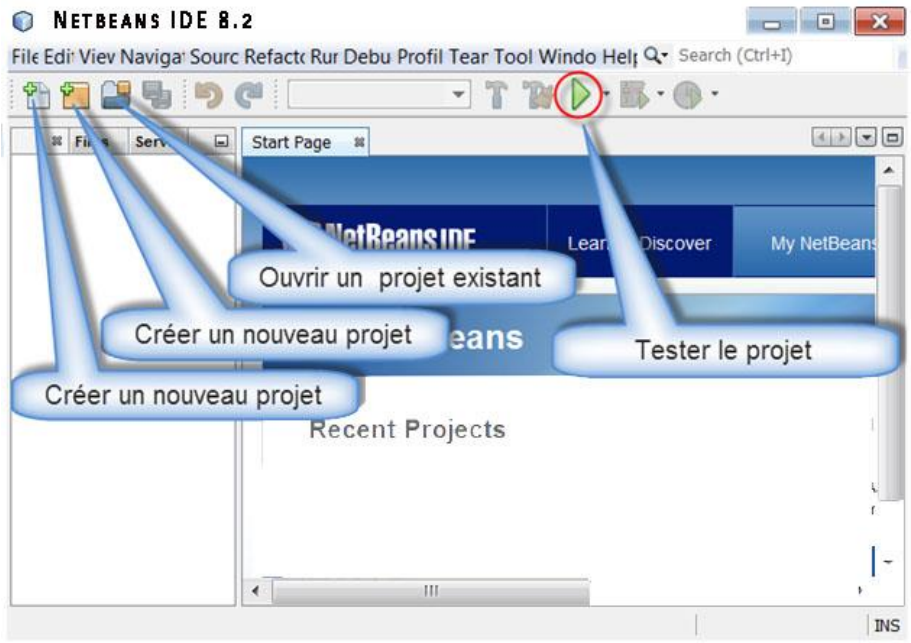
Netbeans (logiciel open source distribué sous différentes versions, il existe une version qui intègre le **JDK** et le **JRE** et vous dispense des étapes de configurations fastidieuses.

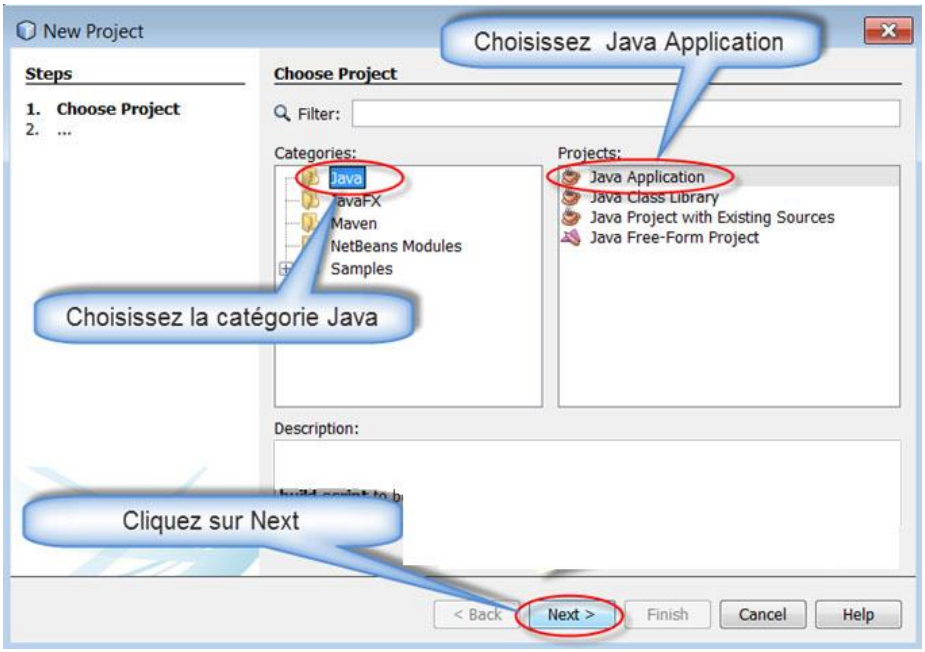
2.2 Premier programme

Après installation, votre machine sera dotée automatiquement des outils **JDK, JRE** et **l'IDE Netbeans** vous aurez donc un environnement complet pour développer en java.

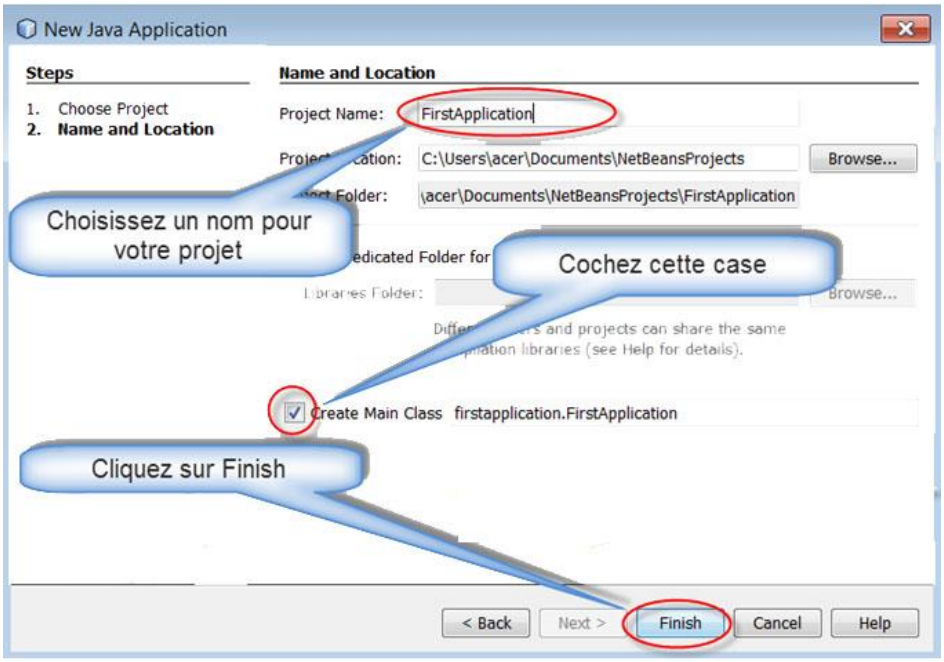
Nous allons maintenant créer notre premier programme Java :

- Démarrez Netbeans et cliquez sur **Fichier → New Project**.
- Sélectionnez ensuite la catégorie **java,** puis sélectionnez ensuite **Java Application** et cliquez sur suivant**.**



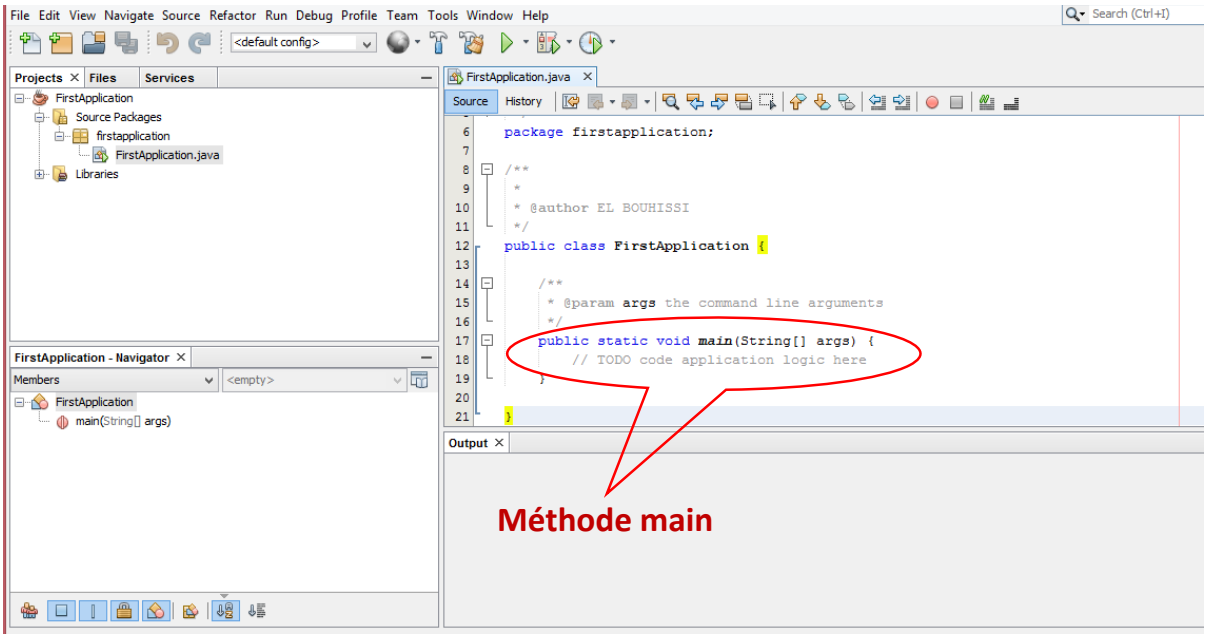


- Saisissez ensuite le nom de votre projet et son emplacement, par exemple : premierPas dans la zone **Project name**.



- Choisissez ensuite un emplacement en cliquant sur **le bouton parcourir** ou laisser celui par défaut si vous le souhaitez.
- Cliquez ensuite sur le bouton **Finish**.

Vous obtenez la vue suivante :



- La ligne du code : *// TODO code application logic here v*ous indique que votre code java doit être tapé à cet endroit.

Nous allons maintenant écrire un petit programme permettant d'afficher le message *Bonjour étudiants de Béjaia*, pour cela, nous allons utiliser la fonction : System.out.println("votre texte ici") ; Sans oublier le point-virgule ; à la fin du code.

Nous allons mettre cette instruction à la place de la ligne de code :
// TODO code application logic here

Pour exécuter le code, il suffit de cliquer sur le bouton vert sous forme de petit triangle de la barre d'outils en haut : 

Il est à noter que Pour qu'un programme en Java puisse être exécuté, une méthode est absolument indispensable, c'est la méthode main (on reviendra sur les méthodes au chapitre suivant). Elle se déclare toujours de la manière suivante :

```
public static void main(String[] args){
   //Ici se trouve le code à exécuter, par exemple :
   System.out.println("Bonjour étudiants de Béjaia");
}
```

Cette petite méthode, une fois exécutée, affichera le message Bonjour étudiants de Béjaia dans la console.

En effet, la méthode System.out.println() permet d'afficher du texte dans la console puis de passer à la ligne. Si vous ne voulez pas aller à la ligne, il faut utiliser la méthode System.out.print().

Il est possible de faire appel à d'autres méthodes dans celle-ci, il n'est donc pas nécessaire de faire tout le programme dans cette méthode, et ce n'est d'ailleurs pas conseillé.

2.3. Variable

Le langage Java offre la possibilité d'utiliser des types primitifs comme « int », « float », « boolean » et « char ». Ce ne sont pas des objets. Ils ne sont pas instanciés mais simplement déclarés.

Voilà les principaux types primitifs en Java :

- **Entiers** : Le format par défaut d'un entier est l'int qui en Java est toujours code sur 32bits (de -231 à 231-1 soit de -2147483648 à 2147483647).

- **Flottants** : en java « float » ou « double ». Attention ! le format d'un flottant est par défaut le double et non le float comme en C, ce qui conduit à de nombreuses erreurs de compilation.

- **Booléens** : en java « boolean» (8bits (faux 1 bit))

- **Caractères** : char ()

- **short** : -32 768 à 32767

- **long** : -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807

- **byte** : -128 à 127

Java exige que chaque variable soit déclarée, faute de quoi le compilateur affiche un message d'erreurs. Pour déclarer une variable il suffit de taper son nom précédé de son type, par exemple :

```
int  n ;  // déclaration d'une variable du type entier
byte  b ;  // déclaration d'une variable du type byte
int  i, j ;  // déclaration de deux  variables du type entier
```

En général, une variable déclarée est suivie généralement d'une valeur initiale (initialisation).

```
int  n  ;  // déclaration d'une variable du type entier
n  =  10  ; // affectation d'une valeur à la variable
// ou simplement :
int i = 10 ; // // déclaration d'une variable du type entier et
affectation en même temps
```

2.4. Opérateurs

Voilà les principaux opérateurs arithmétiques et logiques utilisés en java avec leurs descriptions (Figure 2.1) :

Opérateur	Description
+	Addition
-	Soustraction
*	Produit
/	Quotient de division euclidienne
%	Reste de division euclidienne
&&	"et" conditionnel
||	"ou" conditionnel
=	Affectation
==	Comparaison de valeurs
< <=	Inférieur - inférieur ou égal
> >=	Supérieur - supérieur ou égal
!=	Différent de
!	"non" logique
~	Complément à deux (inverse le signe)
&	"et" arithmétique
|	"ou" arithmétique
^	"ou" exclusif (xor) arithmétique
<<	Décalage à gauche
>>	Décalage à droite signé
>>>	Décalage à droite non signé

Fig. 2.1 : Principaux opérateurs arithmétiques et logiques

2.5. Saisie clavier

Afin de faire interagir l'utilisateur avec son clavier, et de permettre à l'apprenant de faire les essais sur machine, nous allons expliquer comment récupérer les valeurs saisies au clavier.

Le langage Java est doté d'une classe spéciale nommée Scanner. Pour utiliser cette classe nous devrions :

- Importer le package : `import java.util.Scanner ;`
- Faire une instanciation : `Scanner maSaisie = new Scanner(System.in) ;`
- Guider l'utilisateur avec un message qui s'affiche à l'écran : System.out.println(" Tapez votre message ") ;
- Récupérer la saisie clavier dans une variable : String maphrase = maSaisie.nextLine() ;
- Afficher le résultat à l'écran : System.out.println(" vous avez saisie " + maphrase) ;

Et voici le code final :

```
import java.util.Scanner;

public class Essai {
      public static void main(String[] args) {
      Scanner maSaisie = new Scanner(System.in) ;
      System.out.println("Tapez votre message : ");
      String maphrase = maSaisie.nextLine() ;
      System.out.println(" vous avez saisie " +  maphrase  ) ;
      }
}
```

Ceci est un exemple d'introduction, le lecteur pourrait se documenter à propos de la récupération des données numériques ou autre types. Les notions de package et d'instanciation seront vues en détail au chapitre suivant.

2.6. Tableaux

Les tableaux sont utilisés pour stocker plusieurs valeurs dans une seule variable, au lieu de déclarer des variables distinctes pour chaque valeur. Pour déclarer un tableau, définissez le type de variable suivie de crochets :

```
String [] mois ; // Déclaration de tableau nommé mois de chaines de
caractères
```

Nous pouvons accédez à un élément de tableau en se référant au numéro d'index, par exemple mois[i].

i représente un index. Les index d'un tableau commencent par 0 : [0] est le premier élément. [1] est le deuxième élément, etc.

Un tableau multidimensionnel est un tableau contenant un ou plusieurs tableaux. Pour créer un tableau à deux dimensions, ajoutez chaque tableau dans son propre ensemble d'accolades :

```
int [] [] tab;
```

2.7. Chaines de caractères

On appelle chaîne toute suite de caractères contiguës, par exemple "bonjour" est une chaîne de longueur 7 (longueur de la chaîne = nombre de caractères de la chaîne). Une chaîne est peut être considérée comme une variable du langage Java et est déclarée à l'aide de l'instruction String.

```
String  maChaine = "bonjour" ;
//déclaration d'une chaine de caractère nommée maChaine et de valeur
= "bonjour"
```

Pour concaténer des chaines de caractères, on utilise l'opérateur + :

```
String auto="auto";
```

```
String bus="bus";

 public class Essai{
 public static void main(String[] args)
        if ( args.length==0) {
          System.out.println("Bonjour !!! ") ;
         }
// Suite du code
}
System.out.println(auto+ " " +bus);  // ce qui affiche  "auto bus
```

2.8 Structures alternatives

2.8.1 L'instruction if / else

L'instruction if est une instruction conditionnelle : elle exécute des portions de code en fonction d'une condition qu'elle soit vraie ou fausse (la condition doit calculer une valeur booléenne).

Dans beaucoup de langages de programmation (Visual Basic, Pascal, ...) l'instruction « if » est aussi associée au mot clé « then » mais ce n'est pas le cas en Java. Effectivement, Java s'inspire de la syntaxe du langage C dont il en reprend bon nombre d'aspects. C'est donc aussi le cas pour le « if » et en C le mot clé « then » n'existe pas.

La condition doit être globalement mise entre parenthèses, ce qui retire l'obligation d'avoir un autre mot clé pour débuter le bloc de code dans le cas où la condition est vraie.

Si vous oubliez les parenthèses autour de la condition, une erreur de compilation sera produite. Par contre, les accolades autour du code à exécuter, dans le cas où la condition est vraie, sont facultatives. Dans ce cas, une seule et unique instruction pourra être exécutée.

Un autre mot clé est utile au if : le mot clé « else ». Il permet de lancer l'exécution d'un code dans le cas où la condition renvoie false. De même que pour la partie de code à exécuter dans le cas où la condition calcule true, les accolades peuvent être mises ou non.

Voilà un exemple d'une instruction if associée à un else :

En voici un exemple :

```
  public class Essai2{
  public static void main(String[] args)
         if ( args.length==0) {
           System.out.println("Bonjour !!! ") ;
          } else { System.out.println("Bonsoir !!! ") ;
 // Suite du code
```

2.8.2 L'instruction switch / case

Comme l'instruction if, le switch permet de déclencher des traitements en fonction des valeurs d'une expression. Cette instruction est similaire à plusieurs if imbriques. Cependant, il y a des différences notables entre les deux instructions :

La condition d'un "if" doit calculer un booléen. Dans le cas d'un "switch", l'expression doit renvoyer soit un entier (byte, short, int, long), soit un type énumère, soit une chaine de caractères ou encore un caractère.

Si l'expression calcule une valeur flottante (ou même une valeur booléenne), une erreur de compilation sera produite.

Un switch est plus optimise qu'un ensemble de if imbriques. Effectivement, dans le cas d'un "if", n tests seront effectués (ou n représente le nombre de if imbriques). Au contraire, un "switch" sur entier trouvera l'adresse en mémoire du code à exécuter pour chaque valeur en temps constant.

En voici un exemple :

```
public class SwitchExemple {

   public static void main(String args[]){
     int num=2;
     switch(num+2)
     {
        case 1:
         System.out.println("Case1: Valeur est: "+num);
        case 2:
         System.out.println("Case2: Valeur est: "+num);
        case 3:
         System.out.println("Case3: Valeur est: "+num);
        default:
         System.out.println("Default: Valeur est: "+num);
      }
   }
}
```

2.9. Structures répétitives

2.9.1 La boucle for

C'est l'instruction itérative la plus riche en informations à fournir et simple à comprendre. Un "for" attend trois expressions ainsi qu'une ou plusieurs instructions à répéter. Voici le format de l'instruction for :

```
for (initExp ; testExp; incExp ) {
Bloc d'instructions à exécuter
}
```

Et Voici la signification de chacune des expressions :

- **initExp** : Cette première expression est appelée expression d'initialisation. Effectivement, pour effectuer une boucle, nous aurons besoin de créer et d'initialiser une variable. Cette variable sera détruite une fois que l'instruction "for" sera terminée.
- **testExp** : Cette seconde expression doit effectuer un test (donc doit renvoyer un booléen). Si le résultat de cette expression est vrai, alors une nouveau tour de boucle sera effectué. Si l'expression renvoie false, alors la boucle s'arrête. C'est donc une expression de "rebouclage". Il est à noter que cette expression est évaluée avant chaque exécution du bloc de code à répéter. Si le premier coup l'expression renvoie directement false, alors aucun tour de boucle ne sera réalisé.
- **incExp** : Cette dernière expression permet d'incrémenter votre variable de boucle à la n de chaque tour de boucle.

Pour ce qui est du code à répéter, il peut y avoir une instruction ou plusieurs. S'il n'y en a qu'une, alors les accolades ne sont pas obligatoires.

Voici un exemple concret qui affiche les dix premiers entiers positifs (en commençant par un) :

```
public class Exemple {
public static void main(String  []  args ) {
      for( int   i = 1; i <= 10; i++ )
            System.out. println ( i ) ;
}
}
}
```

2.9.2 L'instruction while

L'instruction "while" est simple à utiliser. En effet, une seule expression est attendue par le while (la condition de rebouclage) contre trois pour le "for". En français, while signifie tant que, donc la boucle va tourner tant que l'expression de rebouclage calcule true.

Voici la syntaxe générale du while :

```
while ( testExp ) {
<Bloc d' instructions à executer>
}
```

Voilà un programme qui affiche tous les chiffres de 0 à 9, ce programme effectue deux fois une même boucle, mais la première fois la boucle est gérée par une instruction "for", alors que la seconde fois, c'est l'instruction "while" qui est utilisée. Dans les deux cas, on affiche tous les chiffres de 0 à 9.

```
public class Demo {
public static void main( String[]args){
// La première boucle est réalisée    via   l'instruction for
for (int  i=0; i<10; i++ ) {
System.out. print (i + "  ") ;
}
System.out. println () ;
// La seconde boucle est réalisée via l'instruction  while
int  i=0;
while( i<10 ) {
System.out. print ( i + " " ) ;
i++;
}
}
```

On peut donc aussi bien utiliser le "for" ou le "while" pour réaliser une boucle, il suffit de respecter la syntaxe, seulement, quand utiliser l'instruction "for" et quand utiliser l'instruction "while"?

Quand on connait à l'avance, le nombre de tours de boucle à exécuter, il est préférable d'utiliser l'instruction "for" car elle est alors plus lisible. Par contre, si on ne connait pas à l'avance le nombre de tours de boucle à exécuter, alors l'instruction "while" sera très certainement la plus adaptée. En conséquence, dans l'exemple précèdent, la version "for" devrait être appropriée.

2.9.3 L'instruction do while

L'instruction "do while" est très proche de l'instruction while. On peut littéralement la traduire par "faire tant que". La différence, par rapport à l'instruction "while", c'est que la condition est évaluée après exécution du corps de la boucle. Il en résulte que l'on fera au moins un tour de boucle, contrairement au while qui peut sortir immédiatement (car la condition est évaluée en premier). Voici la syntaxe générale de l'instruction "do while" :

```
do {
Bloc d'instructions à exécuter
while ( testExp ) ;
}
```

Et voilà un exemple qui affiche les nombres entiers de 10 à 2 avec la boucle do..while.

```
class DoWhileLoop {
    public static void main(String args[]){
         int i=10;
         do{
              System.out.println(i);
              i--;
         }while(i>1);
    }
}
```

2.10 Exercices

Exercice1

Ecrire un programme en java qui permet de tester l'âge d'une personne saisi à l'écran, il lui affiche mineur si l'âge est inférieur à 18 ans et majeur sinon.

Correction

```
import java.util.Scanner;
public class si {

	public static void main(String[] args) {
		System.out.println("Veuillez saisir votre âge !");
	Scanner age=new Scanner(System.in);
	int tonAge=age.nextInt();
	if ( tonAge < 18 ) {
System.out.println("vous êtes mineur"+" Votre âge est : "+tonAge);
	}
	else{
		System.out.println("vous êtes Majeur"+" Votre âge est
: "+tonAge);
	}
	age.close();
}
}
```

Exercice2

Ecrire un programme Java qui demande à l'utilisateur de saisir un nombre entier **n** et de lui afficher si le nombre tapé est premier ou non ?

Correction

```
import java.util.Scanner;
public class TestPrimalité {

  public static void main(String[] args) {
System.out.println("Tapez la valeur de l'entier n : ");
Scanner entier=new Scanner(System.in);
 int n=entier.nextInt();
  int i=2;
   while(n%i!=0){i=i+1;
         }
 if(i==n){
	System.out.println("L'entier " + n + " que vous avez tapé est
premier");
}
else{
	System.out.println("L'entier " + n + " que vous avez tapé
n'est pas premier car il est divisible par : " + i);
}
}
}
```

Exercice3

Ecrire un programme java qui demande à l'utilisateur de saisir son nom et de lui afficher son nom avec le message de bienvenue

Correction

```
import java.util.Scanner;
public class Bienvenue {

	public static void main(String[] args) {
   Scanner nom=new Scanner(System.in);
   System.out.println("Veuillez saisir votre nom : ");
   String nm=nom.nextLine();
   System.out.println("Bienvenue : " + nm);
	}
}
```

Exercice4

Ecrire un programme java qui demande à l'utilisateur de saisir un nombre entier et de lui afficher que le nombre est pair ou impair selon la valeur introduite.

Correction

```
import java.util.Scanner;
public class pairOuImpair {
	public static void main(String[] args) {
//affichage à l'écran d'un message demandant à l'utilisateur de
taper un nombre entier
	System.out.println("Veuillez saisir un nombre entier :");
//récupération de la saisie clavier à l'aide de la classe Scanner
	Scanner entier=new Scanner(System.in);
	int n=entier.nextInt();
//récupération du reste de la division euclidienne de n par 2
	int r=n%2;
	if(r==0){
		System.out.println("Le nombre "+ n +" que vous venez
de taper est pair");
	}
	else{
		{
			System.out.println("Le nombre  "+ n +" que vous
venez de taper est impair ");
		}
		entier.close();
	}
	}
	}
```

Exercice5

Ecrire un programme Java qui calcul la somme des 100 premiers entiers

Correction

```
public class SommeDes100PremiersEntiers {
	public static void main(String[] args) {
	   int j=0;
	   for(int i=1;i<=100;i++){
	       j=j+i;
	   }
System.out.println("La somme des 100 premiers entiers est  : " +j);
	}
}
```

Exercice6

Ecrire un programme Java qui demande à l'utilisateur de saisir un nombre entier n et lui affiche la somme des **n** premiers nombres entiers.

Correction

```
import java.util.Scanner;
public class sommeDesNpremiersEntiers {

	public static void main(String[] args) {
		Scanner n=new Scanner(System.in);
		System.out.println("Saisissez la valeur de N");
	   int N=n.nextInt();
	   int j=0;
	   for(int i=1;i<=N;i++){
		j=j+i;
	   }
System.out.println("La somme des " + N +" premiers nombres est  : "
+j);
}
}
```

Exercice7

Ecrire un programme Java qui demande à l'utilisateur de saisir un nombre entier n et de lui afficher successivement tous les nombres pairs qui sont inférieur ou égale n Améliorer le programme de façon qu'il affiche en plus le nombre des entiers pairs inférieur ou égale à n.

Correction

```
import java.util.Scanner;
public class NombrePairInferieurAn {
public static void main(String[] args) {
System.out.println("Veuillez saisir la valeur de n");
Scanner sc=new Scanner(System.in);
int n=sc.nextInt();
for(int i=0;i<=n;i++){
	if(i%2==0){
	System.out.println(i+" est un nombre pair inférieur ou égale
à " +n);
	}
}
}
}
```

```
import java.util.Scanner;
public class NombrePairInferieurAn {
public static void main(String[] args) {
System.out.println("Veuillez saisir la valeur de n");
Scanner sc=new Scanner(System.in);
int n=sc.nextInt();
int j=0;
for(int i=0;i<=n;i++){
	if(i%2==0){
	System.out.println(i+" est un nombre pair inférieur ou égale
à " +n);
```

```
        j+=1;
        }
}
System.out.println("_____________________________");
System.out.println("Le nombre d'entiers pair inférieur ou égale à "
+n +" est : " +j);
}
}
```

Exercice8

Ecrire un programme Java qui demande à l'utilisateur de saisir un nombre entier **n** inférieur ou égale à 9 et de lui afficher la table de multiplication de ce nombre.

Correction

```
import java.util.Scanner;
public class TableDeMultiplication {

public static void main(String[] args) {
 System.out.println(" Veuillez saisir la valeur de l'entier n :");
  Scanner sc=new Scanner(System.in);
  int n=sc.nextInt();
   System.out.println(" la table de multiplication de n est :");
for(int i=1;i<=10;i++){
        System.out.println(i+" x " + n +" = "+ i*n);
}
}
}
```

Chapitre

3

POO : Concepts de base

A l'issue de ce chapitre, l'apprenant sera capable d'appliquer les concepts objets de base de la programmation orientée-objet.

3.1. Classes, Objets et Références

Java est un langage orienté objet. La notion de classe est centrale dans ce langage et tout le code est contenu dans des classes. Seules les variables de type primitif ne sont pas des objets. Les types primitifs servent à créer des variables locales dans les méthodes et des attributs pour les classes. Les objets ne sont pas

manipulés directement mais à travers des références (pointeurs de références de manière automatique).

Dans les langages orientés objets, on crée des classes. Les classes constituent le concept de base de la programmation objet. Elles permettent de définir de nouveaux types de données qui doivent se comporter comme des types prédéfinis et dont les détails d'implantation peuvent être cachés aux utilisateurs de ces classes. Comme nous allons le voir dans ce chapitre, contrairement aux types prédéfinis, les classes peuvent être créées de manière hiérarchique : une classe est souvent une sous-classe d'une autre classe.

Un objet est une instance d'une certaine classe ; au lieu de parler d'une variable d'une certaine classe, on dira plutôt un objet d'une certaine classe.

En Java, on ne peut accéder aux objets qu'à travers une référence vers celui-ci. Une référence est, en quelque sorte, un pointeur vers la structure de donnée. On ne peut connaitre la valeur de la référence et on ne peut évidemment pas effectuer d'opérations arithmétiques sur les références. La seule chose permise est de changer la valeur de la référence pour pouvoir "faire référence" à un autre objet.

Une classe définit généralement :

- Les structures de données associées aux objets de la classe ; les variables désignant ses données sont appelées champs ou attributs.
- Les services que peuvent rendre les objets de cette classe qui sont les méthodes définies dans la classe. Les méthodes ressemblent aux fonctions.

On peut par exemple considérer un Etudiant comme un exemple de classe avec les attributs : matricule, nom, prenom et une méthode passerExamen, Mohamed, Farid et Fatima représentent des instances ou objets de cette classe.

Dans les programmes, un objet est une zone mémoire qui est pointée par une ou plusieurs références. Une fois l'objet instancié, il correspond en réalité à une zone mémoire donnée. Cette zone mémoire, pour être utilisée doit être référencée.

Par exemple, Mohamed est une référence sur l'objet de type Etudiant correspondant et vaut l'adresse mémoire E515 :FB80.

Un objet peut posséder plusieurs références : par exemple, si on fait :

```
Mohamed = Farid ;
```

Cela signifie qu'ils référencent tous les deux le même objet, la même zone mémoire. Dans ce cas, l'objet Mohamed est donc perdu car il n'est plus référencé. Les références Mohamed et Farid désignent maintenant le même objet.

Voilà une représentation préliminaire de la classe Etudiant en java (:

```
class Etudiant {
int  matricule ;
String nom;
String prenoms;
passerExamen()
}
```

Chaque attribut est décrit au moins par un nom (unique dans une même classe) et par un type de données.

3.2. Attribut

Un attribut est considéré comme une caractéristique spécifique à chaque instance d'une classe. C'est pourquoi, on parle d'attribut d'instance.

Par exemple, dans la classe précédemment définie, le matricule représente une variable ou attribut propre à chaque instance de la classe "Etudiant". Et les données renfermées ne sont pas partagées.

D'autre part, un attribut peut contenir une donnée unique, constante ou partagée entre les instances d'une classe : on parle d'attribut de classe (on verra plus loin avec "static").

La déclaration d'un attribut est essentiellement composée de sa visibilité (voir les modificateurs d'accès), le type de la donnée contenue et du nom qu'il porte.

3.3. Méthodes

Les méthodes (ressemblent aux fonctions) permettent d'effectuer des traitements. Une méthode se délimite comme une classe, c'est à dire par deux accolades {}. Il est obligatoire d'y ajouter des parenthèses, même si elles ne contiennent rien. Ce qu'on peut placer à l'intérieur des parenthèses s'appellent les paramètres.

Une opération, ou méthode, est décrite au moins par un nom, par un ensemble d'arguments nécessaires à son invocation et par un type de retour. Chaque argument est décrit par un nom et un type de données.

Il existe principalement deux types de méthodes :

- Une méthode qui ne renvoie pas de résultat (semblable aux procédures du langage pascal), pour ce type de méthode, le nom doit être précédé par le mot clé "void".

Voilà une méthode qui affiche le message "Bonjour tout le monde".

```
Void affiche(){
System.out.println("Bonjour tout le monde") ;
}
```

- Une méthode qui renvoie un résultat (semblable aux fonctions du langage pascal), pour ce type de méthode, le nom doit être précédé par le type de donnée du résultat et le corps

doit contenir le mot clé "return". Notons, qu'il peut exister autant de return dans une même méthode, cependant au moment de l'exécution, un seul return est active.

Voilà une méthode qui calcule la somme de deux entiers et renvoie un entier.

```
Int calcul() {
int a=2;
int b=3;
return a+b;
}
```

Lorsque votre méthode doit retourner un type de donnée, vous devez remplacer void par le type de donnée que vous souhaitez. Lorsque votre méthode doit retourner une donnée, on utilise le mot clé return.

Attention, ce mot clé provoque l'arrêt de la fonction, c'est à dire que la donnée sera retournée et l'éventuel code se trouvant après le mot return ne sera pas exécuté. On peut donc utiliser le mot clé return pour interrompre une fonction comme ceci : return ; (ceci est valable si votre fonction ne renvoie rien, sinon vous devez retourner le type de données approprie, par exemple 0 pour un type de données int).

Exemple d'une méthode acceptant un paramètre et retournant un entier :

```
public class Test {
public int MaMethode(int variable) {
System.out.println ("Le nombre que vous avez passe en
paramètre vaut : " + variable) ;
return variable + 50;
}
public static void main(String args[]) {
 Test t = new Test();
t.MaMethode(6);
}
}
```

3.4. Package

La programmation Java consiste à créer des classes. Or, étant donné qu'un programme est généralement développé par une équipe de programmeurs, c'est-à-dire plusieurs personnes, le fait de concaténer des classes dans un fichier est assez lourd et loin d'être satisfaisant. C'est pour cette raison que Java propose l'utilisation de packages (comparables aux bibliothèques du langage C++/C).

Un package est une unité (un fichier) regroupant des classes. Pour créer un package, il suffit de commencer le fichier source contenant les classes à regrouper par l'instruction package suivi du nom que l'on désire donner au package. Des lors, toutes les classes contenues dans le fichier feront partie de ce package.

Par exemple, la classe maclasse, appartient au package monpak :

```
package monpak class maclasse ...
```

Pour pouvoir utiliser une classe, par exemple, utiliser une de ces méthodes, il faut utiliser la clause "import", comme première instruction après le nom de package, comme l'exemple ci-dessus :

```
import monpak.maclasse ;
```

Pour appeler toutes les classes du package, il suffit de remplacer le nom de la classe par "*", par exemple :

```
import monpak.* ;
```

le langage java est livré avec un grand ensemble de classes. L'ensemble des packages livrés par Java forme ce qu'on appelle l'API (Application Programming Interface). On les regroupe en packages nommés java.* (Vous pouvez consulter la documentation du JDK disponible pour connaître le contenu de ces classes).

On vous donne quelques exemples de packages :

- Le package java.lang est chargé automatiquement, ses classes sont donc toujours utilisables. On y trouve, entre autres :
  - ✓ La classe Object dont dérivent toutes les autres classes.
  - ✓ Les classes représentant les types numériques de bases : Boolean, Byte, Double, Float, Integer, Long.
  - ✓ La classe Math qui fournit des méthodes de calcul des fonctions usuelles en mathématiques.
  - ✓ Les classes Character, String et StringBuffer pour la gestion des caractères et des chaînes de caractères.
  - ✓ La classe System que nous utilisons pour afficher du texte sur la console DOS.
- Le paquetage java.util définit un certain nombre de classes utiles et est un complément de java.lang.
- Le paquetage java.awt (awt = Abstract Window Toolkit) contient des classes pour fabriquer des interfaces graphiques.
- Le paquetage java.applet est utile pour faire des applets.
- Le paquetage java.io contient les classes nécessaires aux entrées-sorties.

Mis à part le package java.lang, tous les autres packages doivent être déclarés (mot clé import) pour pouvoir être utilisés.

L'utilisation d'une classe d'un package spécifique (par exemple java.applet) nécessite d'importer le package entier dans votre programme ou alors de préciser juste la classe à importer.

Ceci se fait en insérant tout au début du programme et avant la définition de la classe :

```
import <suivi du nom du paquetage à importer>
```

L'instruction import peut donc avoir deux formes :

- import nomDePackage.nomDeClasse ; la classe nomDeClasse peut ensuite être désignée par son simple nom,
- import nomDePackage.* ; toutes les classes du package nomDePackage peuvent ensuite être désignée par leur simple nom.

L'instruction import java.lang.*; est implicite dans tout programme JAVA, elle rend directement accessible les classes de base JAVA.

Si deux packages importés contiennent des classes de nom identique, il est nécessaire d'utiliser un nom complet pour désigner chacune de ces deux classes

```
import java.applet.*;
import java.awt.*;
si on souhaite juste une classe
import java.awt.Graphics;
import java.util.Date ;
```

3.5. Constructeur

Lorsque l'on définit un objet d'une classe, il est souvent utile de pouvoir initialiser cet objet. Avec la définition de notre classe Date, il est évidemment possible, avec la méthode affectée, d'affecter les champs jour, mois et année. Mais cette façon de faire, n'est pas la plus agréable. Une meilleure façon de faire consiste à définir une méthode spécifique d'initialisation des champs qui sera automatiquement appelée lors de la création d'un objet. Cette fonction s'appelle constructeur.

Une méthode (ou fonction), en POO, est une fonctionnalité rattachée à un concept d'entité du monde réel ou simplement une routine définie dans une classe utilitaire. Littéralement, un constructeur est une méthode mais il faut préciser que :

- Il prend obligatoirement le nom de la classe dans laquelle il est défini.
- Il n'a pas "de type de retour" et ne retourne" aucune valeur" et il n'est pas précédé par le mot clé "void".

- Il existe toujours un constructeur dans une classe, celui par défaut (vide) en cas d'absence de définition explicite. Ils sont appelés à l'instanciation d'un objet avec le mot-clé " new".

Par exemple, la classe Etudiant :

```
Etudiant Mohamed = new Etudiant ()
```

Il existe deux types de constructeur en Java

- **Constructeur sans argument** : Un constructeur sans paramètre est appelé constructeur par défaut. Si nous ne définissons pas de constructeur dans une classe, le compilateur crée un constructeur par défaut (sans argument) pour la classe. Et si nous écrivons un constructeur avec des arguments ou sans arguments, le compilateur ne crée pas de constructeur par défaut. Le constructeur par défaut fournit les valeurs par défaut à l'objet, telles que 0, null, etc., en fonction du type.

```
public class Personne {
private String nom;
private int age;
//constructeur par défaut
public Personne() {
System.out.println("Je suis le constructeur");
 }
 public void affiche() {
  System.out.println("nom : " + nom);
  System.out.println("age : " + age);
  }
   // méthode principale (main)
     public static void main(String args[]) {
         Personne p = new Personne();
         p.affiche();
     }
}
```

En sortie, nous aurons :

```
Je suis le constructeur
Nom : null
Age : 0
```

- Constructeur **paramétré** : Un constructeur qui a des paramètres est appelé constructeur paramétré. Si nous voulons initialiser les champs de la classe avec vos propres valeurs, utilisez un constructeur paramétré.

```
public class Personne {
    private String nom;
    private int age;
     // constructeur paramétré
    public Personne(String nom, int age) {
         System.out.println("Je suis le constructeur");
         this.nom = nom;
         this.age = age;
```

```
    }
    public void affiche() {
        System.out.println("nom : " + nom);
        System.out.println("age : " + age);
    }
    // méthode principale (main)
    public static void main(String args[]) {
        Personne p = new Personne("Bacha", 16);
        p.affiche();
    }
}
```

En sortie, nous aurons :

```
Je suis le constructeur
Nom : Bacha
Age : 16
```

3.6. Destructeur

Le destructeur est une méthode particulière qui est appelée automatiquement à chaque destruction d'objet. Il permet d'effectuer certaines actions de nettoyage (comme la récupération de mémoire dynamique, la fermeture de fichiers...) avant de perdre tout lien vers l'objet.

La gestion de la destruction des objets est laissée à la charge de l'interprète du langage Java. Les objets qui deviennent inutiles, parce qu'ils ne sont plus utilisés, sont automatiquement détruits par le support d'exécution. Toutefois il est possible de définir dans chaque classe la méthode finalize() `(public void finalize() {...})`, appelée immédiatement avant la destruction de l'objet, afin d'exécuter des actions d'achèvement particulières.

Une classe ne peut disposer que d'un seul destructeur.

3.7. Notion de visibilité

Selon leur niveau de visibilité, les classes, les interfaces, les attributs et les méthodes peuvent être accessibles ou non depuis des classes du même paquetage ou depuis des classes d'autres paquetages. Si on tente de compiler une classe invoquant une classe, une interface ou un champ non accessible, il y a un message d'erreur pour indiquer que l'entité en question n'est pas accessible.

Les classes, attributs et méthodes peuvent être visibles ou non à l'intérieur d'autres paquetages ou d'autres classes.

Pour indiquer les degrés de visibilité, on utilise des modificateurs de visibilité. Les différents modificateurs sont indiqués par les mots clé :

- private
- protected
- public

Ces modificateurs sont indiqués devant l'en-tête d'une classe ou d'une méthode ou devant le type d'un attribut. Lorsqu'il n'y a pas de modificateur, on dit que la visibilité est la visibilité par défaut.

Une classe ne peut que :

- avoir la visibilité par défaut : elle n'est visible alors que de son propre paquetage.
- se voir attribué le modificateur public : elle est alors visible de partout.

Un champ (attribut ou méthode) peut avoir les quatre degrés de visibilité.

On peut écrire par exemple :

```
private int nbDonnees;
public void methode() {}
```

Si un champ d'une classe A :

- est private, il est accessible uniquement depuis sa propre classe ;
- a la visibilité paquetage (par défaut), il est accessible de partout dans le paquetage de A mais de nulle part ailleurs ;
- est protected, il est accessible de partout dans le paquetage de A et, si A est publique, grosso-modo dans les classes héritant de A dans d'autres paquetages ;
- est public, il est accessible de partout dans le paquetage de A et, si A est publique, de partout ailleurs.

Ci-dessus, les niveaux de visibilité sont rangés par visibilité croissante.

Rappelons une règle : une méthode d'instance ne peut pas être redéfinie en diminuant son niveau de visibilité.

Les constantes et les prototypes de méthodes figurant dans une interface sont automatiquement publics et donc visibles de partout si l'interface est publique (on verra les interfaces dans le chapitre 5).

Voilà un exemple qui illustre la visibilité :

```
package p1;
class C1 {
  public int a;
  protected int b;
  int c; // friendly
  private int d;
}
class C2 extends C1 {
    ...
}
class C3 {
    ...
}
```

```
package p2;
class C4 extends C1 {
   ...
}
class C5 {
   ...
}
```

	a	b	c	d
Accessible de C2	oui	oui	oui	-
Accessible de C3	oui	oui	oui	-
Accessible de C4	oui	oui	-	-
Accessible de C5	oui	-	-	-

3.8. Principe d'encapsulation

Le principe d'encapsulation dit qu'un objet ne doit pas exposer sa représentation interne au monde extérieur. Les données stockées par l'objet doivent être cachées de l'utilisateur de l'objet, et toute interaction avec l'objet doit se faire via des méthodes.

Prenons par exemple une classe Hour qui permet de représenter une heure :

```
public class Hour
{
	public int hour, minute;

	public Hour (int h, int m)
	{
		hour = h;
		minute = m;
	}
}
```

On définit deux variables d'instance pour stocker l'heure et les minutes, déclarées public. On définit ensuite une classe Meeting qui représente une réunion, identifiée par un sujet, une heure de début et heure de fin. On peut connaitre la durée d'une réunion grâce à une méthode getDuration.

```
public class Meeting
{
	public String object;
	public Hour start, end;

	public Meeting (String str, Hour s, Hour e)
	{
		object = str;
		start = s;
		end = e;
	}

	public int getDuration()
	{
		return (end.hour * 60 + end.minute) - (start.hour * 60 +
start.minute);
	}
}
```

Pour calculer la durée de la réunion, on soustrait donc l'heure du début de l'heure de fin, après avoir ramené le tout en minutes.

On suppose bien entendu que les réunions commencent et se terminent la même journée. Remarquez qu'on accède directement aux variables d'instances de la classe Hour, puisqu'elles sont publiques et qu'il n'y a pas d'accesseur.

On a ainsi créé une forte cohésion entre les classes Meeting et Hour; elles sont fortement liées. On ne peut plus changer l'implémentation de la classe Hour indépendamment de la classe Meeting. Imaginons par exemple qu'on change l'implémentation de la classe Hour en ne gardant plus qu'une seule variable d'instance pour stocker l'heure de la réunion sous forme de minutes :

```
public class Hour
{
      public int minutes;

      public Hour (int h, int m)
      {
            minutes = h * 60 + m;
      }
}
```

Si on fait ce changement, la méthode getDuration de la classe Meeting ne fonctionnera plus. Ce qu'il faut faire pour rendre la classe Hour plus indépendante, c'est déclarer toutes les variables d'instance privées. Ensuite, il faut ajouter des méthodes qui permettent d'interagir avec l'objet. Ici, tout ce qu'on souhaite pouvoir faire avec un tel objet, c'est connaitre l'heure et les minutes; on ajoute donc une méthode getHour pour obtenir l'heure et une méthode getMinutes pour les minutes.

```
public class Hour
{
      private int minutes;

      public Hour (int h, int m)
      {
            minutes = h * 60 + m;
      }

      public int getHour()
      {
            return minutes / 60;
      }

      public int getMinutes()
      {
            return minutes % 60;
      }
}
```

Maintenant, il suffit d'utiliser ces méthodes dans la classe Meeting :

```
public int getDuration()
{
      return (end.getHour() * 60 + end.getMinute())
                                    -    (start.getHour()    *    60    +
start.getMinute());
```

}

La classe Meeting dépend maintenant beaucoup moins de l'implémentation de la classe Hour, on peut changer sa représentation interne, et donc également les deux méthodes getHour et getMinutes, sans que cela n'ait un quelconque impact sur le bon fonctionnement de la classe Meeting.

Selon le principe d'encapsulation, une classe apparait donc comme une boite noire. Depuis l'extérieur, on ne doit rien voir sur les détails internes de la classe; on ne voit que les constructeurs et méthodes publics. L'ensemble des méthodes publiques d'une classe est appelé *interface* de la classe. Tout utilisateur ne devrait interagir avec un objet que via les méthodes de son interface.

Le principe d'encapsulation vise donc à bien séparer les fonctionnalités publiques offertes par un objet de leur implémentation. On sépare le « Quelles fonctionnalités sont disponibles ? » du « Comment ces fonctionnalités sont implémentées ? ». Pour pouvoir utiliser un objet, il est inutile de savoir comment l'objet gère, de manière interne, ses attributs ; il suffit de connaitre les opérations disponibles et à quoi elles servent

3.9. Getters et Setters (Accesseurs et mutateurs)

Les accesseurs vont être le moyen qui va nous permettre d'agir sur nos variables d'objet en lecture, alors que les mutateurs nous permettrons d'agir sur nos variables d'objet en écriture.

On peut donc afficher les variables de nos objets grâce aux accesseurs et les modifier grâce aux mutateurs.

```
// Nos accesseurs => envoient l'information vers le main
public int getIdentifiant()
{
    return identifiant;
}
public String getNom()
{
    return nom;
}
public String getMarque()
{
    return marque;
}
public Date getDateCreation()
{
    return dateCreation;
}
```

```java
// Nos mutateurs => récupèrent les informations depuis le main
public void setIdentifiant(int pIdentifiant)
{
    identifiant = pIdentifiant;
}
public void setNom(String pNom)
{
    nom = pNom;
}
public void setMarque(String pMarque)
{
    marque = pMarque;
}
public void setDateCreation(Date pDate)
{
    marque = pDate;
}
```

Nos méthodes accesseurs et mutateurs doivent bien évidemment être publiques afin d'être accessibles depuis les autres classes (sans quoi il n'y aurait pas d'intérêt).

On peut retenir qu'un mutateur sera toujours de type void**,** car il ne renvoie rien**.** Alors qu'un accesseur renvoie une variable et sera donc du type de celle-ci**.**

3.10. Surcharge de méthodes

La surcharge (overloading en anglais) ou surdéfinition est la possibilité de définir plusieurs méthodes de même nom dans la même classe.

La surcharge est également possible pour le constructeur d'une classe, autrement dit une classe peut avoir plusieurs constructeurs de même nom.

Voici un exemple de surcharge de méthode :

```
public  class Test {
public Test(){
MaMethode();
MaMethode(50);
public void MaMethode(int variable) {
System.out.println("Le nombre que vous avez passe en paramètre vaut
: " + variable ) ;
}
public void MaMethode(){
System.out.println ("Vous avez appel  la méthode sans paramètre ") ;
}
}
```

La surcharge permet à différentes méthodes d'avoir le même nom, mais des signatures différentes où la signature peut différer en fonction du nombre de paramètres d'entrée, du type de paramètres d'entrée ou des deux.

```
public class Test {

    // Cette méthode prend deux paramètres int
    public int somme(int x, int y) {
        return (x + y);
    }

    // Cette méthode prend trois paramètres int
    public int somme(int x, int y, int z) {
        return (x + y + z);
    }

    // Cette méthode prend deux paramètres double
    public double somme(double x, double y) {
        return (x + y);
    }

    public static void main(String args[]) {
        Test t = new Test();
        System.out.println(t.somme(10, 20));
        System.out.println(t.somme(10, 20, 30));
        System.out.println(t.somme(10.5, 20.5));
    }
}
```

En sortie, nous aurons :

```
30
60
31.0
```

En termes de priorité, le compilateur prend les mesures suivantes :

1. Conversion de type mais vers un type plus élevé (en termes de plage) dans la même catégorie.
2. Conversion de type dans la catégorie immédiatement supérieure (supposez qu'il n'y ait pas de type de données long disponible pour un type de données int, alors il recherchera le type de données float).

3.11. Eléments statiques

static est un modificateur qui s'applique aux éléments suivants :

- Des variables
- Des méthodes
- Des classes

Pour créer un membre statique (bloc, variable, méthode), utiliser le mot-clé static. Lorsqu'un membre est déclaré statique, il est possible d'y accéder avant la création des objets de sa classe et sans référence à aucun objet.

3.11.1 Champ statique

Lorsqu'une variable est déclarée statique, une seule copie de la variable est créée et partagée entre tous les objets au niveau de la classe. Les variables statiques sont des variables globales. Toutes les instances de la classe partagent la même variable statique.

```
public class Foo
{

static int X = 3; // variable statique
int y = 2; // variable non statique

public static void main(String[] args) {

System.out.println("X+1 = " + (X + 1));

Foo foo = new Foo();
System.out.println("y+1 = " + (foo.y + 1));
}
}
```

La sortie

```
X+1 = 4
y+1 = 3
```

Ici, dans le programme ci-dessus, la variable nommée X est déclarée statique alors que y est une variable non statique. Désormais, dans la méthode main, vous devez créer une instance de la classe pour accéder à la variable d'instance y. Toutefois, il est possible d'accéder directement à la variable X.

3.11.2 Méthode statique

Une méthode statique est une méthode qui peut être appelée même sans avoir instancié la classe. Une méthode statique ne peut accéder qu'à des attributs et méthodes statiques.

```
public class Essai {
public  static  String chaine = "bonjour";
public  static  void MaMethodeStatique() {
System.out. println ("Appel de la méthode statique : " + chaine);
}
}
```

Vous pouvez sans avoir instanciéla classe accéder à la méthode statique en tapant ceci :

Essai.MaMethodeStatique() ; n'importe où dans votre code.

3.11.3 Classe membre statique

Une classe membre peut être déclarée statique. Dans ce cas, on notera qu'il est possible à la machine Java de la charger, sans charger la classe dans laquelle cette classe est déclarée.

3.12. Eléments final

Le mot-clé final en Java est un modificateur utilisé pour empêcher la modification d'un code ou d'une valeur. Il est possible d'utiliser ce mot-clé dans 3 contextes :

- Mot-clé final en tant que modificateur de **variable**
- Mot clé final en tant que modificateur de **méthode**
- Mot-clé final en tant que modificateur de **classe**

3.12.1 Variable final

Chaque contexte a son propre but. Le mot-clé final, lorsqu'il est utilisé avec une variable, il est spécifiquement destiné à empêcher l'utilisateur de modifier la valeur. Une variable déclarée final, une fois initialisée, sa valeur ne peut être changée.

Si la variable final vient d'être déclarée mais non initialisée, il est permis d'attribuer une valeur à une variable final une fois dans le code. Toute modification de la valeur d'une variable final entraînera une erreur de compilation.

Par exemple, le code ci-dessous affichera une erreur lors de la compilation si vous tentez de le compiler.

```
public class FinalStr {

final String str = "Hello World!";

public FinalStr() {
str = "Welcome"; //la variable str ne peut pas être modifiée
}

}
```

3.12.2 Méthode final

Une méthode déclarée final ne peut être redéfinie (on verra la notion de redéfinition au chapitre 4).

Ce qui signifie que même une sous-classe peut appeler la méthode final de la classe parente sans aucun problème, mais elle ne peut y redéfinir.

Par exemple, le code ci-dessous affichera une erreur lors de la compilation si vous tentez de le compiler.

```
class Bar{
final void display(){
System.out.println("Méthode de la classe Bar");
}
}

class Foo extends Bar{
void display(){
System.out.println("Méthode de la classe Foo");
}

public static void main(String args[]){
Foo obj= new Foo();
obj.display();
}
}
```

3.12.3 Classe final

Le mot clé final, lorsqu'il est utilisé par une classe en tant que modificateur, il empêche la classe d'être héritée par une autre classe. Par exemple, si vous essayez de compiler le code ci-dessous, il génère une erreur de compilation car la classe « Bar » est une classe final qui ne peut être héritée.

```
final class Bar{
}

class Foo extends Bar{
void display(){
System.out.println("Hello World!");
}
public static void main(String args[]){
Foo obj= new Foo();
obj.display();
}
}
```

3.13. Exercices

Exercice1

Créer une classe Java nommée Compte qui représente un compte bancaire de visibilité public, ayant pour attributs public Double solde
Créer un constructeur ayant comme paramètre solde.
Créer maintenant les méthodes suivantes :
deposer() de type void qui gère les versements
retirer() de type void qui gère les retraits.
méthode afficher() de type void permettant d'afficher le solde

Donner le code complet de la classe Compte.

Créer une classe TestCompte permettant de tester le compte en effectuant un versement et puis un retrait

Corrigé

```
public class Compte {
	Double solde;
	public Compte( Double s){
		this.solde=s;
	}
	public  void deposer(double d){
		this.solde+=d;
	}
	public void retirer(double r){
		this.solde-=r;
	}
public void afficher(){
System.out.println("Votre solde est  " + this.solde +" Euro "+" sauf
erreur ou omission");
}
}
public class TestCompte {

	public static void main(String[] args) {

Compte monCompte=new Compte(5000.0);
monCompte.deposer(3000);
monCompte.retirer(2000);
monCompte.afficher();
	}
}
```

Exercice2

Créer une classe Voiture de visibilité public, ayant pour attributs : String marque, Double prix, de visibilité public.

Créer un constructeur sans paramètres (par défaut)

Créer les getters et setters

Créer une méthode void afficher( ) permettant d'afficher les résultats

Donner le code final de la classe.

Créer une classe Exécution de visibilité public permettant d'exécuter les résultats.

Corrigé

```
public class Voiture {
	public String marque;
	public Double prix;

public Voiture() {

	}

/* Création des setters */
public  void setMarque(String mq){
	this.marque=mq;
}
```

```
public  void setPrix(Double  pr){
   this.prix=pr;
}

/* Création des getters */
public String getMarque(){
       return marque;
}
public Double getPrix(){
       return prix;
}

       public  void afficher() {
       System.out.println("La marque de ma voiture est : "+
this.marque );
       System.out.println("La prix de ma voiture est : "+ this.prix
+ " Euro" );
       }

}
```

Création de la classe Exécution

```
public class Exécution {

       public static void main(String[] args) {
              Voiture maVoiture=new Voiture();
              maVoiture.setMarque("Renault");
              maVoiture.setPrix(17500.0);
              maVoiture.afficher();
       }
}
```

Exercice3

Créer un package Java sous Netbeans nommé geometry.

Au sein du package geometry créer une classe Java nommée Point ayant pour attribut Double Abscisse et double ordonnee.

Au sein du même package créer une classe Cercle doté d'un attribut centre du type Point. Et des méthodes suivantes :

- périmètre() du type Double permettant de calculer le périmètre du cercle.
- surface() du type Double permettant de calculer la surface du cercle.
- testAppartenance() du type void permettant de tester si un point appartient au cercle ou non.
- afficher() du type void permettant d'afficher les résultats

Créer une classe TestCercle contenant une méthode static void main() permettant de tester les résultats.

Correction

Création de la classe Point :

```
package geometry;

public class Point {
```

```
	public Double abscisse;
	public double ordonnee;
	public Point(Double ab,Double or){
		this.abscisse=ab;
		this.ordonnee=or;
	}
}
```

Création de la classe Cercle

```
package geometry;
public class Cercle{
	public Point centre;
	public Double rayon;
	public Cercle(Point point,Double r){
	this.centre=point;
	this.rayon=r;
	}
	public Double périmètre(){
		Double pi=Math.PI;
		return 2*pi*rayon;
	}
	public Double surface(){
		Double pi=Math.PI;
		return pi*rayon*rayon;
	}
	public void testAppartenance(Point q){
		Double dx=q.abscisse-this.centre.abscisse;
		Double dy=q.ordonnee-this.centre.ordonnee;
		Double distance=Math.sqrt(dx*dx+dy*dy);
	if(distance.doubleValue()==this.rayon.doubleValue()){
System.out.println("Le point choisi apprtient au
cercle");
	}
	else{
System.out.println("Le point choisi n'apprtient pas au
cercle");
	}
	}
void afficherCercle(){
System.out.println("Le rayon du cercle est "+ this.rayon);
System.out.println("Le centre du cercle est le point ayant pour
abscisse = "+ this.centre.abscisse + " ayant pour ordonnée = "+
this.centre.ordonnee );
	}
}
```

Création de la classe TestCercle

```
package geometry;
public class TestCercle {
	public static void main(String[] args) {
		Point centre=new Point(0.0,0.0);
		Cercle monCercle=new Cercle(centre,3.0);
		Point M=new Point(3.0,0.0);
		monCercle.afficherCercle();
		monCercle.testAppartenance(M);
```

```
        }
}
```

Ce qui affiche après exécution sur Netbeans :

```
Le rayon du cercle est 3.0
Le centre du cercle est le point ayant pour abscisse = 0.0
ayant pour ordonnée = 0.0
Le point choisi appartient au cercle
```

Chapitre

4

Héritage et Polymorphisme

A l'issue de ce chapitre, l'apprenant sera capable de Comprendre les principes de l'héritage et polymorphisme et savoir appliquer ces concepts dans des programmes.

4.1. Héritage

Dans certaines applications, les classes utilisées ont en commun certaines variables, méthodes de traitement ou même des signatures de méthode. Avec un langage de programmation orient objet, on peut définir une classe à diffèrent niveaux d'abstraction permettant ainsi de factoriser certains attributs communs a plusieurs classes. Une classe générale définit alors un ensemble d'attributs qui sont partages par d'autres classes, dont on dira qu'elles héritent de cette classe générale.

Comme tous les langages orientés objet, JAVA supporte la généralisation spécialisation, le plus communément appelée "héritage" dans les langages de programmation. Pour simplifier la mise en œuvre, JAVA ne permet que l'héritage simple, c'est-à-dire qu'une classe dérive d'une classe et d'une seule. En outre, JAVA introduit le concept d'interface ; une classe peut

"implémenter" plusieurs interfaces. Nous présentons avec plus de détail le concept d'héritage dans ce chapitre.

Il faut retenir que toutes les classes dérivent d'une classe racine appelée "java.lang.Object". Cette classe de nit des comportements stéréotypes dont nous reparlerons dans le cours

Par exemple, les classes Carre et Rectangle peuvent partager une méthode surface() renvoyant le résultat du calcul de la surface de la figure. Plutôt que d'écrire deux fois cette méthode, on peut définir une relation d'héritage entre les classes Carre et Rectangle. Dans ce cas, seule la classe Rectangle contient le code de la méthode surface() mais celle-ci est également utilisable sur les objets de la classe Carre si elle hérite de Rectangle.

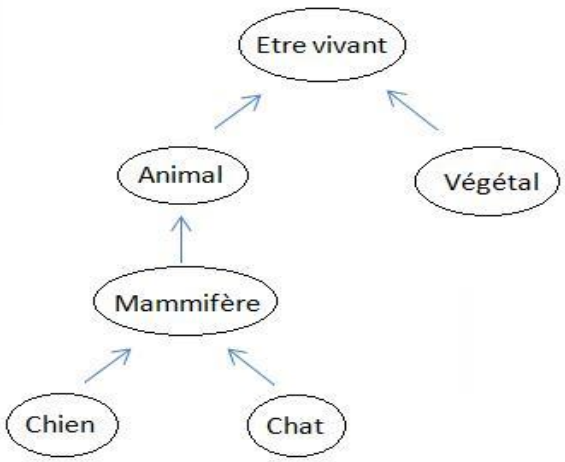
4.1.1 Mécanisme d'héritage

L'idée principale de l'héritage est d'organiser les classes de manière hiérarchique. La relation d'héritage est unidirectionnelle et, si une classe B hérite d'une classe A, on dira que B est une sous-classe de A. Cette notion de sous-classe signifie que la classe B est un cas particulier de la classe A et donc que les objets instanciant la classe B instancient également la classe A.

L'héritage est la transmission de caractéristiques à ses descendants ? L'héritage correspond à la relation "est-un" ? Un Etudiant est une Personne ? Un Etudiant est une spécialisation de Personne ? Une PERSONNE est une généralisation d'un Etudiant

- vision descendante => la possibilité de reprendre intégralement tout ce qui a déjà et fait et de pouvoir l'enrichir.
- vision ascendante => la possibilité de regrouper en un seul endroit ce qui est commun a plusieurs.

Prenons comme exemple des classes Carre, Rectangle et Cercle. La figure suivante propose une organisation hiérarchique de ces classes telle que chien hérite de Mammifère qui hérite de Animal qui hérite, ainsi de Etre vivant.



- vers le haut (en analyse O.O.) => on regroupe dans une classe ce qui est commun a plusieurs classes. Dans la classe Etre vivant, on regroupe les caractéristiques communes aux autres classes.

- vers le bas (lors de la réutilisabilité) => la classe de base étant définie, on peut la reprendre intégralement pour construire la classe dérivée. La classe Etre vivant étant définie, on peut la reprendre intégralement, pour construire la classe Animal.

Une classe dérivée modélise un cas particulier de la classe de base, et est enrichie d'informations supplémentaires.

La classe dérivée possède les propriétés suivantes :

- contient les données membres de la classe de base,
- peut en posséder de nouvelles,
- possède (a priori) les méthodes de sa classe de base,
- peut redéfinir (masquer) certaines méthodes,
- peut posséder de nouvelles méthodes. Pour signifier qu'une classe fille hérite d'une classe mère, on utilise le mot clé extends : class Fille extends mère.

Notons que l'héritage en java est simple (une seule mère).

4.1.2 Empêcher l'héritage

Il est possible d'empêcher qu'une classe soit surchargée par une autre classe, pour cela, il suffit d'utiliser le mot-clé " final", pris en compte à la compilation.

L'exemple suivant illustre l'utilisation de final :

```
Public final class A f // ne peut être étendue
public void mymethod() f
System.out.println ("Je suis dans A") ;
```

4.1.3 Héritage et constructeur

Un objet d'une classe dérivée est un objet de la classe parente plus une partie qui correspond à la classe dérivée. Il est donc nécessaire d'initialiser la partie provenant de la classe parente lorsque l'objet est créé.

Le constructeur de la classe dérivée doit donc faire appel au constructeur de la classe parente pour réaliser cette initialisation. Le plus souvent un constructeur de classe dérivée reçoit un ensemble de paramètres pour initialiser les attributs de la classe parente. Il utilise ces paramètres pour faire appel au constructeur de la classe parente.

Remarques :

- L'appel au constructeur d'une classe supérieure doit toujours se situer dans un constructeur et toujours en tant que première instruction ;
- Si aucun appel à un constructeur d'une classe supérieure n'est fait, le constructeur fait appel implicitement à un constructeur vide de la classe

supérieure (comme si la ligne super() était présente). Si aucun constructeur vide n'est accessible dans la classe supérieure, une erreur se produit lors de la compilation.

4.2. Redéfinition des méthodes

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques).

Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.

Lors de la redéfinition d'une méthode, on ne doit pas restreindre sa visibilité.

4.3. Transtypage

Il existe deux types de transtypage :

- Le transtypage ascendant qui sert a un objet à être vu comme du type de sa mère. Ce transtypage est fait de façon implicite et n'est nécessaire que dans le cas où la classe courante possède une propriété ayant le même nom que sa classe mère et que l'on cherche a accéder à la propriété mère.

- Le transtypage descendant- le plus courant - qui sert a forcer une classe de type mère à être vue comme un type fille. Ce transtypage n'est valide que si l'objet de type mère était de type fille à son instanciation.

Voilà quelques règles à retenir :

- Une classe ne peut hériter que d'une autre classe (héritage simple, l'héritage multiple n'est pas autorisé en Java).
- Une classe hérite d'une autre par l'utilisation du mot réservé "extends".
- Une classe pour laquelle aucune spécialisation n'est explicitée spécialise implicitement la classe "java.lang.Object" (qui est la classe mère par défaut).
- Le mot réservé "final" utilise devant le mot clé "class" interdit toute spécialisation (héritage) de la classe sur laquelle il est utilise

4.4. Polymorphisme

Dans le paragraphe précédent, nous avons parlé de superclasses et de sous-classes. Si une classe hérite d'une méthode de sa superclasse, alors il y a une chance de reformuler la méthode à condition qu'il ne soit pas marqué final.

L'avantage de la substitution est la capacité de définir un comportement spécifique au type de sous-classe, ce qui signifie qu'une sous-classe peut implémenter une méthode de classe parent en fonction de son besoin et ses exigences.

```
public class Animal {
          public void seDeplacer() {
             System.out.println("Un animal peut se déplacer");
          }
       }

       class Chien extends Animal {
          public void seDeplacer() {
             System.out.println("Un chien peut courir");
          }
       }

       class Oiseau  extends Animal {
                 public void seDeplacer() {
                    System.out.println("Un oiseau peut voler");
                 }
              }
```

Et pour exécuter le code de cette classe nous aurons besoin d'une autre classe doté d'une méthode static void main().

```
public class ExecutionAnimal {

                 public static void main(String args[]) {
                    Animal a = new Animal();
                    Animal b = new Chien();
                    Animal c= new Oiseau();

                    a.seDeplacer();
                    b.seDeplacer();
                    c.seDeplacer();
```

Ce qui affiche après exécution :

```
Un animal peut se déplacer
Un chien peut courir
Un oiseau peut voler
```

4.5. Exercices

Exercice1

La classe Voiture dérive de la classe Véhicule. Soit le programme suivant se trouvant dans une classe quelconque : (les notations ne correspondent pas à celles d'un véritable langage de programmation).

```
class Vehicule
rouler ()f
seGarer()f
class  Voiture extends Vehiculef
seGarer()f
```

```
klaxonner()f
rouler ()f
```

Soit une106, une instance de Voiture et monVehicule, une instance de Vehicule.

1. Peux-t-on faire une106.rouler() ? Pourquoi ?
2. Si l'on fait une106.seGarer(), est-ce la méthode de la classe Vehicule ou de la classe Voiture qui est appelée ?
3. Peux-t-on faire monVehicule.klaxonner() ? Pourquoi ?

Correction

1. Oui : car une106 est un objet de type Voiture qui dérive de Vehicule.
2. C'est la méthode de Voiture car par défaut, la méthode exécutée est celle de la classe courante.
3. Non : car la classe Vehicule ne possède pas de méthode klaxonner().

Exercice2

Qu'affiche le programme suivant ?

```
class A {
private  int val=0;
public  static  void  afficher(int  i) {
i++;
System.out. println  ( i ) ;
public  static  void  afficher  (A i) f

i . val++;
System.out. println  ( i . val ) ;
public  static  void main (String []  args) f
A a = new A ();
A b = new A();
afficher(a.val) ;
afficher(a.val) ;
afficher(a) ;
afficher(b) ;
afficher(b);
if (a==b) System.out.println("Egales") ;
else System.out. println ("Différentes ") ;
```

Exercice 3

Les exemples suivants sont-ils corrects ? Justifier

Exemple1:

```
class A {
public void f (){ System.out. println ("Bonjour.") ; }
class B extends A {
private void f (){ System.out. println ("Bonjour les amis. ") ; }
```

Exemple2:

```
class A {
```

```
public  int f( int a) { return a++;
class B extends A {
public boolean f( int a) { return (a==0);}
```

Exemple3 :

```
class A {
public  int f( int a) { return a++; }
class B extends A {
public  int f( int a, int b) { return a+b;}
class  test {
B obj = new B();
int x = obj.f(3) ;
int y = obj.f(3,3) ;
}
}
```

Exemple4:

```
class A {
public  int f( int a) { return a++;
class B extends A {
public  int f( int a, int b) { return a+b;
class  test {
A obj = new B();
int x = obj.f(3) ;
int y = obj.f(3,3) ;
}
}
```

Correction

- Exemple 1 : Non, on ne peut redéfinir une méthode en restreignant sa visibilité.
- Exemple 2 : Non, on ne peut redéfinir une méthode et changer le type du résultat.
- Exemple 3 : Oui, L'appel f(3) utilise la méthode de la super classe.
- Exemple 4 : Non, le compilateur va rejeter le programme car obj a et déclaré de la classe A pour le quel f est defini sur un seul argument.

Exercice 4

Qu'affiche le programme suivant ?

```
class A {
public String f (){ return ("A");}
}
class B extends A {
public String f (){ return ("B");}
}
class test {
public static  void main (String  []  args) {
A a1 = new A();
A a2 = new B();
B b = new B();
System.out.println(a1.f());
System.out.println(a2.f()) ;
System.out.println(b.f());
```

```
}
}
```

Correction

Cet exercice illustre le principe de la liaison dynamique, le choix de la méthode est fait dynamiquement pendant l'exécution.

```
A

B

B
```

Chapitre

5

Classes abstraites et Interfaces

A l'issue de ce chapitre, l'apprenant sera capable de Comprendre et savoir créer et manipuler des classes abstraites et des interfaces

.

5.1. Classes abstraites

Une classe abstraite est quasiment identique à une classe normale. Elle est identique aux classes que vous avez maintenant l'habitude de coder. Cela dit, elle a tout de même une particularité : on ne peut pas l'instancier.

Pour comprendre, voici un code qui ne compilera pas :

```
public  class Testf
public  static  void main(String []  args)f
A obj = new A(); //Erreur de compilation !
```

Pour bien en comprendre l'utilité, il vous faut un exemple de situation (de programme, en fait) qui le requiert.

Imaginez que vous êtes en train de réaliser un programme qui gère différents types d'animaux. Dans ce programme, vous aurez des loups, des chiens, des chats, des

lions et des tigres. Mais vous n'allez tout de même pas faire toutes vos classes : il va de soi que tous ces animaux ont des points communs, c'est à dire, il existe une relation d'héritage.

Que pouvons-nous définir de commun à tous ces animaux ? Le fait qu'ils aient une couleur, un poids, un cri, une façon de se déplacera, qu'ils mangent et boivent quelque chose.

Nous pouvons donc créer une classe mère que nous appelons "Animal". Avec ce que nous avons de ni de commun, nous pouvons lui définir des attributs et des méthodes.

```
public class Test public static void main(String[] args)
Animal ani = new Animal() ;
((Loup)ani).manger() ; //Que doit-il faire ?
```

C'est là qu'entrent en jeu nos classes abstraites. En fait, ces classes servent à définir une superclasse : par là, vous pouvez comprendre qu'elles servent essentiellement a créer un nouveau type d'objets. Voyons maintenant comment créer une telle classe. Une classe Animal très abstraite.

En fait, il existe une règle pour qu'une classe soit considérée comme abstraite. Elle doit être déclarée avec le mot clé abstract. Voici un exemple illustrant la classe :

```
abstract class Animal
```

Une telle classe peut contenir la même chose qu'une classe normale. Ses enfants pourront utiliser tous ses éléments déclarés (attributs et méthodes déclarés public ou protected).

Cependant, ce type de classe permet de définir des méthodes abstraites qui présentent une particularité : elles n'ont pas de corps.

En voici un exemple :

```
abstract class Animal
abstract void manger() ;  //Une méthode abstraite
```

Vous voyez pourquoi on dit méthode abstraite : difficile de voir ce que cette méthode sait faire.

Il faut que savoir qu'une méthode abstraite ne peut exister que dans une classe abstraite. Si, dans une classe, vous avez une méthode déclarée abstraite, vous devez déclarer cette classe comme étant abstraite.

Voyons à quoi cela peut servir. Vous avez vu les avantages de l'héritage et du polymorphisme. Eh bien nos classes enfants hériteront aussi des méthodes abstraites, mais étant donné que celles-ci n'ont pas de corps, nos classes enfants seront obligées de redéfinir ces méthodes.

Elles présentent donc des méthodes polymorphes, ce qui implique que la covariance des variables pointe à nouveau le bout de son nez.

```
public class Test
public static void main(String args[])
Animal loup = new Loup() ;
Animal chien = new Chien() ;
loup.manger() ;
chien.crier() ;
```

Nous n'avons pas instancié notre classe abstraite. Nous avons instancié un objet Loup que nous avons mis dans un objet de type Animal (il en va de même pour l'instanciation de la classe Chien). Vous devez vous rappeler que l'instance se crée avec le mot clé new.

En aucun cas, le fait de déclarer une variable d'un type de classe donne, ici, Animal, n'est une instanciation ! Ici, nous instancions un Loup et un Chien.

Vous pouvez aussi utiliser une variable de type Object comme référence à un objet Loup, à un objet Chien etc. Vous saviez déjà que ce code fonctionne :

```
public class Test
public static void main(String[] args)
Object obj = new Loup() ;
(Loup)obj).manger() ;
```

En revanche, ceci pose problème :

```
public static void main(String[] args)
Object obj = new Loup() ;
Loup l = obj ; //Problème de référence
```

5.2. Interfaces

Une interface du langage de programmation Java est un type abstrait utilisé pour spécifier un comportement que les classes doivent implémenter. Ils sont similaires aux protocoles. Les interfaces sont déclarées à l'aide du mot-clé interface et ne peuvent contenir que des déclarations de méthode et des déclarations de constante (variable déclarative déclarée à la fois statique et finale).

Toutes les méthodes d'une interface ne contiennent pas d'implémentation (corps de méthodes) de toutes les versions antérieures à Java 11. À partir de Java 8, les méthodes par défaut et les méthodes statiques peuvent avoir une implémentation dans la définition d'interface.

Les interfaces ne peuvent pas être instanciées, mais sont plutôt implémentées.

Une classe qui implémente une interface doit implémenter toutes les méthodes non définies par défaut décrites dans l'interface ou être une classe abstraite.

Les références d'objet en Java peuvent être spécifiées comme étant d'un type d'interface ; dans chaque cas, ils doivent être nuls ou être liés à un objet qui implémente l'interface.

L'un des avantages de l'utilisation des interfaces est qu'elles simulent un héritage multiple. Toutes les classes en Java doivent avoir exactement une classe de base, la seule exception étant java.lang.Object (la classe racine du système de types Java) ; l'héritage multiple de classes n'est pas autorisé.

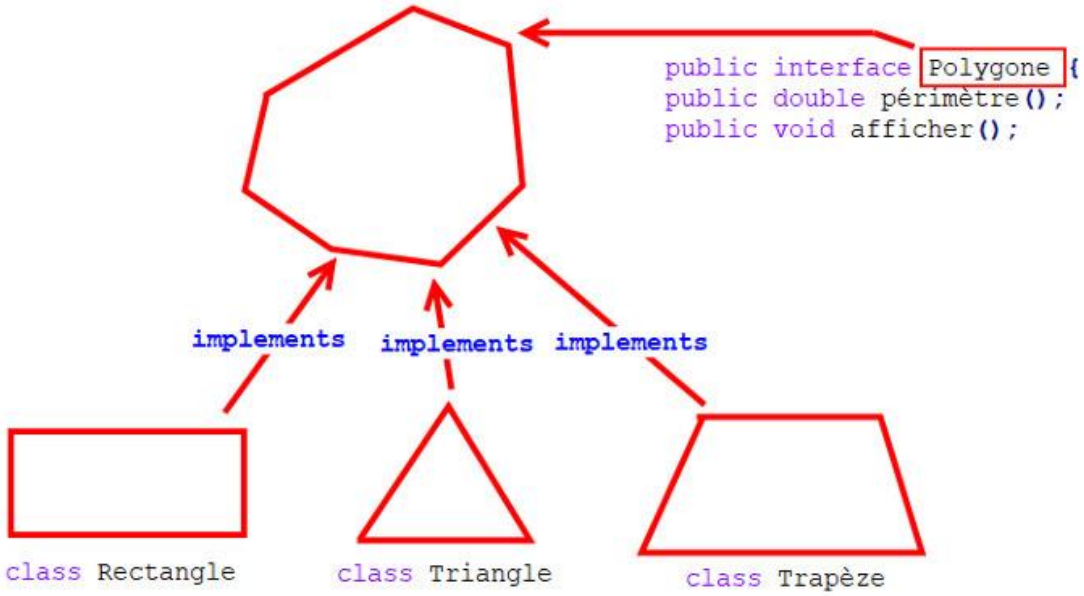
Cependant, une interface peut hériter de plusieurs interfaces et une classe peut implémenter plusieurs interfaces (interface1 extends interface2).

Pour bien comprendre l'utilité des interfaces en Java, prenons un exemple simple :

- On souhaite par exemple créer une classe Rectangle qui permet de générer un objet rectangle
- On souhaite aussi créer une classe triangle permettant de générer un objet triangle
- On souhaite créer une classe trapèze permettant de générer un objet trapèze
- …….

Les objets rectangle, triangle et trapèze possèdent tous des propriétés communes comme : surface, périmètre, nombre de cotés… Et on souhaite créer un moyen qui permet de gérer ce processus ie, un triangle doit obligatoirement posséder une méthode qui calcule le périmètre et une méthode qui calcul la surface, faute de quoi le développeur rencontre un message d'erreur lui indiquant qu'il doit obligatoirement insérer les méthodes périmètre() et surface()

C'est à ce niveau-là qu'une interface intervient ! On doit créer une et appelons-la par exemple interface polygone qui contient l'ensemble des méthodes communes aux formes géométriques cités ci-dessus :



En pratique, la création et la déclaration d'une interface est semblable à une classe, à la différence que l'on remplace le mot-clé class par interface.

```
public interface Polygone {
       public double périmètre();
       public int nombreCotes();
}
```

Les classes rectangle, triangle et trapèze vont être maintenant déclarées de la même façon comme celle de l'héritage, à l a différence de remplacer le mot clé extends par implements:

```
public class Rectangle implements Polygone {
public double Longueur;
public double largeur;
public Rectangle(double L,double l){
       this.Longueur=L;
       this.largeur=l;
}

       public double périmètre(){
              return 2*(this.Longueur+this.largeur);
       }
    public int nombreCotes(){
              return 4;
       }

       public void afficher(){
 System.out.println("La périmètre du rectangle est : "+
this.périmètre());
       }
       public static void main(String[] args) {
       Polygone R1=new Rectangle(7,4);
       Rectangle R2=new Rectangle(9,3);
```

```
R1.afficher();
       R2.afficher();
       }
}
```

Class Rectangle

```
public class Triangle implements Polygone{
       public double coté1;
       public double coté2;
       public double coté3;
       //création du constructeur
       public Triangle(double c1,double c2,double c3){
              this.coté1=c1;
              this.coté2=c2;
              this.coté3=c3;
       }

       public double périmètre(){
              return this.coté1+this.coté2+this.coté3;
       }
       public int nombreCotes(){
              return 3;
       }
       public void afficher(){
    System.out.println("La périmètre du triangle est : "+
this.périmètre());
       }
       public static void main(String[] args) {
       Polygone T1=new Triangle(2,3,5);
       Triangle T2=new Triangle(4,4,5);
       T1.afficher();
       T2.afficher();

       }
}
```

Enfin, ce que vous devez retenir :

- Une classe est définie comme abstraite avec le mot clé "abstract".
- Les classes abstraites sont à utiliser lorsqu'une classe mère ne doit pas être instanciée.
- Une classe abstraite ne peut donc pas être instanciée.
- Une classe abstraite n'est pas obligée de contenir de méthode abstraite.
- Si une classe contient une méthode abstraite, cette classe doit alors être déclarée abstraite.
- Une méthode abstraite n'a pas de corps.
- Une interface est une classe 100 % abstraite.
- Aucune méthode d'une interface n'a de corps.
- Une interface sert à définir un super-type et à utiliser le polymorphisme.

- Une interface s'implémente dans une classe en utilisant le mot clé "implements".
- Vous pouvez implémenter autant d'interfaces que vous voulez dans vos classes
- Vous devez redéfinir toutes les méthodes de l'interface (ou des interfaces) dans votre classe.

5.3. Exercices

Exercice1

Cet exercice est un exercice de base pour entrevoir l'intérêt des classes abstraites. Avec un peu d'imagination, on doit se rendre compte que le procédé qui y est utile peut servir dans un cadre plus réel. On définit quatre classes.

- La classe Animal est abstraite et déclare uniquement une méthode abstraite nommée action, sans paramètre et qui ne retourne rien.
- La classe Chien hérite de Animal et définit la méthode action qui écrit à l'écran "J'aboie".
- La classe Chat hérite de Animal et définit la méthode action qui écrit à l'écran "Je miaule".
- La classe EssaiChat contient trois champs statiques :
  - un champ statique pour un attribut de type java.util.Random qui est initialisé dès sa définition
  - une méthode statique nommée tirage sans paramètre qui retourne un Animal qui a une chance sur deux d'être un Chat et une chance sur deux d'être un Chien.
  - une méthode main qui utilise la méthode tirage et invoque la méthode action sur l'animal obtenu par cette méthode.

Correction

```
abstract class Animal
{
   abstract void action();
}
class Chien extends Animal
{
    void action()
    {
     System.out.println("J'aboie");
    }
}
class Chat extends Animal
{
    void action()
    {
     System.out.println("Je miaule");
    }
}
class EssaiAnimal
{
    static java.util.Random alea = new java.util.Random();
```

```
    static Animal tirage()
    {
     return (Math.abs(alea.nextInt()) % 2 == 0 ? new Chien() : new
Chat());
    }
    public static void main(String[] arg)
    {
     tirage().action();
    }
}
```

Exercice2

On considère la classe suivante :

```
abstract class Client {
protected String nom, prenom, phone, mail; int age;
Client (String nom, String prenom,int age) { this.nom = nom;
this.prenom = prenom; this.age=age;}
public abstract String getPhone(); //permet de retourner le numéro
de téléphone
public abstract String getMail(); //permet de retourner l'adresse
mail
}
```

Et l'interface suivante :

```
public interface TrancheAge {
 public String getAge();  // renvoie Adulte si l'âge courant est supérieur à 18, et Mineur sinon
}
```

Définir une classe Personne héritant de la classe Client de telle sorte qu'à tout moment, on saura si la personne est adulte ou non. On note que lors de la création des instances, les variables sont initialisées.

Correction

```
class Personne extends Client implements TrancheAge {
Personne(String nom, String prenom, int age, String mail,
String phone) {
super(nom,prenom,age) ;
this.mail = mail ;
this.phone = phone ;
}
public String getPhone() {
return phone ;
}
public String getMail() {
return mail ;
}
public String getAge(){
        if (this.age>18) {return "Adulte";}
            else return "Mineur";
};
```

Annales

Sujets d'examens

Cette partie regroupe un ensemble de sujets d'examen avec correction pour permettre à l'apprenant de tester ses connaissances.

Exercice 1 : **Considérons la classe Point suivante** ***(14 Points)***

```
public class Point {
        // Les coordonnées du point
        private double abs,
        private double ord;
        //constructeur par défaut initialiser les cordonnées du point à 0.0
        public Point () {
                abs = 0; ord = 0;
        }
        // Initialiser le point courant à partir des cordonnées d'un autre point
        public Point (Point p) {
                abs = p.abs; ord = p.ord;
        }
        // Initialiser le point courant avec les coordonnées passées en paramètres
        public Point (double abs, double ord) {
                this.abs = abs; this.ord = ord;
        }
        // Déplacer le point avec les valeurs données en paramètres
        public void deplacer(double dx, double dy) {
                abs += dx; ord += dy;
         }
        public void afficher() {
                System.out.println("abscisse : " + abs + " ordonnée : " + ord);
        }
        // Retourner une copie du point courant
         public Point clone() {
                Point p = new Point(); p.abs = this.abs; p.ord = this.ord; return p;
        }
}
```

A) Soit une classe Rectangle qui utilise la classe Point :

```
public class Rectangle {
        private Point pig; //le point inférieur gauche du rectangle
        private double largeur;
        private double longueur ;
        public void monIdentite() {
                System.out.println("Je suis un rectangle");
        }
}
```

Complétez les méthodes suivantes de la classe Rectangle:

1. **Initialiser les cordonnées du rectangle, sa largeur et sa longueur à 0.0** ***(1 Point)***

```
public Rectangle() {
. . . . . .
. . . . . .
 . . . . . .
}
```

2. Initialiser les coordonnées du rectangle avec les coordonnées du point passé en paramètre. La largeur et la longueur sont également données en paramètres. ***(01 Point)***

public Rectangle (Point p, double largeur, double longueur) {

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

}

3. Les coordonnées du rectangle, sa largeur, sa longueur sont données en paramètres ***(01Point)***

public Rectangle(double abs, double ord, double largeur, double longueur) {

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

}

4. Déplacer un rectangle avec les données passées en paramètres. ***(1.5 Points)***

public void deplacer(double dx, double dy) {

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

}

5. Afficher l'identité et toutes les informations d'un rectangle ***(1.5 Points)***

public void afficher() {

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

}

6. Retourner une copie du rectangle courant ***(1 Point)***

public Rectangle clone() {

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

}

B) Considérons maintenant une classe Carre qui est un rectangle particulier.

public class Carre extends Rectangle {

}

1. Complétez la définition de la classe Carre

public Carre () { ***(01 Point)***

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

}

public Carre(Point p, double cote) { ***(01 Point)***

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

}

public Carre(double abs, double ord, double cote) { ***(01 Point)***

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

}

public void deplacer(double dx, double dy) { ***(01 Point)***

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

}

public void monIdentite() { ***(0.5 Point)***

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

}

public void afficher() { ***(0.5 Point)***

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

}

2. Quel est le résultat de l'exécution des instructions suivantes :

Rectangle a = new Carre(new Point(2,5),10); ***(01 Point)***
a.afficher();

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

Carre b = new Rectangle(new Point(3,4),10,15); ***(01 Point)***
b.afficher();

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

Exercice 2 : ***(06 Points)***

Considérons deux interfaces Imprimable et Joignable ainsi qu'une classe nommée Humain

```
public interface Joignable {
  public String getPhone();  //permet de retourner le numéro de téléphone
  public String getMail();  //permet de retourner l'adresse mail
}

public interface Imprimable {
   public void afficher(); //permet d'afficher le nom, prénom, age, nss, phone, mail
}

public class Humain {
   private String nom, prenom;
   private int age;
   Humain(String nom, String prenom, int age) {
```

```
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
  }
  public void imprime() {

System.out.println("nom : " + nom + "prenom : " + prenom + "age : " + age);
}
}
```

Sachant qu'un enseignant est un humain qui possède un grade (grade) et un numéro de la sécurité sociale (nss) et qui doit être à la fois imprimable et joignable, définissez la classe Enseignant.

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

Correction - EMD 2016-2017

Exercice 1 : **Considérons la classe Point suivante** ***(14 Points)***

```
public class Point {
        // Les coordonnées du point
        private double abs,
        private double ord;
        //constructeur par défaut initialiser les coordonnées du point à 0.0
        public Point () {
                abs = 0; ord = 0;
        }
        // Initialiser le point courant à partir des cordonnées d'un autre point
        public Point (Point p) {
                abs = p.abs; ord = p.ord;
        }
        // Initialiser le point courant avec les coordonnées passées en paramètres
        public Point (double abs, double ord) {
                this.abs = abs; this.ord = ord;
        }
        // Déplacer le point avec les valeurs données en paramètres
        public void deplacer(double dx, double dy) {
                abs += dx; ord += dy;
         }
        public void afficher() {
                System.out.println("abscisse : " + abs + " ordonnée : " + ord);
        }
        // Retourner une copie du point courant
         public Point clone() {
                Point p = new Point(); p.abs = this.abs; p.ord = this.ord; return p;
        }
}
```

A) Soit une classe Rectangle qui utilise la classe Point :

```
public class Rectangle {
        private Point pig; //le point inférieur gauche du rectangle
        private double largeur;
        private double longueur ;
        public void monIdentite() {
                System.out.println("Je suis un rectangle");
        } }
```

Complétez les méthodes suivantes de la classe Rectangle:

7. **Initialiser les cordonnées du rectangle, sa largeur et sa longueur à 0.0** ***(1 Point)***

```
public Rectangle() {
. . . . . pig = new Point() ;. . . . .0.5 Pt . . . . . .
. . . . . largeur = 0 ;  . . . . . .
. . . . . longueur = 0 ;  . . . . . .
}
```

8. **Initialiser les coordonnées du rectangle avec les coordonnées du point passé en paramètre. La largeur et la longueur sont également données en**

paramètres. ***(01 Point)***

public Rectangle (Point p, double largeur, double longueur) {

. . . . . ***pig = new Point(p) ;*** . . . . . ***0.5 Pt***. . . . . .
. . . . . ***this.largeur = largeur ;*** . . . . . .
. . . . . ***this.longueur = longueur ;*** . . . . . .

}

9. Les coordonnées du rectangle, sa largeur, sa longueur sont données en paramètres ***(01Point)***

public Rectangle(double abs, double ord, double largeur, double longueur) {

. . . . . ***pig = new Point(abs,ord) ;*** . . . . . ***0.5 Pt***. . . . . .
. . . . . ***this.largeur = largeur ;*** . . . . . .
. . . . . ***this.longueur = longueur ;*** . . . . . .

0.5 Pt

}

10. Déplacer un rectangle avec les données passées en paramètres. ***(1.5 Points)***

public void deplacer(double dx, double dy) {

. . . . . .
. . . . . ***pig.deplacer(dx,dy) ;***. . . . . .
. . . . . .
}

11. Afficher l'identité et toutes les informations d'un rectangle ***(1.5 Points)***

public void afficher() {

. . . . ***monIdentite() ;***. . . ***0.5 Pt***. . . . . .
. . . . . ***pig.afficher()***; . . . . ***0.5 Pt***.. . . . . .
. . . . ***System.out.println("largeur : " + largeur + " longueur : " + longueur);*** . . ***0.5 Pt***.

}

12. Retourner une copie du rectangle courant ***(1 Point)***

public Rectangle clone() {

Solution 1:

```
public Rectangle clone() {
return new Rectangle(pig,largeur,longueur);
}
```

Solution 2 :

```
public Rectangle clone() {
Rectangle r = new Rectangle();
r.pig = pig.clone();
r.largeur = this.largeur;
r.longueur = this.longueur;
return r;
}
```

}

B) Considérons maintenant une classe Carre qui est un rectangle particulier.

public class Carre extends Rectangle {

}

3. Complétez la définition de la classe Carre

public Carre () { ***(01 Point)***

. . . . . .
. . . . . ***super() ;*** . . . . . .
. . . . . .
}

public Carre(Point p, double cote) { ***(01 Point)***

. . . . . .
. . . . . ***super(p,cote,cote);*** . . . . . .
. . . . . .
}

public Carre(double abs, double ord, double cote) { ***(01 Point)***

. . . . . .
. . . . . ***super(abs,ord, cote,cote);*** . . . . . .
. . . . . .
}

public void deplacer(double dx, double dy) { ***(01 Point)***

. . . . . .
. . . . . ***super.deplacer(dx,dy);*** . . . . . .
. . . . . .
}

public void monIdentite() { ***(0.5 Point)***

. . . . . .
. . . . . ***System.out.println("Je suis un carre");*** . . . . . .
. . . . . .
}

public void afficher() { ***(0.5 Point)***

. . . . . .
. . . . . ***super.afficher();*** . . . . . .
. . . . . .
}

4. Quel est le résultat de l'exécution des instructions suivantes :

Rectangle a = new Carre(new Point(2,5),10); ***(01 Point)***
a.afficher();

. . . . . ***Je suis un carre*** . . . . . .
. . . . . ***abscisse : 2.0 ordonnee : 5.0*** . . . . . .
. . . . . ***largeur : 10.0 longueur : 10.0*** . . . . . .

Carre b = new Rectangle(new Point(3,4),10,15); ***(01 Point)***
b.afficher();

Erreur : L'affectation n'est pas autorisée vu que Carre est une classe descendante de la classe Rectangle

. . . . . ***Cette instruction nécessite un transtypage*** . . . . . .

Exercice 2 : ***(06 Points)***

Considérons deux interfaces Imprimable et Joignable ainsi qu'une classe nommée Humain

```
public interface Joignable {
  public String getPhone();  //permet de retourner le numéro de téléphone
  public String getMail();  //permet de retourner l'adresse mail
}

public interface Imprimable {
   public void afficher(); //permet d'afficher le nom, prénom, age, nss, phone, mail
}
public class Humain {
   private String nom, prenom;
   private int age;
   Humain(String nom, String prenom, int age) {
       this.nom = nom;
      this.prenom = prenom;
      this.age = age;
   }
   public void imprime() {

System.out.println("nom : " + nom + "prenom : " + prenom + "age : " + age);
}
}
```

Sachant qu'un enseignant est un humain qui possède un grade (grade) et un numéro de la sécurité sociale (nss) et qui doit être à la fois imprimable et joignable, définissez la classe Enseignant.

```
public class Enseignant extends Humain implements Imprimable, Joignable { (01 Point)
   private String nss;
   private String grade;
   private String phone;
   private String mail;
```

(01 Point = 0.25 pour chaque ligne)

```
   Enseignant(String nom, String prenom, int age, String nss, String grade, String phone,
String mail ) {
      super(nom,prenom,age); (01 Point)
      this.nss = nss;
      this.grade = grade;
      this.phone = phone;
      this.mail = mail;
   }
     public void afficher() { (01 Point)
     imprime(); // un autre solution : super.imprime()
```

(01 Point = 0.25 pour chaque ligne)

```
    System.out.println("nss : " + nss + "grade : " + grade + "phone : " + phone + "mail :
"+ mail);
  }
  public String getPhone() { (0.5 Point)
    return phone;
  }
  public String getMail() { (0.5 Point)
    return mail;
  }
   }
```

EMD 2017-2018

***Exercice 1* : (2.5 pts)**

Considérons la classe Compteur

class Compteur { ***private int compteur = 0;*** ***Compteur( ) { compteur += 1;}*** ***public static void main(String[] args) {*** ***Compteur c1 = new Compteur();*** ***Compteur c2 = new Compteur();*** ***Compteur c3 = c1; Compteur c4 = c2;*** ***System.out.println(" compteur est : " +*** ***compteur);*** ***}*** ***}***	Est ce qu'il y a une erreur de compilation dans ce programme ? OUI / NON. Si c'est oui, justifiez : . . . . .

***Exercice 2* : (2.5 pts)**

Considérons les classes suivantes :	Mettez une croix dans la case correspondant à la bonne réponse, si l'instruction proposée est insérée à l'endroit proposé

```
class A { void fairececi() { }; }
class B extends A { void fairececi() { } }
class C { void fairececi() { } }
class Principale {
        public static void main(String[] args)
{
                A a = new A(); B b = new B();
                /*insérer  ici*/
        }
}
```

	Erreur de compilation	Pas d'erreur
A x = a;		
A x = b;		
B x = a;		
B x = b;		
A x = (A) b ;		

Exercice 3 : (3 pts)

Entourez l'unique bonne réponse.

```
1)  class Forme {
2)     private String color;
3)     public Forme(String color) {
4)        System.out.print("Forme");
5)        this.color = color; }
6)  }
7)  class Rectangle extends Forme {
8)     public Rectangle() {
9)        super("bleu");
10) System.out.print("Rectangle"); }
11) }
12) public class Exo3 {
13)    public static void main(String[] args)
14) {
15)       new Rectangle(); }
    }
```

Erreur de compilation dans la ligne 9
Erreur de compilation ligne 14
Pas d'erreur, le programme affiche Forme
Pas d'erreur, le programme affiche Rectangle
Pas d'erreur, le programme affiche FormeRectangle
Pas d'erreur, le programme affiche RectangleForme

Exercice 4 : (2 pts)

Sachant qu'un cheval peut se comporter comme un Animal et il peut se comporter comme un moyen de transport. Parfois il est de la classe Animal et parfois de la classe MoyenDeTransport. Comme l'héritage multiple n'est pas supporté en java, proposez un mécanisme permettant de remédier à cette lacune du langage JAVA.

Expliquez votre démarche :

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

Exercice 5 : (10 pts)

Considérons la classe de base EtreVivant dont le comportement crie() et marche() ne peuvent pas être définis à ce niveau d'abstraction. En revanche, tous les êtres vivants mangent de la même façon.

```
abstract class EtreVivant {
	private String nom;
	private int age;
	public EtreVivant (String nom, int age) {
		this.nom = nom;
		this.age = age;
	}
	abstract void crie();
	abstract void marche();
	void mange() {
		System.out.println("Je mange comme tout être vivant");
	}
	void affiche() {
		System.out.println("(" + nom + "," + age + ")");
	}
}
```

Considérons deux classes : les bipèdes et les quadrupèdes, tels qu'un bipède est un être vivant marchant sur deux pattes et un quadrupède est un être vivant qui marche sur 4 pattes. Les bipèdes ne crient pas tous de la même façon et même chose pour les quadrupèdes. Le comportement marche() pour un bipède consiste à afficher le message « **je marche sur deux pattes** », quant au quadrupède c'est le message « je marche sur 4 pattes »

1) Définir les deux classes Bipede et Quadrupede

Classe Bipede **(2pts)**	Classe Quadrupede **(2pts)**
. . . . .	. . . . .

Considérons maintenant deux classes Homme et Chien de sorte qu'un homme est un bipède et un chien est un quadrupède. Le comportement crie() pour un homme consiste à afficher le message : « **je crie comme un homme** » et quant au chien ça sera le message : « **je crie comme un chien** »

2) Définir les deux classes Homme et Chien

Classe Homme **(2pts)**	Classe Chien **(2pts)**
. . . . .	. . . . .

3) Que fournit le programme suivant : (2 pts)

Programme	**Résultats**
class Exo5 { ***public static void main(String[] args) {*** ***EtreVivant etrevivant = new Homme("mohand",30);*** ***etrevivant.marche();*** ***etrevivant.crie();*** ***etrevivant = new Chien("felix",5);*** ***etrevivant.marche();*** ***etrevivant.crie();*** ***}*** ***}***	. . . . .

Correction - EMD 2017-2018

***Exercice 1* : (2.5 pts)**

Considérons la classe Compteur

class Compteur { ***private int compteur = 0;*** ***Compteur( ) { compteur += 1;}*** ***public static void main(String[] args) {*** ***Compteur c1 = new Compteur();*** ***Compteur c2 = new Compteur();*** ***Compteur c3 = c1; Compteur c4 = c2;*** ***System.out.println(" compteur est : " + compteur);*** ***}*** ***}***	Est ce qu'il y a une erreur de compilation dans ce programme ? OUI / NON. Si c'est oui, justifiez : . . . . ***OUI***. . . . . ***Attribut compteur non statique manipulé dans méthode statique.*** . . . .

***Exercice 2* : (2.5 pts)**

Considérons les classes suivantes :	Mettez une croix dans la case correspondant à la bonne réponse, si l'instruction proposée est insérée à l'endroit proposé

```
class A { void fairececi() { }; }
class B extends A { void fairececi() { } }
class C { void fairececi() { } }
class Principale {
        public static void main(String[] args)
{
                A a = new A(); B b = new B();
                /*insérer  ici*/
        }
}
```

	Erreur de compilation	Pas d'erreur
A x = a;		X
A x = b;		X
B x = a;	X	
B x = b;		X
A x = (A) b ;		X

Exercice 3 : (3 pts)

Entourez l'unique bonne réponse.

```
1)  class Forme {
2)     private String color;
3)     public Forme(String color) {
4)        System.out.print("Forme");
5)        this.color = color; }
6)  }
7)  class Rectangle extends Forme {
8)     public Rectangle() {
9)        super("bleu");
10) System.out.print("Rectangle"); }
11) }
12) public class Exo3 {
13)    public static void main(String[] args)
14) {
15)       new Rectangle(); }
    }
```

- Erreur de compilation dans la ligne 9
- Erreur de compilation ligne 14
- Pas d'erreur, le programme affiche Forme
- Pas d'erreur, le programme affiche Rectangle
- Pas d'erreur, le programme affiche FormeRectangle
- Pas d'erreur, le programme affiche RectangleForme

Exercice 4 : (2 pts)

Sachant qu'un cheval peut se comporter comme un Animal et il peut se comporter comme un moyen de transport. Parfois il est de la classe Animal et parfois de la classe MoyenDeTransport. Comme l'héritage multiple n'est pas supporté en java, proposez un mécanisme permettant de remédier à cette lacune du langage JAVA.

Expliquez votre démarche :

En utilisant les interfaces. Nous définissions deux interfaces MoyenDeTransport dans laquelle nous allons déclarer tous les comportements des moyens de transport. Une autre interface Animal dans laquelle nous allons également déclarer tous les comportements de la classe Animal. Puis nous définissons une classe Cheval qui va implémenter les deux interfaces précédentes.

Exercice 5 : (10 pts)

Considérons la classe de base EtreVivant dont le comportement crie() et marche() ne peuvent pas être définis à ce niveau d'abstraction. En revanche, tous les êtres vivants mangent de la même façon.

```
abstract class EtreVivant {
    private String nom;
```

```
    private int age;

public EtreVivant (String nom, int age) {
            this.nom = nom;
            this.age = age;
    }

abstract void crie();
abstract void marche();
void mange() {
System.out.println("Je mange comme tout être vivant");

}
void affiche() {

System.out.println("(" + nom + "," + age + ")");

 }
}
```

Considérons deux classes : les bipèdes et les quadrupèdes, tels qu'un bipède est un être vivant marchant sur deux pattes et un quadrupède est un être vivant qui marche sur 4 pattes. Les bipèdes ne crient pas tous de la même façon et même chose pour les quadrupèdes. Le comportement marche() pour un bipède consiste à afficher le message « **je marche sur deux pattes** », quant au quadrupède c'est le message « je marche sur 4 pattes »

1) Définir les deux classes Bipede et Quadrupede

Classe Bipede **(2pts)**	Classe Quadrupede **(2pts)**
abstract class Bipede extends EtreVivant { ***Bipede(String nom, int age) {*** ***super(nom,age);*** ***}*** ***void marche() {*** ***System.out.println("Je marche sur deux pattes");*** ***}*** ***}***	***abstract class Quadrupede extends EtreVivant {*** ***Quadrupede(String nom, int age) {*** ***super(nom,age);*** ***}*** ***void marche() {*** ***System.out.println("Je marche sur 4 pattes");*** ***} }***

Considérons maintenant deux classes Homme et Chien de sorte qu'un homme est un bipède et un chien est un quadrupède. Le comportement crie() pour un homme consiste à afficher le message : « **je crie comme un homme** » et quant au chien ça sera le message : « **je crie comme**

un chien »

2) Définir les deux classes Homme et Chien

Classe Homme **(2pts)**	Classe Chien **(2pts)**
class Homme extends Bipede { ***Homme(String nom, int age) {*** ***super(nom,age);*** ***}*** ***void crie() {*** ***System.out.println("Je crie comme un homme");*** ***}*** ***}***	***class Chien extends Quadrupede {*** ***Chien(String nom, int age) {*** ***super(nom,age);*** ***}*** ***void crie() {*** ***System.out.println("Je crie comme un chien");*** ***}*** ***}***

3) Que fournit le programme suivant : (2 pts)

Programme	**Résultats**
class Exo5 { ***public static void main(String[] args) {*** ***EtreVivant etrevivant = new Homme("mohand",30);*** ***etrevivant.marche();*** ***etrevivant.crie();*** ***etrevivant = new Chien("felix",5);*** ***etrevivant.marche();*** ***etrevivant.crie();*** ***}*** ***}***	. . . . **Je marche sur deux pattes** **Je crie comme un homme** **Je marche sur 4 pattes** **Je crie comme un chien** . . . .

EMD 2018-2019

Exercice 1 **: Pour chacun des morceaux de programme suivants, donnez le résultat d'exécution s'il existe (expliquez le résultat obtenu). En cas d'erreur, justifier. (4 Pts)**

(a) (2 pts)

```
class Parent {
        Parent() {
                System.out.print("A");
        } }
class Child extends Parent {
        public Child(int x) {
                System.out.print("B");
        }
        public Child() {
                this(123);
                System.out.print("C");
        } }
class Test1{
        public static void main(String[] args) {
                new Child();
        } }
```

. . . . . .

(b) (2 pts)

```
abstract class Oeuf {
protected abstract void getNomVolaille() ;
public double getPrixVolaille(int x) {
return x*2 ; } }
class Poule extends Oeuf {
void getNomVolaille()  {
 System.out.println("Je suis une Poule "); }
public static void main (String args[]) {
   Oeuf p=new Poule();
   p.getNomVolaille();
   p.getPrixVolaille(2);
} }
```

. . . . . .

Exercice 2 **:** ***(2.5*** pts)

Considérons les classes suivantes :	Mettez une croix dans la case correspondant à la bonne réponse, si l'instruction proposée est insérée à l'endroit proposé.

```
class Arme {
void attaquer() {…….}
}
class Pistolet extends Arme {
void tirer(){}
}
class Arsenal {
public void main (String [] args) {
Arme a1 = new Arme();
Arme a2 = new Pistolet();
Pistolet p = new Pistolet();
/*** insérer ici ****/
} }
```

	Erreur de compilation/ exécution	Pas d'erreur
a2.tirer()		
(Pistolet) a2.tirer()		
p.attaquer()		
(Pistolet) a1.tirer()		
a1=p ; a2.tirer() ;		

***Exercice 2* : (2.5 pts)**

On vous donne le morceau de programme suivant :	Dites si ce programme est correct, si oui donnez le résultat (justifier votre réponse).
class Test { ***static int i=0;*** ***static void f(){System.out.println("i="+i);}*** ***static void g(){ this.f();}*** ***public static void main (String args[]) {*** ***Test a = new Test();*** ***a.g();*** ***} }***	…… …… …… …… …… …… …… …… ……

***Exercice 3* : (3 pts)**

On vous donne le morceau de programme suivant :	Prédire l'exécution (expliquez le résultat obtenu)
class A{ ***String s="A";*** ***void f(String s1) { System.out.println(s+" "+s1); }*** ***}*** ***class B extends A {*** ***String s="B";*** ***void f(String s1) { super.f(s+" "+s1); }*** ***}*** ***class C extends B {*** ***String s="C";*** ***CC(String s) {***	. . . . .

`` this.s= s; } void f(String s1) { super.f(s+" "+s1); } public static void main (String args[]) { A a=new C("D"); a.f("X"); } } ```	

Exercice 4 : (3pts)

Soit la classe Compte	Completer les deux méthodes suivantes : **virerVers :** permet de transférer le montant donné du compte courant vers le compte donné en paramètre **virerDe :** permet de transférer le montant donné du compte donné vers le compte courant.
`` class Compte { private int solde ; Compte(int solde) { this.solde = solde; } void deposer(int montant) { solde += montant; } void retirer(int montant) { solde -= montant; } } ```	``` void virerVers(int montant, Compte destination) { . . . . } void VirerDe(int montant, Compte source) { . . . . } ```

Exercice 5 : (5pts)

Dans le programme correct suivant, que sera affiché dans le programme principal ?

```
class A {
        public String f(B objet) { return "A et B"; }
        public String f(A objet) { return "A et A"; }
        public String f() { return "f() de A"; }
}
class B extends A {
        public String f(B objet) { return "B et B";}
        public String f(A objet) { return "B et A";}
        public String f() { return "f() de B";}
}
class Test {
        public static void main(String[] args) {
                A a1 = new A() ;
                A a2 = new B() ;
                B b = new B() ;
                System.out.println(a1.f(a1) + "," +
a1.f());  . . . . . .
                System.out.println(a1.f(a2) + "," +
a2.f());  . . . . . .
                System.out.println(a1.f(b) + "," + a1.f());  . . . . . .
                System.out.println(b.f(a1) + "," + a1.f());  . . . . . .
                System.out.println(b.f(a1) + "," + a1.f());  . . . . . .
                System.out.println(a2.f(a1) + "," + a1.f());  . . . . . .
                System.out.println(a2.f(a2) + "," + a2.f());  . . . . . .
                System.out.println(b.f(a2) + "," + a2.f());  . . . . . .
                System.out.println(a2.f(b) + "," + b.f());  . . . . . .
                System.out.println(b.f(b) + "," + b.f());  . . . . . .
        }
}
```

Correction - EMD 2018-2019

Exercice 1 **: Pour chacun des morceaux de programme suivants, donnez le résultat d'exécution s'il existe (expliquez le résultat obtenu). En cas d'erreur, justifier. (4 Pts)**

Pour chaque cas, si la réponse est correcte : 0.5 Pt

1.5 Pt pour l'explication

(b) (2 pts)	
`` class Parent { Parent() { System.out.print("A"); } } class Child extends Parent { public Child(int x) { System.out.print("B"); } public Child() { this(123); System.out.print("C"); } } class Test1{ public static void main(String[] args) { new Child(); } } ```	*Ce programme est correct, le résultat est : ABC* 1. Appel au 2ème constructeur de child 2. Appel au constructeur de Parent : Affichage de "A" 3. Appel au 1er constructeur de child avec le paramètre 123 : Affichage de "B" 4. Affichage de "C"

(b) (2 pts)	
`` abstract class Oeuf { protected abstract void getNomVolaille() ; public double getPrixVolaille(int x) { return x*2 ; } } class Poule extends Oeuf { void getNomVolaille() { System.out.println("Je suis une Poule "); } public static void main (String args[]) { Oeuf p=new Poule(); p.getNomVolaille(); p.getPrixVolaille(2); } } ```	*Ce programme est incorrect.* *La méthode getNomVolaille() est déclarée avec protected dans la classe Oeuf , nous l'avons redéfini dans la classe Poule en réduisant sa visibilité.*

Exercice 2 **: (2.5** pts) **0.5 Pt pour chaque cas**

Considérons les classes suivantes :	Mettez une croix dans la case correspondant à la bonne réponse, si l'instruction proposée est insérée à l'endroit proposé.
class Arme { ***void attaquer() {…….}*** ***}*** ***class Pistolet extends Arme {*** ***void tirer(){}*** ***}*** ***class Arsenal {*** ***public void main (String [] args) {*** ***Arme a1 = new Arme();*** ***Arme a2 = new Pistolet();*** ***Pistolet p = new Pistolet();*** ***/*** insérer ici ****/*** ***} }***	

	Erreur de compilation/ exécution	Pas d'erreur
a2.tirer()	☒	
(Pistolet) a2.tirer()	☒	
p.attaquer()		☒
(Pistolet) a1.tirer()	☒	
a1=p ; a2.tirer() ;	☒	

Exercice 2 Bis ***: (2.5 pts)*** **0.5 Pt pour la réponse correcte et 2 Pts pour l'explication**

On vous donne le morceau de programme suivant :	Dites si ce programme est correct, si oui donnez le résultat (justifier votre réponse).
class Test { ***static int i=0;*** ***static void f(){System.out.println("i="+i);}*** ***static void g(){ this.f();}*** ***public static void main (String args[]) {*** ***Test a = new Test();*** ***a.g();*** ***} }***	***Ce programme est incorrect, nous avons utilisé une méthode non static dans une méthode static, en particulier pas de this dans une méthode static***

Exercice 3 ***: (3 pts)*** **1 Pt pour la réponse correcte et 2 Pts pour l'explication**

On vous donne le morceau de programme suivant :	Prédire l'exécution (expliquez le résultat obtenu)
class A{ ***String s="A";*** ***void f(String s1) { System.out.println(s+" "+s1); }*** ***}*** ***class B extends A {*** ***String s="B";*** ***void f(String s1) { super.f(s+" "+s1); }*** ***}***	**Ce programme est correct, le résultat est : A B D X** **1. Création de la référence a et appel au constructeur de la classe C avec le paramètre "C" : s="D"** **2. Exécution de f("X"), on utilise la méthode f de la classe C (principe de la liaison dynamique) : super.f("D"+" "+"X")** **3. On continue le calcul : super appelle la méthode de la classe mère B : super.f("B"+ "D X")** **4. On continue le calcul, super appelle la**

class C extends B { ***String s="C";*** ***C(String s) {*** ***this.s= s;*** ***}*** ***void f(String s1) { super.f(s+" "+s1); }*** ***public static void main (String args[]) {*** ***A a=new C("D"); a.f("X");*** ***} }***	**m éthode de la classe mère A : :** **super.f("A"+ "B D X")** 5. **Affichage de A B D X**

Exercice 4 *: (3pts) 1.5Pt pour chaque réponse*

Soit la classe Compte	Compléter les deux méthodes suivantes : **virerVers :** permet de transférer le montant donné du compte courant vers le compte donné en paramètre **virerDe :** permet de transférer le montant donné du compte donné vers le compte courant.
class Compte { ***private int solde ;*** ***Compte(int solde) {*** ***this.solde = solde;*** ***}*** ***void deposer(int montant) {*** ***solde += montant;*** ***}*** ***void retirer(int montant) {*** ***solde -= montant;*** ***}*** ***}***	***void virerVers(int montant, Compte destination) {*** ***retirer(montant);ou this.retirer(montant);*** ***destination.deposer(montant) ;*** ***}*** ***void VirerDe(int montant, Compte source) {*** ***source.retirer(montant) ;*** ***deposer(montant);ou this.deposer(montant);*** ***}***

Exercice 5 *: (5pts) 0.5Pt pour chaque affichage*

Dans le programme correct suivant, que sera affiché dans le programme principal ?

```
class A {
        public String f(B objet) { return "A et B"; }
        public String f(A objet) { return "A et A"; }
        public String f() { return "f() de A"; }
}
class B extends A {
        public String f(B objet) { return "B et B";}
        public String f(A objet) { return "B et A";}
        public String f() { return "f() de B";}
}
class Test {
        public static void main(String[] args) {
                A a1 = new A() ;
                A a2 = new B() ;
                B b = new B() ;
                System.out.println(a1.f(a1) + "," + a1.f());    . . . . . A et A,f() de A . . . . . .
. .

                System.out.println(a1.f(a2) + "," + a2.f());    . . . . A et A,f() de B ..................
```

```
        System.out.println(a1.f(b) + "," + a1.f());      . . . . . A et B,f() de A . . . . . .

        System.out.println(b.f(a1) + "," + a1.f());      . . . . . B et A,f() de A . . . .

        System.out.println(b.f(a1) + "," + a1.f());      . . . . . B et A,f() de A . .

        System.out.println(a2.f(a1) + "," + a1.f());     . . . . . B et A,f() de A . . . .

        System.out.println(a2.f(a2) + "," + a2.f());     . . . . . B et A,f() de B . . . . . .

        System.out.println(b.f(a2) + "," + a2.f());      . . . . . B et A,f() de B . . . . .

        System.out.println(a2.f(b) + "," + b.f());       . . . . . B et B,f() de B . . .

        System.out.println(b.f(b) + "," + b.f());        . . . . . .. . B et B,f() de B . . .

    }
}
```