

Leçon N°1

Syntaxe du langage JAVA

Introduction

Java est un langage de programmation orientée objet développé par **Sun Microsystems** en 1995 puis racheté par **Oracle** en 2009. Ce langage est utilisé par un grand nombre de programmeurs professionnels, ce qui en fait un langage dont l'apprentissage est incontournable.

Voici les principales caractéristiques de Java :

- Java permet de développer différents types d'applications :
 - ✓ des applications qui fonctionnent en *mode graphique* (fenêtres, menus, boutons, ...) ou en *mode console*
 - ✓ des applets, qui sont des programmes Java incorporés à des pages web
 - ✓ des applications pour appareils mobiles (tablettes, smartphones, etc.)
 - ✓ des animations en 3D, etc.
- Java se distingue par son excellente **portabilité** : une fois le programme créé, il fonctionnera automatiquement sous Windows, Mac OS, Linux, etc. (à condition de disposer d'une machine virtuelle Java).
- Java supporte le **multi-thread** (exécution de plusieurs tâches ou processus en parallèle).
- Le **JDK** (Java Development Kit) contient un ensemble de classes de base regroupées en *packages* (*java.lang*, *javax.swing*, *java.util*, *java.sql*, *java.net*, ...).

Remarque : Il ne faut pas confondre Java avec JavaScript (langage de script utilisé principalement sur les sites web), car Java n'a rien à voir.

I. Notion de machine virtuelle

D'habitude, la compilation d'un programme source génère un code exécutable (en langage machine) compréhensible par un système d'exploitation (Windows, Linux, Mac OS, ...).

Au contraire, la compilation d'un programme source en java donne lieu à un programme dans une forme intermédiaire appelée **bytecode**. Celui-ci peut être interprété par n'importe quelle machine sur laquelle est installé un logiciel appelé *machine virtuelle Java* (JVM). La machine virtuelle est dépendante de la plate-forme. On parle plus communément de **JRE** (**J**ava **R**untime **E**nvironment) (voir figure 1).

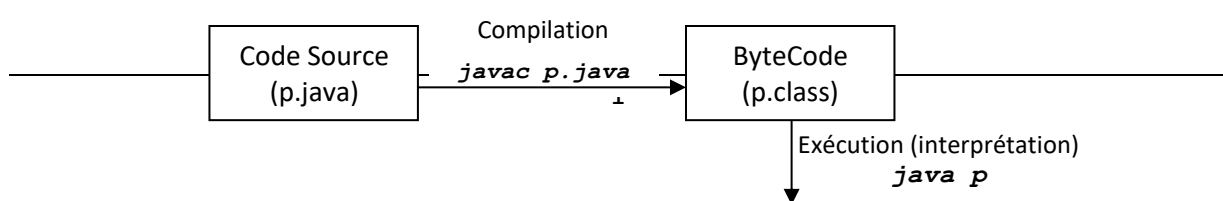


Figure 1 : Schéma d'exécution d'un programme Java
(Illustration du slogan *Compile once, run everywhere*)

Remarque : Afin de faciliter l'écriture des codes sources, on utilise généralement un outil de développement ou **IDE** (**I**ntegrated **D**evelopment **E**nvironment) comme *Eclipse*, *Netbeans* ou *IntelliJ IDEA*.

II. Structure d'un programme en Java

Tous les programmes Java sont composés d'au moins une **classe**. Pour être exécutable, une classe doit contenir une fonction (*méthode*) appelée **main()** qui constitue le point de démarrage du programme.

Exemple

```
// exemple de classe
public class HelloWorld {
    public static void main(String[] args) {
        System.out.print("Bonjour tout le monde !");
    }
}
```

Dans ce programme, l'instruction :

```
System.out.print("Bonjour tout le monde !");
```

signifie littéralement « demander à la méthode **print()** de l'objet **System.out** d'écrire le message "Bonjour tout le monde !" sur la console ».

Pour ajouter un saut de ligne à la fin du message, on peut :

- soit utiliser la méthode **println()** à la place de la méthode **print()** ;
- soit utiliser le caractère d'échappement `"\n"`.

```
System.out.println("Bonjour tout le monde !");
```

⇔

```
System.out.print("Bonjour tout le monde !\n");
```

Remarques

- Java est un langage sensible à la casse (il fait la différence entre minuscule et majuscule)
- Toutes les instructions en Java se terminent par un point-virgule ” ;”
- Les blocs de code sont délimités par des accolades « { » et « } »
- En Java, les commentaires unilignes sont introduits par les symboles //
- Les commentaires multilignes sont introduits par les symboles /* et se terminent par les symboles */.
- Le kit de développement Java (JDK) contient un outil très utile, appelé **javadoc**, qui génère une documentation automatique au format HTML à partir des fichiers source. Les commentaires destinés à cet outil sont introduits par les symboles /** et se terminent par les symboles */.

III. Les identificateurs

Un identificateur (de classe, d’objet, de variable, ...) commence obligatoirement par une lettre, un trait de soulignement « _ » ou un signe dollar « \$ ». Les caractères qui suivent peuvent contenir des chiffres.

Un identificateur ne doit pas appartenir à la liste des mots réservés du langage Java :

abstract	const	final	int	public	throw
assert	continue	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true
byte	double	goto	new	strictfp	try
case	else	if	null	super	void
catch	enum	implements	package	switch	volatile
char	extends	import	private	synchronized	while
class	false	instanceof	protected	this	

Le respect des règles et conventions suivantes permet d’augmenter la *lisibilité* d’un programme Java et d’en faciliter la compréhension par un autre programmeur :

- Les noms des classes commencent par une majuscule (exemple : `public class Personne`).
- Les identificateurs de variables ou de fonctions commencent par une minuscule (exemples : `main`, `somme`, ...).
- Si l’identificateur consiste en la juxtaposition de plusieurs mots, on utilise la notation **Camel Case** qui consiste à mettre en capitale la première lettre de chaque mot (exemples : `tabPersonne`, `calculSalaireMensuel`, ...).
- Les constantes symboliques sont écrites entièrement en majuscule (exemple : `final double PI = 3.14 ;`).

Comme dans la plupart des langages modernes, les déclarations sont obligatoires, cependant, il n’est pas nécessaire qu’elles soient regroupées en début de programme (comme cela est le cas en C ou en Pascal) ; il suffit simplement qu’une variable ait été déclarée avant d’être utilisée.

IV. Les Types de données primitifs (types de base)

Les types primitifs de Java se répartissent en quatre grandes catégories selon la nature des informations qu'ils permettent de représenter :

- nombres entiers,
- nombres flottants (réels),
- caractères,
- booléens.

Le tableau suivant montre une description des différents **types** de données offerts par Java :

Type	Nombre de bits	Intervalle	Valeur par défaut
byte	8 bits (1 octet)	-128 à 127	0
short	16 bits (2 octets)	-32768 à 32767	0
int	32 bits (4 octets)	-2E32 à 2E32-1	0
long	64 bits (8 octets)	-2E63 à 2E63-1	0
float	32 bits (4 octets)	-1,4E45 à 3,4E38	0.0
double	64 bits (8 octets)	-4,9E-324 à 1,8E308	0.0
char	16 bits (2 octets)	0 à 65 535	'\u0000'
boolean	1 bit	true/false	false

A chacun de ces types correspond une classe «enveloppe» (*wrapper class*) : **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character** et **Boolean**. Ces classes ont été développées principalement pour effectuer des conversions de types (voir paragraphe VI.2).

Remarques

1. Tous les types numériques sont signés
2. Une constante caractère figure toujours entre deux quotes simples (par exemple 'A').
3. Java ne dispose pas d'un type prédéfini pour les chaînes de caractères mais offre une classe **String**. Comme tous les objets, la valeur par défaut d'une variable de type String est **null**.
4. Le type **void** est utilisé principalement pour créer des fonctions (méthodes) qui ne retournent aucun résultat.

V. Les variables et les constantes

V.1. Les variables

Une variable possède un *nom* et un *type* qui peut être un type de base ou une classe. Il est également possible d'attribuer une valeur initiale à la variable lors de sa déclaration.

Syntaxe

nomType nomVariable [= expression] ;

Exemples

```
int e;
double d1 = 3.65, d2 = 365e-2;
float f = 3.33f;
char c = 'e';
String s = "le langage Java";
```

Remarques

- 1- Une variable peut contenir soit une donnée de type primitif soit une *référence* à un objet.
- 2- Par défaut un littéral représentant une valeur décimale est de type double : pour définir un littéral représentant une valeur décimale de type float il faut le suffixer par la lettre f ou F.
- 3- Java utilise le standard *Unicode* qui représente chaque caractère sur plusieurs octets ce qui permet de couvrir les caractères des différentes langues mondiales (y compris l'arabe). Le code des 256 premiers caractères est le même que le code ASCII.
- 4- La portée d'une variable s'étend jusqu'à la fin du bloc dans lequel elle est définie.

Exemple :

```
{
    {
        int a;
        ...
        // a est accessible
    }
    // a n'est plus accessible
}
```

V.2. Les constantes

Une constante est déclarée avec le modificateur **final**. Il est obligatoire de l'initialiser au moment de la déclaration. Une convention de nommage veut que l'on utilise toujours des noms en majuscule pour identifier les constantes. La déclaration d'une constante se fait donc conformément à la syntaxe suivante :

final nomType nomConstante = valeur ;

Exemples

```
final double PI = 3.14;
final double G = 9.8;
final int MAX = 100;
```

VI. Les expressions

Une expression est une combinaison entre variables et/ou constantes à l'aide d'opérateurs. Java est l'un des langages les plus riches en opérateurs.

VI.1. Les opérateurs usuels

Le tableau suivant fournit une description des principaux opérateurs disponibles en Java du plus prioritaire au moins prioritaire :

Niveau de priorité	Opérateurs	Notation
1	les parenthèses	()
2	les opérateurs d'incrément et décrémentation	++ --
3	les opérateurs de multiplication, division et modulo	* / %
4	les opérateurs d'addition et soustraction	+ -
5	les opérateurs de comparaison	< <= > >=
7	les opérateurs d'égalité	= !=
8	l'opérateur ET logique	&&
9	l'opérateur OU logique	
10	les opérateurs d'affectation	= += -= *= /= %=

Etant donné que les parenthèses possèdent une forte priorité, on peut les utiliser pour modifier l'ordre d'interprétation des opérateurs dans une expression.

Exemples

```

▪ boolean f = (5 > 3) && (16%3 == 0) ; // f = ...
▪ boolean t = !f; // t = ...
▪ int i = 2 + 6 / 4 ; // i = ...
▪ int j = i++ ; // j = ...
▪ int k = ++j ; // k = ...
▪ k += 3 ; // équivalent à .....

```

Remarques

1. Le tableau suivant explique la différence entre une incrémentation *préfixée* de la forme (++x) et une incrémentation *postfixée* de la forme (x++) ; on suppose qu'initialement x vaut 10 :

Incrémentation	Instructions équivalentes	Valeurs finales
y = x++;	y = x; x = x + 1;	x = .. y = ..
y = ++x;	x = x + 1; y = x;	x = .. y = ..

2. Lorsqu'on utilise deux opérandes de type *int* avec l'opérateur de division '/', c'est toujours la *division entière* qui est effectuée même si le résultat est affecté à une variable de type *float* ou *double*.

Exemple

```

double x = 3/2 ; // x = .....
double y = 3.0/2 ; // y = .....

```

VI.2. Les conversions de type (cast)

On appelle *conversion de type de données* ou *transtypage* le fait de modifier le type d'une donnée afin de pouvoir l'affecter à une variable ou effectuer un certain calcul. Cette conversion peut se faire de façon implicite ou explicite.

- **Conversion implicite** : une conversion implicite consiste en une modification du type de donnée effectuée automatiquement par le compilateur. Cela signifie que lorsque l'on va stocker un type de donnée dans une variable déclarée avec un autre type (plus grand), le compilateur ne retournera pas d'erreur mais effectuera une conversion *implicite* de la donnée avant de l'affecter à la variable.

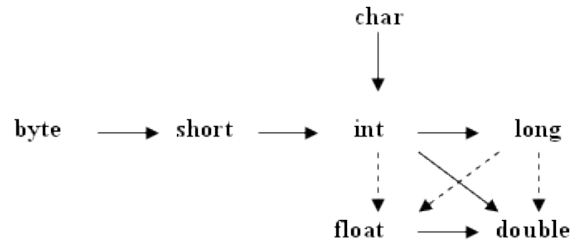
Exemple

```

short s = 2;
int n = s; // conversion short -> int
double x = n; // conversion int -> double (x = 2.0)
n = x; // Type mismatch: cannot convert from double to int

```

Dans l'image ci-dessous, les flèches pleines indiquent les conversions automatiques sans perte d'information alors que les flèches en pointillés indiquent les conversions automatiques avec une possible perte d'information :



- **Conversion explicite** : une conversion explicite consiste en une modification forcée du type de donnée. Pour cela, on utilise un *opérateur de cast* qui est tout simplement le type de donnée, dans lequel on désire convertir une variable ou une valeur, mis entre des parenthèses devant l'expression.

Exemple

```
double x = 8.694;
int n = (int) x;           // conversion double -> int (n = 8)
double x = (double) 3/2;   // x = .....
double y = (double) (3/2); // y = .....
```

VII. Les structures de contrôle

VII.1. L'instruction if

Syntaxe

```
if (cond°)
    Traitement_1;
[else
    Traitement_2;]
```

Exemple

```
if (a > b)
    max = a ;
else
    max = b ;
```

Remarques

1. Si le bloc *if* ou *else* contient plus d'une instruction, celles-ci doivent être obligatoirement mises entre accolades.
2. L'instruction précédente aurait pu être exprimée à l'aide de l'*opérateur conditionnel* sous la forme suivante :

```
max = (a > b) ? a : b ;
```

3. Dans le cas de plusieurs instructions *if* imbriquées, le *else* se rapporte toujours au dernier *if* rencontré auquel un *else* n'est pas encore attribué.

Exemple : Que va afficher le code suivant si a = 2; b = 1 et c = 2 ?

```
if (a<=b)
    if (b<=c)
        System.out.println("liste ordonnée");
    else
        System.out.println("Liste non ordonnée");
```

VII.2. La structure switch

Syntaxe

```
switch (var)           // var de type entier ou caractère
{
    case v1 :
        traitement_1;
        break;
    case v2 :
        traitement_2;
        break ;
    ...
    default : autre_taitement;
}
```

Exemple

```
public class Test {
    public static void main(String[] args) {
        String jour = "Samedi";
        switch (jour.toUpperCase()){
            case "LUNDI":
            case "MARDI":
            case "MERCREDI":
            case "JEUDI" :
            case "VENDREDI" : System.out.println("Journée de travail"); break;
            case "SAMEDI" :
            case "DIMANCHE" : System.out.println("Week End"); break;
            default : System.out.println("Journée inexistante");
        }
    }
}
```


VII.3. La boucle while

Syntaxe

```
while (condition) {  
    instruction_1 ;  
    ...  
    instruction_n ;  
}
```

Exemple : Le code suivant permet de calculer le factoriel d'un entier positif n en appliquant la formule :

$$n! = 1 * 2 * \dots * n$$

```
int f = 1 ;  
int i = 1;  
while (i <= n){  
    f = f * i;           // équivalente à f *= i;  
    i++;  
}  
System.out.println(n+"! = "+f);
```

VII.4. La boucle for

Syntaxe

```
for (initialisation; cond° de répétition; avancement)  
{  
    instruction 1 ;  
    ...  
    instruction n ;  
}
```

Exemple

Le code suivant permet de calculer le factoriel d'un entier positif n en appliquant la formule :

$$n! = 1 * 2 * \dots * n$$

```
int f = 1 ;  
for(int i = 1; i <= n; i++)  
    f = f * i;  
System.out.println(n+"! = "+f);
```

VII.5. La boucle do .. while (répéter .. tant que)

Syntaxe

```
do {
    instruction 1 ;
    ...
    instruction n ;
} while (condition) //condition de répétition
```

Notons que dans cette structure la vérification de la condition ne se fait qu'à la fin. Par conséquent, cette boucle est toujours exécutée au moins une fois.

Exemple

Le code suivant permet de calculer le factoriel d'un entier positif n en appliquant la formule :

$$n! = 1 * 2 * \dots * n$$

```
int f = 1, i = 1;
do {
    f = f * i;
    i++;
} while (i <= n);
System.out.println(n+"! = "+f);
```

VIII. Les entrées/sorties

Les nombreuses classes prédéfinies de Java sont regroupées par catégorie dans des *packages* (bibliothèques de classes).

- les packages de noyau commencent par java (comme *java.util*)
- les packages d'extension commencent par javax (comme *javax.swing*) qui permet de créer des interfaces graphiques.

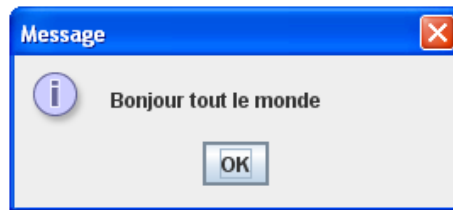
La classe **JOptionPane** contenue dans le package javax.swing fournit des boîtes de dialogue permettant d'afficher des messages, lire des données, etc.

Pour utiliser les services d'une classe particulière, on utilise la clause **import** au début du programme.

Exemple 1

```
import javax.swing.JOptionPane;
public class Bonjour {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Bonjour tout le monde");
    }
}
```

Output

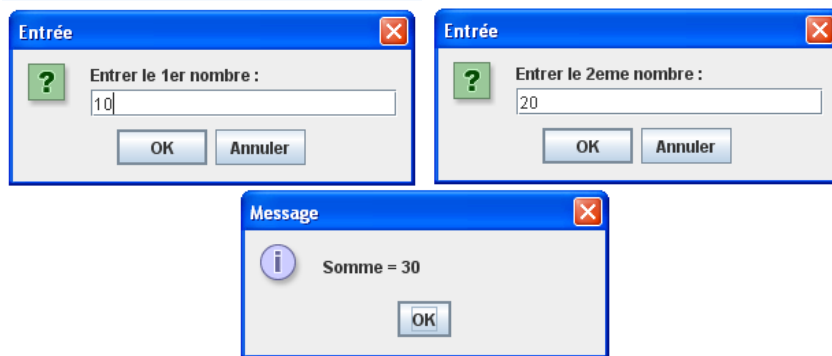


La méthode **showMessageDialog()** qui permet de créer une boîte de message requiert 2 arguments:

- le 1^{er} argument détermine le composant parent de la boîte de dialogue (généralement un frame ou cadre). L'option **null** affiche la boîte au milieu de l'écran.
- le 2^{ème} argument représente le contenu du message à afficher.

Exemple 2 : Le programme suivant permet de lire 2 entiers en utilisant des boîtes d'entrée puis affiche leur somme dans une boîte de message :

```
import javax.swing.JOptionPane;
public class Calcul {
    public static void main(String[] args) {
        String nombre1, nombre2;
        int n1, n2, somme;
        nombre1 = JOptionPane.showInputDialog("Entrer le 1er nombre :");
        n1 = Integer.parseInt(nombre1);
        nombre2 = JOptionPane.showInputDialog("Entrer le 2ème nombre :");
        n2 = Integer.parseInt(nombre2);
        somme = n1+n2;
        JOptionPane.showMessageDialog(null, "Somme = "+somme);
    }
}
```



- La méthode `showInputDialog()` permet d'afficher une boîte d'entrée pour saisir des données à partir du clavier. Cette méthode renvoie comme résultat un objet de type String (chaîne de caractères).
- La méthode `Integer.parseInt()` permet de convertir un texte en entier.
- Pour convertir la chaîne entrée en réel, on peut utiliser la méthode `Double.parseDouble()` ou `Float.parseFloat()`.

Remarque : En java, la saisie d'une donnée au clavier en mode **console** nécessite les étapes suivantes :

1. Importation de la classe « Scanner » du package « java.util » :

```
import java.util.Scanner;
```

2. Création d'une instance de la classe Scanner comme par exemple :

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

3. Lecture de la donnée de type int, double, String, ... dans une variable en utilisant la fonction appropriée :

```
int e = clavier.nextInt();
double d = clavier.nextDouble();
String s = clavier.nextLine();
etc.
```

Exemple : le programme suivant permet de lire un réel positif de type double puis affiche sa racine carrée.

```
import java.util.Scanner;
public class RacineCarree {
    public static void main(String[] args) {
        double x;
        Scanner clavier = new Scanner(System.in);
        do {
            System.out.print("Entrer un nombre positif : ");
            x = clavier.nextDouble();
        } while (x < 0);
        System.out.println("racine("+x+") = "+Math.sqrt(x));
    }
}
```

La boucle do .. while est utilisée pour obliger l'utilisateur à entrer un nombre positif.

4. Le programme suivant montre comment formater un résultat de type décimal avant de l'afficher :

```
import java.text.DecimalFormat;
public class Test {
    public static void main(String parms[]){
        final double PI = 3.14159265;
        DecimalFormat précision4 = new DecimalFormat("0.0000");
        System.out.println("PI = " + précision4.format(PI));
    }
}
```

Résultat obtenu

PI = 3,1416

Exercice (QCM) : Cocher la/les réponse(s) correctes(s)

1. Java est un langage
 - a. compilé
 - b. interprété
 - c. compilé et interprété
2. Pour transformer un code lisible en code compréhensible par la machine, on utilise :
 - a. Un compilateur
 - b. Un exécuteur
 - c. Un débogueur
 - d. Un traducteur
3. À quoi peut-on comparer un IDE ?
 - a. À une base de données
 - b. À un outil
 - c. À un site web
4. Un JRE sert à :
 - a. Écrire des programmes Java
 - b. Faire fonctionner tous les IDE pour Java
 - c. Exécuter les programmes Java (en bytecode) sur notre machine.
5. Un retour à la ligne est un caractère.
 - a. vrai
 - b. faux
6. Quelle est la valeur de l'expression $39 \% 7$?
 - a. 4
 - b. 5
 - c. 5.6
 - d. 6
7. Supposons qu'on ait déclaré `char c = 'a'`; que vaudra l'expression `(int) c` ?
 - a. On ne peut pas le faire, c'est une narrowing conversion !
 - b. On ne peut pas convertir un type caractère en type entier
 - c. 0, tous les caractères de type char convertit en int valent 0
 - d. 97, l'unicode de la lettre a
8. Lequel de ces identificateurs est incorrect :
 - a. `nom_Prenom`
 - b. `_choix1`
 - c. `$prix$`
 - d. `2emeChoix`

9. Pour spécifier que la valeur d'une variable ne peut pas changer, on la déclare comme une constante avec le mot réservé :
 - a. finalize
 - b. const
 - c. define
 - d. final
10. De quel type primitif est le littéral 25.5F ?
 - a. double
 - b. float
 - c. long
 - d. short
11. Quelle est la valeur de l'expression $5 * 3 \geq 3 + 5 * 2$?
 - a. True
 - b. False
12. Quelle est la valeur de l'expression $17 \geq 8 * 2 \ \&\& \ ! (17 / 2 \geq 8)$?
 - a. True
 - b. False
13. Que vaut le résultat de l'affectation `int x = 10F / 8;` ?
 - a. 1.25
 - b. 1
 - c. Une erreur de compilation (type mismatch)
 - d. Une erreur d'exécution
14. Toute instruction `if` peut s'écrire comme une instruction `switch` équivalente :
 - a. Vrai
 - b. Faux
15. Toute instruction « `for` » peut s'écrire comme une instruction « `while` » équivalente :
 - a. Vrai
 - b. Faux
16. Que va produire le code ci-dessous ?


```
for ( ; ; ) {}
```

 - a. Une erreur de compilation
 - b. Une erreur d'exécution
 - c. Une boucle infinie
 - d. Rien du tout, c'est comme-ci on n'avait rien mis
17. Que va afficher le code suivant à la console ?


```
int x = 7 ;
switch (x)
{
    case 6:
    case 7:
    case 8:
        System.out.print ("A");
```

```

        case 9:
        case 10:
            System.out.print ("B"); break;
        default:
            System.out.print ("C");
    }

```

- a. A
- b. B
- c. AB
- d. ABC
- e. Rien du tout, le code ne compile pas !

18. Que va afficher cette instruction à la console ?

```
System.out.println ("Nbre d'enfants : "+4+3);
```

- a. Nbre d'enfants : 4
- b. Nbre d'enfants : 7
- c. Nbre d'enfants : 43
- d. Erreur

19. Soit la déclaration suivante :

```
int a = (int)(7.5/3);
```

- a. a = 2.5
- b. a = 2
- c. a = 3
- d. Une erreur sera signalée lors de la compilation

20. Soit le code suivant :

```
int n1 = 3, n2 = 2 ;
double r = (double) (n1/n2) ;
```

- a. r = 1.0
- b. r = 1.5
- c. Une erreur sera signalée lors de la compilation

21. Soit la déclaration suivante :

```
int a = (int)(Math.round(4.5));
```

- a. a = 4
- b. a = 5
- c. Une erreur sera déclarée lors de la compilation

22. Que manque-t-il pour que cette déclaration de méthode compile ?

```

public somme (int a, int b){
    return a + b;
}

```

- a. Rien du tout, c'est correct
- b. Il manque le type de retour
- c. On ne peut pas renvoyer directement a+b, il faut créer une variable locale, y placer le résultat de a+b puis la renvoyer
- d. Il manque le modificateur private.

Exercices d'application

Exercice 1

Ecrire un programme qui permet de résoudre une équation du premier degré de la forme $ax+b=0$ (discuter les différents cas).

Exercice 2

Ecrire un programme qui permet de résoudre une équation du second degré de la forme $ax^2+bx+c=0$ (on suppose toujours que $a \neq 0$).

Pour tester votre programme, considérer les cas suivants :

1. $a = 1$ $b = 2$ $c = 2$
2. $a = 1$ $b = 2$ $c = 1$
3. $a = 4$ $b = 5$ $c = 1$

Exercice 3

Ecrire un programme qui calcule le plus grand commun diviseur (PGCD) de deux entiers a et b lus au clavier.

On vous rappelle que :

- $\text{PGCD}(a,a) = a$
- $\text{PGCD}(a,b) = \text{PGCD}(a - b, b)$ si $a > b$
- $\text{PGCD}(a,b) = \text{PGCD}(a, b-a)$ si $b > a$.

Exemple : $\text{PGCD}(36,21) = \text{PGCD}(15,21) = \text{PGCD}(15,6) = \text{PGCD}(9,6) = \text{PGCD}(3,6) = \text{PGCD}(3,3)=3$

Exercice 4

Ecrire un programme qui permet de lire deux entiers a et b puis calcule et affiche a^b (discuter les différents cas possibles).

Exercice 5

1. Ecrire un programme qui permet de convertir une température du degré Celsius vers le Fahrenheit en utilisant la formule :

$$F = C * 9/5 + 32$$

2. Modifier votre programme pour donner à l'utilisateur la possibilité de choisir le sens de conversion ($^{\circ}\text{C} \rightarrow \text{F}$ ou $\text{F} \rightarrow ^{\circ}\text{C}$)
3. Modifier le programme pour qu'il donne à l'utilisateur la possibilité de faire autant de conversions qu'il le désire. Vous pouvez afficher un petit menu avec 3 options :

1. Conversion $^{\circ}\text{C}$ vers F
2. Conversion F vers $^{\circ}\text{C}$
3. Fin

Leçon N°2

Les tableaux en JAVA

I. Les tableaux simples

I.1. Définition

Un tableau est une structure de données contenant un groupe d'éléments tous du même type. Le type des éléments peut être un type primitif ou une classe.

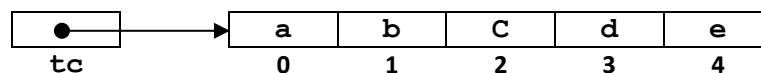
En Java, le type tableau est assimilable à une classe et un tableau est assimilé à un objet de cette classe.

- Pour déclarer un tableau et allouer l'espace nécessaire aux éléments, il faut utiliser l'opérateur **new** comme dans les exemples suivants :

```
int ti[] = new int[10];    // ti peut contenir 10 entiers
char tc[] = new char[5];  // tc peut contenir 5 caractères
String ts[] = new String[20]; // tableau d'objets.
```

- Une variable de type tableau ne représente pas l'objet en lui-même mais une référence vers l'objet (contient son adresse mémoire).
- Un tableau peut être initialisé comme suit :

```
char tc[] = {'a', 'b', 'c', 'd', 'e'};
```



- Les éléments d'un tableau sont toujours indicés à partir de 0.

Remarques

1. Lors de la définition d'un tableau, les crochets [] peuvent être placés avant ou après le nom du tableau :

```
int ti[];  ⇔  int[] ti;
```

2. Pour les tableaux dont le type de données est primitif, chaque élément est initialisé par la valeur nulle du type (0 pour les entiers, 0.0 pour les réels, false pour les booléens, '\u0000' pour les caractères). Pour les tableaux dont les éléments sont des objets, chaque élément est initialisé à null.
3. L'utilisation d'un tableau pour lequel l'espace mémoire n'a pas été alloué avec **new** provoque la levée d'une exception *NullPointerException*.

I.2. Accès aux éléments d'un tableau

- Chaque élément d'un tableau peut être accédé individuellement en fournissant le nom du tableau suivi de l'indice entre crochets.

Exemples

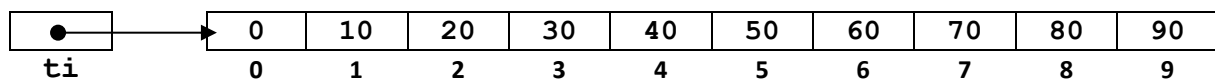
```
tc[0] = 'A';           // premier élément du tableau tc
tc[1] = '\u0042';      // 'B'
...
tc[4] = 'N';
```

- Un tableau possède un attribut **length** qui permet de connaître la taille de ce tableau.
 - ti.length vaut 10
 - tc.length vaut 5.

Cet attribut est très utile pour le parcours du tableau.

Exemple

```
int ti[] = new int[10];
for (int i = 0 ; i < ti.length ; i++)
    ti[i] = i *10;
```



Remarques

- La boucle suivante montre une forme plus simple de parcours d'un tableau en Java (structure *for each*):

```
for (int e : ti)
    System.out.print(e+"\t");
```

- La tentative d'accès à un élément en dehors des bornes du tableau provoque la levée d'une exception *ArrayIndexOutOfBoundsException*.

Exemple : Que va afficher le programme suivant :

```
public class Tablo {
    public static void main(String[] args){
        int t[] = {10, 20, 30, 40, 50};
        for (int i = 5 ; i >= 0 ; i--)
            System.out.print(t[i]+"\\t");
    }
}
```

Exercice 1

1. Ecrire un programme en Java qui permet de :
 - Lire les notes de 30 étudiants et les ranger dans un tableau tabNotes. Les notes doivent obligatoirement appartenir à l'intervalle [0 .. 20].
 - Calculer et afficher la moyenne des notes (m)
 - Calculer et afficher le nombre d'étudiants qui ont obtenu une note supérieure à la moyenne m.
2. Modifier le programme pour qu'il affiche également la meilleure et la mauvaise note.

NB : Pour tester le programme, on peut se limiter à 5 étudiants.

Exercice 2

Soit t un tableau qui peut contenir 5 éléments de type entier (int).

Ecrire un programme affiche à chaque fois un menu avec 5 options :

1. Remplissage du tableau
2. Tri dans le sens croissant
3. Tri dans le sens décroissant
4. Affichage du tableau
5. Recherche d'un élément
6. Fin

Remarque : vous pouvez expérimenter plusieurs méthodes de tri et de recherche.

Rappel : La méthode de *tri à bulles* nécessite deux étapes

1. Parcourir les éléments du tableau de gauche à droite (on s'arrête à l'avant dernier élément) ; si l'élément d'indice i et l'élément d'indice (i+1) ne sont pas ordonnés, alors on les permute.
2. Si au moins une permutation a été réalisée pendant le dernier parcours, revenir à l'étape 1 et recommencer un nouveau parcours ; sinon on s'arrête.

II. Les tableaux à plusieurs dimensions

- Lors d'une définition de tableau, le nombre de crochets indique le nombre de dimensions du tableau.

A titre d'exemple, la déclaration :

```
int m[][] = new int[5][10] ;
```

définit une matrice de 5 lignes et 10 colonnes. Les éléments de cette matrice seront indicés de `m[0][0]` à `m[4][9]`.

m	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										

- Un tableau à plusieurs dimensions peut être également initialisé.

Exemple

```
int m1[][] = {{1, 2, 3},{4, 5, 6}};
System.out.println(m1[1][2]) ;           // affiche la valeur ...
```

- En Java, les lignes d'un tableau à deux dimensions peuvent ne pas avoir le même nombre de colonnes.

Exemple

m2	0	1	2
0	1	2	3
1	4	5	

```
int m2[][] = {{1, 2, 3},{4, 5}};
System.out.println(m2[1][1]);           // affiche ...
System.out.println(m2.length);           // affiche ...
System.out.println(m2[0].length);        // affiche ...
System.out.println(m2[1].length);        // affiche ...
```

III. Les tableaux dynamiques (La classe Vector)

La classe **Vector** du package **java.util** permet de stocker des objets dans un tableau dynamique dont la taille évolue avec les besoins.

III.1. Constructeurs

La création d'un objet en général se fait par l'appel à une méthode (fonction) spéciale appelée *constructeur* qui porte le même nom que la classe. Parmi les constructeurs de la classe **Vector**, l'un n'attend aucun paramètre, l'autre attend une taille initiale. On construira donc un objet **vect** de la classe **Vector** en écrivant simplement:

- `Vector vect = new Vector();` // capacité initiale égale à 10

ou

- `Vector vect = new Vector(n);` // capacité initiale égale à n

III.2. Principales méthodes

La classe « **Vector** » contient de nombreuses méthodes qui rendent son utilisation très aisée :

Méthode	Rôle
<code>boolean isEmpty()</code>	vérifier si le vecteur est vide
<code>int size()</code>	retourner le nombre actuel d'éléments dans le vecteur
<code>boolean add(Object o)</code>	ajouter l'objet o à la fin du vecteur
<code>void addElement(Object o)</code>	
<code>void insertElementAt(Object o, int index)</code>	insérer l'objet o à la position indiquée. Les éléments situés à la suite sont décalés
<code>Object get(int index)</code>	retourner l'élément à la position fournie en paramètre
<code>Object elementAt(int index)</code>	
<code>Object firstElement()</code>	retourner le premier élément du vecteur
<code>Object lastElement()</code>	retourner le dernier élément du vecteur
<code>Object set(int index, Object o)</code>	remplacer l'élément à la position fournie en paramètre
<code>Object setElementAt(Object o, int index)</code>	remplacer l'élément à la position fournie en paramètre
<code>boolean remove(Object o)</code>	supprimer la première occurrence de l'objet indiqué
<code>boolean removeElement(Object o)</code>	
<code>Object remove(int index)</code>	supprimer l'élément à la position fournie en paramètre
<code>boolean removeElementAt(int index)</code>	
<code>Void clear()</code>	vider le vecteur
<code>boolean removeAllElements()</code>	
<code>boolean contains(Object o)</code>	vérifier si le vecteur contient l'objet o
<code>int indexOf(Object o)</code>	retourner la première position dans le vecteur de l'élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
<code>int lastIndexOf(Object o)</code>	retourner la dernière position dans le vecteur de l'élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé

Exemple

```

import java.util.Vector;
public class Vect {
    public static void main(String[] args){
        Vector vect = new Vector();
        vect.addElement(99);
        vect.addElement("mardi");
        vect.addElement(true);
        vect.setElementAt("mercredi",1);
        System.out.println("Capacity = "+ vect.capacity());
        System.out.println("Size = "+ vect.size());
        for (int i = 0; i<vect.size();i++)
            System.out.print(vect.get(i)+"\t");
    }
}

```

Output :

```

Capacity = 10
Size = 3
99    mercredi    true

```

Remarques

1. Il est possible d'utiliser la structure « for each » pour parcourir un vecteur comme dans l'exemple suivant :

```

for (Object e:vect)
    System.out.print(e+"\t");

```

2. Par défaut, un vecteur contient des éléments de type « Object » ; mais il est possible de spécifier un autre type lors de la déclaration. A titre d'exemple, voici comment déclarer un vecteur dont les éléments sont des chaînes de caractères :

```

Vector<String> ve = new Vector();

```

Exercice 1 (QCM)

1. La déclaration suivante :

```
int t[5] ;
```

- a. définit un tableau de 5 entiers indicés de 1 à 5
- b. définit un tableau de 5 entiers indicés de 0 à 4
- c. définit un tableau de 6 entiers indicés de 0 à 5
- d. n'est pas syntaxiquement correcte

2. La déclaration suivante :

```
int t1[], n, t2[] ;
```

Permet de déclarer

- a. 1 tableau
- b. 2 tableaux
- c. 3 tableaux

3. La déclaration suivante :

```
int[] t1[], n, t2[] ;
```

Permet de déclarer

- a. 1 tableau
- b. 2 tableaux
- c. 3 tableaux

4. Que va afficher le fragment de code suivant :

```
public static void main(String args) {
    int[] tab = {1, 2, 3};
    for (int i : tab)
        System.out.print(i);
}
```

- a. 1
- b. 123
- c. Erreur

5. Que va afficher le fragment de code suivant :

```
public static void main(String[] args) {
    int[] notes = new int[] {1, 3, 5};
    System.out.print(notes.length());
}
```

- a. 2
- b. 3
- c. Erreur

6. Laquelle des instructions suivantes va générer une erreur de compilation ?
- `int[] scores = {3, 5, 7};`
 - `int [][] scores = {2,7,6}, {9,3,45};`
 - `String cats[] = {"Fluffy", "Spot", "Zeus"};`
 - `boolean results[] = new boolean [] {true, false, true};`
 - `Integer results[] = {new Integer(3), new Integer(5), new Integer(8)};`
7. `v` étant un vecteur, les appels `v.firstElement()` et `v.elementAt(1)` sont équivalents.
- vrai
 - faux
8. `v` étant un vecteur, Les méthodes « `v.capacity()` » et « `v.size()` » sont équivalentes.
- vrai
 - faux
9. La méthode « `remove` » de la classe `Vector` supprime un élément. La case où se trouvait l'élément enlevé reste vide.
- vrai
 - faux
10. Le code suivant compile sans erreur
- ```
Vector<String> v1 = new Vector<String>();
Vector<String> v2 = new Vector();
Vector v3 = new Vector<String>();
```
- vrai
  - faux

## Exercice 2

- Créer une classe « `NbresPremiers` » réduite à une méthode `main()` permettant de :
  - Créer un vecteur « `Premiers` » et y ranger les entiers entre 1 et 100.
  - Parcourir le vecteur et supprimer les multiples de 2 (sauf 2)
  - Parcourir le vecteur et supprimer les multiples de 3 (sauf 3)
  - ...
  - Parcourir le vecteur et supprimer les multiples de 10 (10 est la racine carrée de 100)
  - Afficher les nombres restants dans le vecteur (les nombres premiers).
- Modifier le programme pour qu'il affiche les nombres premiers inférieurs à un nombre quelconque `n` dont la valeur est lue à partir du clavier.



## Leçon N°3

# Fondements de la Programmation Orientée Objet en JAVA

## I. Principe de base

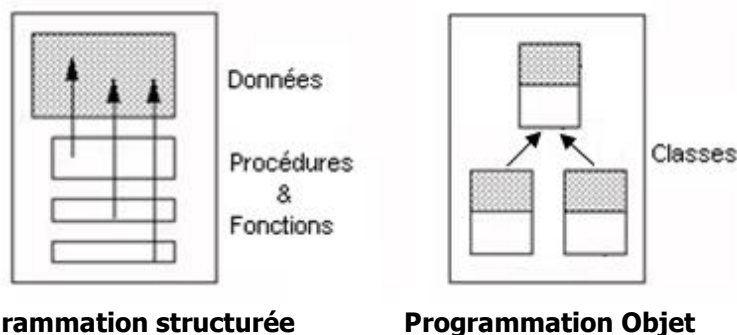
---

Les langages *structurés* ou *procéduraux* comme C et Pascal permettent de définir des données grâce à des variables et des traitements sur ces données grâce à des procédures et des fonctions.

La programmation orientée objet (POO) est un paradigme ou approche de programmation qui consiste à structurer le code source en un ensemble de *classes*.

Une **classe** est un type abstrait ou un modèle de spécification qui décrit un ensemble d'objets similaires. Une classe regroupe :

- des données membres, appelées *attributs*
- des fonctions membres permettant de manipuler ces données et appelées *méthodes*



A titre d'exemple, la classe « Voiture » peut être représentée par le schéma suivant :

| Voiture                  |  |
|--------------------------|--|
| Attributs                |  |
| - Modèle                 |  |
| - Année                  |  |
| - Couleur                |  |
| Méthodes                 |  |
| - Démarrer( )            |  |
| - Arrêter( )             |  |
| - Accélérer(int vitesse) |  |

A partir d'une classe, on peut créer (instancier) plusieurs **objets** qui possèdent la même structure (les mêmes *attributs*) et le même comportement (les mêmes *méthodes*). L'objet est donc la matérialisation concrète d'une classe (tout comme la variable qui est une matérialisation concrète d'un type et la maison qui est la matérialisation concrète d'un plan).

A titre d'exemple, l'objet « maVoiture » qui est une *instance* de la classe « Voiture » peut être représenté comme suit :

| maVoiture      |
|----------------|
| Modèle : Clio  |
| Année : 2018   |
| Couleur : Gris |

Le regroupement des attributs et des méthodes au sein d'une même entité s'appelle **encapsulation**. La règle *d'encapsulation* préconise que tous les attributs d'une classe soient cachés (*privés*). L'utilisateur ne doit pouvoir accéder qu'aux fonctions membres ou méthodes *publiques* qui lisent et modifient les attributs des objets de cette classe pour les faire évoluer.

Parmi les langages de programmation orientés objet on peut citer : C++, C#, Java, Scala, Python, Smalltalk, Visual Basic, Delphi, PHP, etc.

### Remarques

1. Une classe peut être comparée à un moule ou une matrice à partir de laquelle seront fabriqués des objets réels qui s'appellent des instances de la classe considérée.



2. Avant la création des classes et des objets, il faut passer par une étape de réflexion dans laquelle il faut ressortir la liste des classes à définir ainsi que les attributs et les méthodes de chaque classe. Il est important de savoir qu'un langage spécial appelé **UML** (Unified Modeling Language) a été conçu pour modéliser les classes avant de commencer à les coder.

## II. Les avantages de la POO

---

La POO consiste à concevoir un programme dans l'espace plutôt que dans le temps. Il s'agit de penser en termes d'objets, capables d'interagir entre eux, et pas en termes de lignes de codes qui s'enchainent les unes après les autres.

La POO offre de nombreux avantages, parmi lesquels on peut citer :

- l'accroissement de la stabilité des programmes
- la simplification de la maintenance
- la réutilisabilité du code source grâce au mécanisme d'**héritage**
- l'amélioration de la productivité des programmeurs
- l'encouragement du travail collaboratif.

**Exercice (QCM)**

1. Qu'est-ce qu'une méthode ?
  - a. Une variable qui décrit un objet
  - b. Une fonction qui décrit un objet
  - c. Un type d'objet particulier
2. Quel est le rapport entre un objet et une classe ?
  - a. Une classe est une instance d'objet
  - b. Un objet est une instance de classe
  - c. Il n'y a aucun rapport
3. Que dit la règle de l'encapsulation ?
  - a. Toutes les méthodes d'une classe doivent être privées
  - b. Tous les attributs d'une classe doivent être publics
  - c. Tous les attributs d'une classe doivent être privés
4. Qu'est-ce que UML ?
  - a. Un langage de programmation orienté objet
  - b. Une méthode de conception orientée objet
  - c. Un langage de modélisation objet basé sur des diagrammes.

**III. Définition d'une classe**

---

Pour pouvoir créer et manipuler des objets, il faut d'abord définir des classes, c'est-à-dire définir la structure et le comportement communs de ces objets. En Java, la définition d'une classe se fait de la manière suivante :

```
[niveau de visibilité] class Nom_Classe {
 // définition des données membres (attributs) de la classe
 // définition des fonctions membres (méthodes) de la classe
}
```

**Remarques**

1. Le niveau de visibilité d'une classe peut être **public** (visible par tout) ou **par défaut** (*package friendly*) c'est-à-dire visible uniquement au niveau de son package.
2. Une classe **publique** doit être obligatoirement codée dans un fichier **.java** qui porte le même nom que la classe. Ce fichier peut contenir d'autres classes (non publiques).
3. Par convention le nom d'une classe commence toujours par une majuscule et respecte la notation Camel Case.

**III.1. Déclaration des données membres (attributs)**

---

Les données membres sont des variables stockées au sein d'une classe. Elles doivent être précédées de leur type et (éventuellement) d'une étiquette précisant leur portée, c'est-à-dire les classes ayant le droit d'y accéder :

```
[niveau de visibilité] <type> nomAttribut [= expression] ;
```

Les principaux niveaux de visibilité sont :

- **private** : seules les méthodes de la classe dans laquelle l'attribut est défini peuvent y accéder et le modifier. Il s'agit du niveau de protection des données le plus élevé.
- **public** : toutes les méthodes peuvent accéder à l'attribut en question. Il s'agit du plus bas niveau de protection des données.

Lorsqu'aucun niveau de visibilité n'est indiqué, java applique le niveau par défaut qui est **package friendly** ; ce niveau autorise l'accès aux méthodes de la classe courante et aux méthodes des classes contenus dans le même package.

Il existe un autre niveau de visibilité (**protected**) mais qui n'est utile que dans le cas d'un *héritage*.

Exemple : La définition d'une classe « Voiture » comportant les quatre données membres (modèle, année, couleur et prix) peut se faire de la façon suivante :

```
public class Voiture {
 private String modèle;
 private int année;
 private String couleur;
 private double prix;
}
```

Remarque : **String** n'est pas un type primitif mais une classe prédéfinie sous java permettant la gestion des chaînes de caractères.

### III.2. Déclaration des fonctions membres (méthodes)

La déclaration d'une méthode se fait selon la syntaxe suivante :

```
[niveau de visibilité] <type_de_retour> nom_Méthode(type1 arg1, type2 arg2, ...) {
 // liste d'instructions
}
```

- « type\_de\_retour » représente le type de valeur que la méthode va retourner, cela peut-être un type primitif, une classe ou le mot-clé **void** si la méthode ne retourne aucune valeur.
- Une méthode qui est censée retourner une valeur doit se terminer par une instruction **return**. Une méthode peut contenir plusieurs instructions return, ce sera toutefois la première instruction return rencontrée qui provoquera la fin de l'exécution de la méthode et le renvoi de la valeur qui la suit à la méthode appelante. La syntaxe de l'instruction return est la suivante :

```
return expression;
```

- Le nom de la méthode suit les mêmes règles que les noms de variables :
  - un nom peut comporter des chiffres, mais pas pour le premier caractère
  - les caractères spéciaux ' \_ ' et ' \$ ' peuvent être utilisés mais ils sont déconseillés (à éviter)
  - le nom de la méthode est sensible à la casse (Java différencie entre minuscule et majuscule)
  - par convention, le nom de la méthode commence par une minuscule et respecte la notation Camel Case (exemples : calculSalaire( ), toString( ), ...)

- Les arguments sont facultatifs, mais les parenthèses doivent être présentes même s'il n'y a pas d'arguments.
- Il ne faut pas oublier de refermer les parenthèses et les accolades. Le nombre d'accolades ouvrantes doit être égal au nombre d'accolades fermantes. La même chose s'applique pour les parenthèses, les crochets ainsi que les guillemets.
- Pour qu'un programme Java puisse être exécuté, il faut que la classe principale contienne une méthode **main()**. Cette méthode se déclare toujours de la manière suivante :

```
public static void main(String[] args){
 //instructions à exécuter
}
```

Remarque : En Java, le passage des paramètres se fait ***par valeur*** pour les arguments de type simple et ***par référence*** pour les objets (y compris les tableaux).

### Exercice

Définir la classe Cercle qui comporte un seul attribut (le rayon exprimé en cm) et deux méthodes :

- **double** Perimetre() qui retourne le périmètre du cercle actuel
- **double** Surface() qui retourne la surface du cercle actuel.

### Solution

```
public Cercle {
 Rayon; // en cm

 double Perimetre(){
 2*Math.PI*Rayon;
 }

 {
 return Math.PI*Rayon*Rayon;
 }
}
```

---

## III.3. La surcharge de méthode

---

Un des apports les plus intéressants de l'approche objet, est la possibilité d'appeler plusieurs méthodes avec le même nom à condition que leurs arguments diffèrent (en nombre et/ou en type). Ce principe est appelé ***surcharge de méthode***. Il permet de donner le même nom à des méthodes comportant des paramètres différents et simplifie donc l'écriture de méthodes sémantiquement similaires sur des paramètres de types différents.

### Exemple

```
public int somme(int p1, int p2) {
 return p1+p2;
}
```

```

public int somme(int p1, int p2, int p3) {
 return p1+somme(p2,p3);
}

public double somme(double p1, double p2){
 return p1+p2;
}

```

**Exercice (QCM)**

1. Quand un membre d'une classe (attribut ou méthode) est déclaré avec l'étiquette *private*, cela signifie :

- a. Qu'on ne peut y accéder que depuis l'intérieur de la classe
- b. Qu'on ne peut y accéder que depuis l'extérieur de la classe
- c. Qu'on peut y accéder depuis l'intérieur et l'extérieur de la classe

2. Dans quel ordre doit-on placer les attributs et les méthodes dans une classe ?

- a. D'abord les attributs, après les méthodes
- b. D'abord les méthodes, après les attributs
- c. L'ordre n'a aucune importance

3. Quel est le niveau d'accès par défaut des attributs d'une classe ?

- a. private
- b. public
- c. protected
- d. package friendly

4. Toute classe doit comporter une méthode main( )

- a. Vrai
- b. Faux

5. Soient les deux méthodes calcul( ) définies comme suit :

```

public int calcul(int p1, int p2) {
 ...
}

public double calcul(int p1, int p2) {
 ...
}

```

- a. Ce type de surcharge est accepté par Java
- b. Ce type de surcharge est interdit par Java

## IV. Les constructeurs

---

Chaque classe doit définir une ou plusieurs méthodes particulières appelées **constructeurs**. Un **constructeur** est une méthode invoquée lors de la création d'un objet.

Cette méthode, qui peut être vide, effectue les opérations nécessaires à l'initialisation des attributs d'un objet. Chaque constructeur doit avoir le même nom que la classe où il est défini et n'a aucune valeur de retour (c'est l'objet créé qui est renvoyé).

Dans l'exemple suivant, le constructeur initialise le rayon du nouveau cercle avec la valeur fournie en argument :

```
public class Cercle {
 private double rayon ;

 // définition d'un constructeur
 public Cercle(double r) {
 rayon = r;
 }
 ...
}
```

Plusieurs constructeurs peuvent être définis s'ils acceptent des paramètres d'entrée différents (surcharge).

**Remarque :** Si dans une classe, aucun constructeur n'est défini, le compilateur Java crée un constructeur par défaut (sans arguments). Mais, dès qu'un constructeur est défini, le constructeur par défaut disparaît.

## V. Les accesseurs et les mutateurs

---

L'un des aspects les plus essentiels du paradigme « orienté objet » est l'**encapsulation**, qui consiste à rassembler les données et les traitements au sein d'une même structure (classe) et définir des étiquettes pour les données et les fonctions membres afin de préciser si celles-ci sont accessibles à partir d'autres classes ou non.

En Java, des données membres portant l'étiquette *private* ne peuvent pas être manipulées directement par les fonctions membres des autres classes. Ainsi, pour pouvoir manipuler ces données membres, le créateur de la classe doit prévoir des fonctions membres spéciales portant l'étiquette *public*.

Les fonctions membres permettant de récupérer la valeur d'un attribut protégé (accès en lecture) sont appelées **accesseurs** ou **getters**. Un accesseur doit avoir comme type de retour le type de la variable à renvoyer. Une convention de nommage veut que l'on fasse commencer de façon préférentielle le nom de l'accesseur par le préfixe *get* afin de faire ressortir sa fonction première.

Les fonctions membres permettant de modifier la valeur d'un attribut protégé (accès en écriture) sont appelées **mutateurs** ou **setters**. Un mutateur doit avoir comme paramètre la valeur à assigner à l'attribut membre et ne doit pas nécessairement renvoyer de valeur (il possède dans sa plus simple expression le type *void*). Une convention de nommage veut que l'on fasse commencer de façon préférentielle le nom du mutateur par le préfixe *set*.

**Exemple**

```

public class Personne {
 private int age;
 private double taille;
 private char genre; //M : Masculin et F : Féminin

 ...

 public int getAge(){ // définition d'un accesseur
 return age;
 }

 public void setAge(int a){ // définition d'un mutateur
 if (a > 0)
 age = a;
 else
 System.out.println("valeur non valide ... ");
 }
}

```

L'intérêt principal d'un tel mécanisme est le contrôle de la validité des données membres qu'il procure. En effet, il est conseillé de contrôler la valeur de l'argument avant de l'affecter à la donnée membre.

**Exercice 1 (QCM)**

1. Qu'est-ce qu'un accesseur ?
  - a. Une méthode qui permet de lire un attribut indirectement
  - b. Une méthode qui permet de modifier un attribut indirectement
  - c. Une méthode qui permet d'accéder à tous les éléments privés de la classe
  - d. Une méthode qui affiche tout le contenu de la classe
2. Qu'est-ce qu'un mutateur ?
  - a. Une méthode qui permet de lire un attribut indirectement
  - b. Une méthode qui permet de modifier un attribut indirectement
  - c. Une méthode qui permet d'accéder à tous les éléments privés de la classe
  - d. Une méthode qui affiche tout le contenu de la classe
3. Peut-on avoir un accesseur de type void ?
  - a. Oui
  - b. Non
4. Peut-on avoir un mutateur sans argument ?
  - a. Oui
  - b. Non
5. Peut-on surcharger un mutateur ?
  - a. Oui
  - b. Non



## VI. Les objets

### VI.1. Instanciation

Un objet est une instance (ou « exemplaire ») d'une classe qui est référencée par une variable et qui possède un état (valeurs des attributs). Pour créer un objet, il est nécessaire de déclarer une variable dont le type est la classe à instancier, puis de faire appel à un constructeur de cette classe en utilisant le mot clé **new** selon la syntaxe suivante :

```
nomClasse nomObjet = new Constructeur(arguments);
```

L'exemple ci-dessous illustre la création d'un objet C de classe Cercle :

```
Cercle C = new Cercle(10);
```

### VI.2. Accès aux attributs et aux méthodes

Pour accéder à un attribut d'un objet, il faut préciser le nom de l'objet qui le contient. Le symbole '.' sert à séparer l'identificateur de l'objet de l'identificateur de l'attribut.

Ainsi, une copie de la valeur du rayon du cercle C dans la variable temp s'écrit :

```
double temp = C.rayon ;
```

La même syntaxe est utilisée pour appeler une méthode d'un objet. Par exemple :

```
System.out.println("Surface du cercle = "+C.surface());
```



Pour qu'un tel appel soit possible, il faut que la méthode, au sein de laquelle est fait cet appel, ait le droit d'accéder à l'attribut ou à la méthode en question.

**Remarque :** Pour référencer l'objet « courant », il faut utiliser le mot-clé **this** comme dans l'exemple suivant :

```
public class Cercle {
 private double rayon ;
 ...

 // définition d'un mutateur
 void setRayon(double rayon) {
 this.rayon = rayon;
 }
}
```

Ici le mot clé « this » permet de lever l'ambiguïté en précisant qu'il s'agit d'affecter la valeur du paramètre rayon à l'attribut qui porte le même nom.

**Exercice :** Corriger les erreurs syntaxiques et sémantiques présentes dans le programme suivant :

```
public classe Etudiant {
 private int matricule;
 public String nom;
 Private double moy;
```

```

public void Etudiant(){
 matricule = 0;
 nom = "";
 moy = 0.0;
}

public etudiant (String nom, int matricule, double moy){
 matricule = matricule;
 nom = nom;
 moy = moy;
}

public float getMoy(){
 return moy;
}

public string toString() {
 return "Etudiant [matricule = " + matricule + ", nom = "
 + nom + ", moy= " + moy + "]\n";
}

public boolean equals(Etudiant e) {
 return (this.matricule == e.matricule) ;
}

public void main(String[] params){
 Etudiant e1 = new Etudiant(111, "Ali", 12.5);
 Etudiant e2 = new Etudiant(112, "Ali", 12.5);
 System.out.println(e1.toString());
 System.out.println(e1.equals(e2));
}
}

```

## VII. Les membres de classe (statiques)

Contrairement aux membres d'instance (attributs et méthodes) qui sont spécifiques à chaque objet de la classe, un **membre de classe** est un membre commun à toutes les instances de la classe et existe dès que la classe est définie en dehors et indépendamment de toute instantiation.


Les membres de classe sont déclarés à l'aide du mot-clé **static**.

L'accès à un membre de classe se fait en appliquant l'opérateur « . » sur le nom de la classe comme suit :

|                                                                                                                                                                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p style="text-align: center;"><b>NomClasse.nomAttribut</b></p> <p style="text-align: center;">ou</p> <p style="text-align: center;"><b>NomClasse.nomMéthode(paramètres)</b></p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

A l'intérieur de la classe, l'appel d'une méthode statique peut se faire sans avoir besoin d'un objet :

|                                                                  |
|------------------------------------------------------------------|
| <p style="text-align: center;"><b>nomMéthode(paramètres)</b></p> |
|------------------------------------------------------------------|

 Les méthodes statiques, étant indépendantes de toute instance, n'ont pas accès aux variables ou méthodes non statiques. De même, une méthode statique ne peut pas utiliser la référence this.

### Exemple

```

public class Personne {
 private static int nbPers = 0; // variable de classe initialisée à zéro
 private int age;
 private double taille;
 private char genre; // M : Masculin et F : Féminin

 public Personne(int age, double taille, char genre){ // constructeur
 nbPers++;
 this.age = age ;
 this.taille = taille ;
 this.genre = genre ;
 }

 public static void main(String[] args) {
 Personne Ali = new Personne(21, 1.80, 'M');
 Personne Alia = new Personne(23, 1.70, 'F');
 System.out.println("Nombre de personnes :"+Personne.nbPers);
 }
}

```

### Output du programme

```
Nombre de personnes :2
```

## VIII. Le mot clé final

Le mot-clé **final** est un modificateur qui peut s'appliquer à une variable, à une classe ou à une méthode :

- Dans la déclaration d'une variable, **final** indique que cette variable est constante. Une fois initialisée, sa valeur ne peut plus être modifiée.

### Exemple

```

public class Trigonométrie {
 final double PI = 3.14 ; // définition d'une constante
 ...
}

```

- Dans la déclaration d'une classe, **final** indique que celle-ci ne pourra pas être étendue (on ne peut pas en dériver des sous-classes).
- Dans la déclaration d'une méthode, **final** indique qu'elle ne peut pas être redéfinie dans une classe fille (sous-classe). Si la méthode est *static* ou *private*, elle est automatiquement de type **final**.
- On peut associer static à final pour définir un attribut de classe (commun à tous les objets de la classe) qui est constant.

## IX. Classe interne

---

Une classe interne (ou imbriquée) est une classe ayant un nom et définie au même niveau qu'un attribut ou une méthode de classe.

Une classe interne est visible uniquement par les méthodes de la classe dans laquelle elle est définie.

Une classe interne a accès à tous les membres (même privés) de la classe qui l'englobe.

### Exemple

```
public class Etudiant {
 public class Date { // classe interne
 int jour, mois, année;
 public Date(int j, int m, int a){
 jour = j; mois = m; année = a;
 }

 public String toString(){
 return jour + "/" + mois + "/" + année;
 }
 }

 private String nom;
 private String prénom;
 private Date dateNais;

 public Etudiant(String nom, String prénom, int jour, int mois, int
année) {
 this.nom = nom;
 this.prénom = prénom;
 this.dateNais = new Date(jour, mois, année);
 }

 public String toString() {
 return "Etudiant{" + "nom=" + nom + ", prénom=" + prénom + ",
dateNais=" + dateNais.toString() + '}';
 }

 public static void main(String[] args){
 Etudiant e = new Etudiant("Ben Amor", "Amor", 20, 10, 2000);
 System.out.println(e.toString());
 }
}
```

### Output du programme

```
Etudiant{nom=Ben Amor, prénom=Amor, dateNais=20/10/2000}
```

### Exercice (QCM)

1. Quel est le rôle du constructeur ?

- a. Nettoyer la mémoire
- b. Déclarer des variables
- c. Initialiser les attributs d'un objet

2. Qu'est-ce qu'un constructeur par défaut ?

- a. Un constructeur de type void
- b. Un constructeur sans paramètres ajouté automatiquement par le compilateur
- c. Un constructeur qui initialise chaque attribut à sa valeur par défaut

3. Peut-on surcharger un constructeur ?

- a. Oui
- b. Non

4. Que fait cette instruction Java ?

```
Date today = new Date();
```

- a. Déclaration d'une référence
- b. Création d'un objet
- c. Invocation d'une méthode
- d. Opération d'affectation

5. Quelle est la syntaxe correcte pour un appel de méthode qui renvoie un résultat ? (On suppose qu'on possède une variable `myString` de type `String`)

- a. `int longueur = length(myString);`
- b. `int longueur = myString.length;`
- c. `int longueur = myString.length();`
- d. `int longueur = myString::length;`

6. Pour connaître les paramètres à fournir pour créer un nouvel objet, il faut consulter ...

- a. Les constructeurs
- b. Les accesseurs
- c. Les méthodes
- d. Les packages

7. La déclaration suivante est faite dans une classe `Person` :

```
private int age;
```

Où l'attribut `age` sera-t-il visible ?

- a. Dans les méthodes de la classe `Person`
- b. Dans les méthodes des classes qui héritent de la classe `Person`
- c. Dans les méthodes des classes du même package que la classe `Person`
- d. Il sera visible partout

8. Lorsque plusieurs méthodes ont le même nom (surcharge), comment la machine virtuelle Java sait-elle laquelle on veut invoquer ?

- a. Elle les essaie toute une à une et prend la première qui fonctionne
- b. Elle ne devine pas, il faut lui spécifier lorsqu'on compile le code
- c. On indique le numéro de la méthode que l'on veut invoquer
- d. Elle se base sur le type des paramètres

## 9. Une classe peut contenir d'autres classes

- a. vrai
- b. faux

**Exercice :** Définir une classe **Point** qui possède le schéma suivant :

| Classe Point                                                                                                                                                                                                                                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| - x : entier // abscisse                                                                                                                                                                                                                     |
| - y : entier // ordonnée                                                                                                                                                                                                                     |
| - Point ( ) // constructeur qui initialise les coordonnées à (0,0)                                                                                                                                                                           |
| - Point(int abs, int ord) // 2 <sup>ème</sup> constructeur qui initialise les coordonnées à (abs,ord)                                                                                                                                        |
| - Deplacer(int dx, int dy) // déplace un point d'une distance horizontale dx et d'une distance verticale dy (les distances dx et dy peuvent être positives ou négatives)                                                                     |
| - toString() // retourne une chaine de caractères contenant les coordonnées actuelles du point sous la forme Point[x,y]                                                                                                                      |
| - public static void main(String[] args) : méthode principale dans laquelle on crée un point A de coordonnées (2,3), on le déplace de 2 cm sur l'horizontale et 1 cm sur la verticale puis on affiche les nouvelles coordonnées de ce point. |

## Exercices d'application

### Exercice 1

1. Corriger toutes les erreurs qui se trouvent dans le programme suivant :

```
public classe Employe {
 private int matricule;
 private string nom;
 private double salBase;
 private double prime;
 private double retenue;

 // définition d'un constructeur
 Public Employe (int matricule, String nom, double salBase, double
 prime, double retenue){
 matricule = matricule;
 nom = nom;
 salBase = salBase;
 prime = prime;
 retenue = retenue;
 }
 public void calculSalaireNet(){
 return salBase + prime - retenue;
 }

 Public void afficher(){
 system.out.println("Matricule = ",matricule);
 System.out.println("Nom = "+nom);
 System.out.println("Salaire Net = "+calculSalaireNet());
 }

 public static void main(String[] args) {
 Employe E = new Employe(2525, "Ali Ben Salah", 450, 150, 100);
 afficher();
 }
}
```

2. Que va afficher ce programme sur la console (après correction) ?

## Exercice 2

Définir une classe **Rectangle** qui comporte deux attributs (longueur et largeur) et cinq méthodes :

- Rectangle(double longueur, double largeur) : constructeur
- double périmètre( ) qui retourne le périmètre du rectangle actuel
- double surface( ) qui retourne la surface du rectangle actuel
- void afficher( ) qui affiche la longueur, la largeur, le périmètre et la surface du rectangle actuel.
- public static void main(String[] args) : méthode principale dans laquelle on crée et on affiche toutes les informations relatives à un rectangle R de longueur 5 cm et de largeur 4 cm.

Le résultat attendu doit être semblable à ceci :

```
longueur = 5.0
largeur = 4.0
Périmètre = 18.0 cm
Surface = 20.0 cm2
```

## Exercice 3

Définir la classe CompteBancaire qui comporte trois attributs (numCompte, nomClient, solde) et six méthodes :

- CompteBancaire(int numCompte, string nomClient, double solde) : constructeur de la classe
- void déposer(double montant) qui permet de mettre à jour le solde d'un compte suite à une opération de versement
- void retirer(double montant) qui permet de mettre à jour le solde d'un compte suite à une opération de retrait (si le solde actuel le permet)
- void consulter() qui affiche le numéro du compte, le nom du titulaire du compte et le solde actuel.
- void virer(double montant, CompteBancaire C) qui permet de transférer un montant donné du compte actuel vers un autre compte C (si le solde actuel le permet)
- public static void main(String[] args) qui permet de :
  - créer un compte A ayant les propriétés suivantes (Numéro : 1041 ; Titulaire du compte : «Ali Ben Salah » ; Solde initial : 450 D)
  - créer un compte B ayant les propriétés suivantes (Numéro : 1042 ; Titulaire du compte : «Mohamed Ben Salah » ; Solde initial : 300 D)
  - déposer 250 D dans le compte A
  - retirer 200 D du compte A
  - virer 100 D du compte A vers le compte B
  - consulter le compte A
  - consulter le compte B.

**Exercice 4**

1. Définir une classe **Adresse** qui comporte trois attributs (numéro, rue et ville) et 2 méthodes :

- Adresse (int numéro, String rue, String ville) : constructeur
- toString() qui renvoie l'adresse actuelle dans le format suivant :

Adresse : 17, rue Ibn Rochd, Sousse

2. Définir une classe **Client** qui comporte 4 attributs (numCli, nomCli, adrCli, telCli) et 2 méthodes :

- Client(int numCli, String nomCli, Adresse adrCli, int telCli) : constructeur
- toString() qui renvoie une description textuelle du client actuel dans le format suivant :

numCli : 1111  
nomCli : Ali Ben Salah  
Adresse : 17, rue Ibn Rochd, Sousse  
telCli : 41646605

3. Définir la classe **Compte** qui comporte trois attributs (numCompte, client, solde) et trois méthodes :

- Compte(int numCompte, Client client, double solde) : constructeur de la classe
- void consulter() qui affiche le numéro du compte, le numéro du client, son nom, son adresse, son numéro de téléphone ainsi que le solde actuel.
- public static void main(String[] args) qui permet de :

- créer deux objets ABS et KBA de type client et qui possèdent les caractéristiques suivantes :

| Client | numCli | nomCli         | adrCli                     | telCli   |
|--------|--------|----------------|----------------------------|----------|
| ABS    | 1111   | Ali Ben Salah  | 17, rue Ibn Rchod, Sousse  | 73483342 |
| KBA    | 2222   | Karim Ben Amor | 18, rue Ibn Sina, Monastir | 41646605 |

- créer les 3 comptes C1, C2 et C3 ayant les caractéristiques suivantes :

| Compte | Numéro | Titulaire | Solde Actuel |
|--------|--------|-----------|--------------|
| C1     | 1041   | ABS       | 450          |
| C2     | 1042   | KBA       | 500          |
| C3     | 1043   | KBA       | 600          |

- créer un vecteur vectComptes capable de contenir des objets de type « Compte » puis y ranger les comptes C1, C2 et C3 dans l'ordre.
- demander à l'utilisateur d'entrer un numéro de compte; si le compte existe en faire une consultation si non afficher un message d'erreur.

4. Ajouter à la classe Compte un compteur (attribut de classe) qui compte le nombre de comptes ouverts, puis afficher sa valeur juste avant la fin de la méthode main( ).



**Exercice 5**

1- Définir la classe **Complexe** qui comporte 2 attributs privés (**réel** et **imag**) qui représentent respectivement la partie réelle et la partie imaginaire et 6 méthodes :

- **Complexe()** : un premier constructeur qui initialise le nombre complexe à (0,0)
- **Complexe (double a, double b)** : un deuxième constructeur
- **double getRéel ()** : accesseur qui retourne la partie réelle
- **double getImag ()** : accesseur qui retourne la partie imaginaire
- **String toString()** qui convertit le nombre complexe actuel en une chaîne sous la forme **a+bi**
- **double module()** : retourne le module du nombre complexe actuel, sachant que le module d'un nombre complexe  $u = a + bi$  est calculé selon la formule :

$$|u| = \sqrt{a^2 + b^2}$$

2- Définir la classe **TestComplexe** qui comporte 3 méthodes :

- **static Complexe somme(Complexe u, Complexe v)** qui retourne la somme de 2 nombres complexes
- **static Complexe produit(Complexe u, Complexe v)** qui retourne le produit de 2 nombres complexes
- **public static void main(String[] args)** qui permet de :
  - créer un nombre complexe  $c1 = 2 + 2i$
  - créer un nombre complexe  $c2 = 1 + 3i$
  - calculer et afficher le module du nombre complexe  $c2$
  - calculer et afficher le nombre complexe  $c3 = c1 + c2$
  - calculer et afficher le nombre complexe  $c4 = c1 \times c2$

**Exercice 6 :** Une entreprise est composée de plusieurs services qui contrôlent chacun un certain nombre de projets. Pour simplifier, on suppose qu'un service ne peut pas contrôler plus de 5 projets en même temps.

1. Définir la classe « **Projet** » qui possède la structure suivante :

| Projet                                                                   |
|--------------------------------------------------------------------------|
| - nomProj<br>- budget                                                    |
| - Projet(nomProj, budget)<br>- double getBudget()<br>- String toString() |

La méthode `toString` doit retourner une chaîne de la forme `Projet{nomProj, budget}`.

2. Définir la classe « **Service** » qui possède la structure suivante :

| Service                                                                                                 |
|---------------------------------------------------------------------------------------------------------|
| - nomServ<br>- responsable<br>- nbProjets // initialisé à 0<br>- projets[]                              |
| - Service(nomServ, responsable)<br>- ajouterProjet(P)<br>- double totalBudgets()<br>- String toString() |

La méthode `totalBudgets()` retourne le cumul des budgets des projets sous le contrôle du service actuel.

La méthode `toString()` renvoie une chaîne de la forme `Service{nomServ, responsable, nbProjets, budgetTotal}`.

3. Définir une classe « TestProj » réduite à une méthode `main()` qui permet de :

- créer 5 projets identifiés de P1 à P5 dont les caractéristiques sont fournies dans le tableau suivant :

| Service                 | Projets en cours |                       |                           |
|-------------------------|------------------|-----------------------|---------------------------|
| <u>S1</u>               |                  |                       |                           |
| - Nom : Technique       | <u>P1</u>        | <b>Nom : Projet 1</b> | <b>Budget : 100 000 D</b> |
| - Responsable : Mohamed | <u>P2</u>        | <b>Nom : Projet 2</b> | <b>Budget : 200 000 D</b> |
| - Nbre de projets : 2   |                  |                       |                           |
| <u>S2</u>               | <u>P3</u>        | <b>Nom : Projet 3</b> | <b>Budget : 300 000 D</b> |
| - Nom : Production      | <u>P4</u>        | <b>Nom : Projet 4</b> | <b>Budget : 400 000 D</b> |
| - Responsable : Salah   | <u>P5</u>        | <b>Nom : Projet 5</b> | <b>Budget : 500 000 D</b> |
| - Nbre de projets : 3   |                  |                       |                           |

- créer les deux services S1 et S2 dont les caractéristiques sont fournies dans le tableau ci-dessus ;
- affecter à chaque service la liste des projets qu'il contrôle ;
- afficher une description détaillée de chaque service (nom, responsable, nombre de projets en cours, budget total de ses projets).

### Exercice 7

On désire écrire un programme en Java pour gérer l'emprunt des livres d'une bibliothèque par les étudiants.

Un étudiant est défini par les attributs privés suivants :

- matricule : Entier
- nom : Chaîne de caractères
- prénom : Chaîne de caractères
- adresse : Chaîne de caractères.

Un livre est défini par les attributs privés suivants :

- numLiv : Entier
- titre : Chaîne de caractères
- auteur : Chaîne de caractères
- disponible : Booléen ; ce champ prend la valeur « faux » si le livre est emprunté par un étudiant
- emprunteur : champ de type objet qui indique l'étudiant qui a emprunté le livre.

La bibliothèque sera représentée par une classe « Bibliothèque » ayant 3 attributs privés :

- nomBib : Chaîne de caractères
- nbLiv : Entier (nombre total de livres possédés par la bibliothèque)
- tabLiv[ ] : Tableau de livres (on suppose que le nombre total de livres ne dépasse pas 1000).

1. Définir la classe « Etudiant » qui doit comporter, en plus des attributs privés mentionnés ci-dessus :

- un constructeur **Etudiant(matricule, nom, prénom)**
- un constructeur **Etudiant(matricule, nom, prénom, adresse)**
- une méthode **toString()** qui retourne une chaîne décrivant un étudiant sous la forme « l'étudiant Ali Ben Salah de matricule 1011 qui habite à Sousse »

2. Définir la classe « Livre » qui doit comporter, en plus des attributs privés mentionnés ci-dessus :

- un constructeur **Livre(numLiv, titre, auteur)**. On suppose par défaut que le livre est disponible et de ce fait l'emprunteur sera initialisé à « null ».
- une méthode **toString()** qui retourne une description textuelle du livre sous l'une des formes suivantes :

Le livre numéro 4921 ayant pour titre « Penser en Java » écrit par « B. Eckel » est disponible

ou

Le livre numéro 4921 ayant pour titre « Penser en Java » écrit par « B. Eckel » est emprunté par l'étudiant Ali Ben Salah de matricule 1011 qui habite à Sousse

- un accesseur **getNumLiv()**
- un accesseur **isDisponible()**
- une méthode **void emprunter(Etudiant e)** qui enregistre l'opération d'emprunt du livre actuel à l'étudiant e
- une méthode **void retourner()** qui actualise l'état d'un livre suite à un retour.

3. Définir la classe « Bibliothèque » qui doit comporter, en plus des attributs privés mentionnés ci-dessus :

- un constructeur **Bibliothèque(nom)**. On suppose qu'initialement la bibliothèque ne contient aucun livre.
- une méthode **ajouterLivre(liv)** qui permet d'ajouter un livre à la bibliothèque
- Un accesseur **getLivres()** qui retourne le tableau des livres
- une méthode **rechercherLivre(num)** qui permet de rechercher un livre dont on connaît le numéro. Si le livre est trouvé, la méthode affiche une description complète du livre avec notamment son état actuel (disponible ou emprunté par un étudiant) sinon elle affiche le message « Livre introuvable ... »
- une méthode **compter()** qui renvoie le nombre total de livres actuellement disponibles à la bibliothèque.

4. Définir une classe « TestBib » réduite à une méthode **main()** et qui permet de :

- Créer les 2 étudiants suivants :

|                                |                        |                       |                         |
|--------------------------------|------------------------|-----------------------|-------------------------|
| ▪ E1 : <u>Matricule</u> : 1011 | <u>Nom</u> : Ben Salah | <u>Prénom</u> : Ali   | <u>Adresse</u> : Sousse |
| ▪ E2 : <u>Matricule</u> : 1022 | <u>Nom</u> : Ben Amor  | <u>Prénom</u> : Walid |                         |

- Créer les 3 livres suivants :

|                             |                                      |                                 |                 |
|-----------------------------|--------------------------------------|---------------------------------|-----------------|
| ▪ L1 : <u>Numéro</u> : 4921 | <u>Titre</u> : « Penser en Java »    | <u>Auteur</u> : « B. Eckel »    | Disponible      |
| ▪ L2 : <u>Numéro</u> : 4922 | <u>Titre</u> : « BD Relationnelles » | <u>Auteur</u> : « G. Gardarin » | Emprunté par E1 |
| ▪ L3 : <u>Numéro</u> : 4923 | <u>Titre</u> : « UML2 »              | <u>Auteur</u> : « C. Soutou »   | Emprunté par E2 |

- Créer une bibliothèque B de nom « Almaarif »

- Ajouter les 3 livres L1, L2 et L3 à la bibliothèque B
- Enregistrer les deux opérations d'emprunt
- Enregistrer le retour du livre L3
- Afficher le nombre total de livres actuellement disponibles à la bibliothèque
- Rechercher le livre dont le numéro est 4922.

### Exercice 8

Définir la classe « Date » qui permet de représenter une date dans le format « JJ/MM/AAAA ». Cette classe doit contenir 3 attributs de type entier (j, m, a) et 6 méthodes :

- **Date(int j, int m, int a)** : constructeur de classe
- **int nbJours()** qui retourne le nombre de jours du mois courant. On rappelle que selon le calendrier grégorien, une année a est bissextile (le mois de février contient 29 jours) si l'une des conditions suivantes est vraie :
  - a est divisible par 4 et non divisible par 100
  - a est divisible par 400
- **boolean valide()** qui permet de vérifier si la date « courante » est valide ou non. On suppose qu'une date valide remplit les 3 conditions suivantes :
  - 1 <= jour < nbJours()
  - 1 <= mois <= 12
  - année > 0
- **Date lendemain()** : qui retourne la date qui correspond au lendemain de la date « courante »
- **toString()** : qui retourne la date sous la forme textuelle « JJ/MM/AAAA »
- **public static void main(String[] args)** qui permet de saisir le jour, le mois et l'année d'une date à partir du clavier. Si la date entrée est valide elle l'affiche dans le format JJ/MM/AAAA ainsi que la date du lendemain ; sinon elle affiche un message d'erreur.

### Exercice 9

1- Définir la classe **Time** qui comporte 3 attributs privés :

- heure : 0 .. 23
- minutes : 0 .. 59
- secondes : 0 .. 59.

La classe Time doit disposer des méthodes suivantes :

- **Time()**: constructeur qui initialise le temps à 00 :00 :00
- **Time(hr, min, sec)** : un deuxième constructeur
- **int getHeure()** : accesseur
- **int getMinutes()** : accesseur
- **int getSecondes()** : accesseur
- **void setHeure(int hr)** : mutateur
- **void setMinutes(int min)** : mutateur
- **void setSecondes(int sec)** : mutateur
- **void ajouterHeures(hr)**
- **void ajouterMinutes(mn)**
- **void ajouterSecondes(Sec)**
- **String toString()** qui retourne le temps actuel dans le format « hh:mm:ss »

2- Définir la classe **TestTime** qui comporte 2 méthodes :

- **static Time somme(Time t1, Time t2)** qui retourne la somme de 2 temps
- **public static void main(String[] args)** qui permet de :
  - créer un temps t1 = 22h 58 mn 50 sec
  - créer un temps t2 = 1mn 2 sec
  - régler les secondes de t1 à 57
  - ajouter 1 heure à t1
  - calculer t3 = t1 + t2
  - ajouter 1 seconde à t3
  - afficher t3 dans le format « hh:mm:ss ».

### Exercice 10

1. Ecrire une classe java permettant de représenter des nombres rationnels. Un rationnel est caractérisé par son numérateur et son dénominateur. On vous demande de définir une classe nommée **Rationnel** avec :
  - a. Deux constructeurs :
    - Le premier **Rationnel()** initialise le numérateur et le dénominateur à 1
    - Le second **Rationnel(int p, int q)**.
  - b. Deux méthodes d'accès aux attributs nommées respectivement **getNumumérateur()** et **getDénominateur()**
  - c. Une méthode nommée **addition(Rationnel r)** permettant de faire la somme du rationnel courant avec le rationnel passé en paramètre (ex :  $(1/2) + (1/3) = 5/6$ ). On accepte les résultats même sans simplification.
  - d. Une méthode nommée **produit(Rationnel r)** permettant de faire le produit du rationnel courant par le rationnel passé en paramètre (ex :  $(5/2) * (1/3) = 5/6$ ).
  - e. Une méthode nommée **division(Rationnel r)** permettant de faire la division du rationnel courant par le rationnel passé en paramètre (ex :  $(5/2) / (1/3) = 15/2$ ).
  - f. Une méthode nommée **inverse()** permettant d'inverser le rationnel courant.
  - g. Une méthode nommée **toString()** qui retourne une chaîne de caractères de la forme « numérateur/dénominateur ».
  - h. Une méthode nommée **equals(Rationnel r)** permettant de tester l'égalité du rationnel courant avec le rationnel passé en paramètre (ex 1/2 et 2/4 sont égaux).
2. Ecrire une classe nommée **TestRationnel** contenant uniquement la méthode **main**. Vous créez 3 rationnels r1, r2 et r3 et vous ferez appel aux différentes méthodes définies précédemment.



## Leçon N°4

# Les chaînes de caractères

## I. Les objets de la classe String

---

### I.1. Instanciation d'un objet String

---

Les chaînes de caractères sont des instances de la classe **String** contenue dans le package **java.lang** et non pas des tableaux de caractères.

Les objets de type String peuvent être définis de plusieurs façons :

- `String maChaine = "toto";` // méthode simplifiée
- `String maChaine = new String("toto");` // appel explicite du constructeur
- `char[] listeCaracteres={'t','o','t','o'};` // tableau de caractères  
`String maChaine = new String(listeCaracteres);`

### Remarques

1. Les objets de type String sont immutables (constantes). Il n'existe pas de méthode permettant de modifier directement ces objets. A titre d'exemple, l'instruction suivante génère une erreur de compilation :

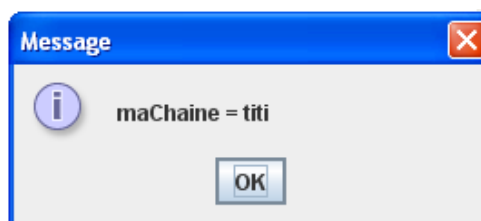
```
maChaine[3] = 'a' ; //interdit, une chaine n'est pas un tableau
```

En revanche, il est possible d'utiliser les méthodes qui renvoient une chaîne de caractères pour modifier le contenu de la chaîne courante.

### Exemple

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 String maChaine = "toto";
 maChaine = maChaine.replace('o','i');
 JOptionPane.showMessageDialog(null, "maChaine = "+ maChaine);
 }
}
```

### Output du programme



2. Java ne fonctionne pas avec le jeu de caractères ASCII ou ANSI, mais avec Unicode (Universal Code). Ceci concerne les types char et les chaînes de caractères. Le jeu de caractères Unicode code un caractère sur plusieurs octets ce qui permet de supporter les caractères accentués, les caractères arabes, etc. Les caractères 0 à 255 correspondent exactement au jeu de caractères ASCII étendu.

## I.2. Principales méthodes de la classe String

| Méthode                                                      | Description                                                                                                                                                                                        |
|--------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| int length()                                                 | Renvoie la longueur de la chaîne actuelle.                                                                                                                                                         |
| char charAt(int index)                                       | Renvoie le caractère à la position index. L'index du premier caractère d'une chaîne est 0.                                                                                                         |
| boolean equals(String autreChaine)                           | Renvoie vrai si la chaîne sur laquelle est appliquée la méthode est égale à autreChaine                                                                                                            |
| int indexOf(String chn)                                      | Renvoie l'index de la première occurrence de la chaîne chn dans l'objet en cours (ou -1 si la chaîne est absente).                                                                                 |
| String substring(int Depart, int Fin)                        | Renvoie un String contenant la chaîne de départ de l'index Debut à l'index Fin-1.<br><i>Exemple : System.out.println("String".substring(3,5));</i>                                                 |
| char[] toCharArray()                                         | Renvoie un tableau de caractères                                                                                                                                                                   |
| String toUpperCase()                                         | Renvoie la version tout en majuscules de la chaîne actuelle                                                                                                                                        |
| String toLowerCase()                                         | Renvoie la version tout en minuscules de la chaîne actuelle                                                                                                                                        |
| String trim()                                                | Renvoie une chaîne valant la chaîne de départ sans les blancs en début et fin de chaîne.                                                                                                           |
| static String valueOf(int / boolean / double / float / long) | Renvoie une chaîne correspondant à la valeur du primitif donné en argument.<br>Par exemple, valueOf(2) renverra "2".<br>L'opération inverse peut être réalisée par<br>int i=Integer.parseInt("2"); |

### I.2.1. Longueur d'une chaîne

Pour obtenir la longueur d'une chaîne de caractères, il est possible d'utiliser la méthode **length()** qui renvoie un entier représentant le nombre de caractères dans la chaîne. La chaîne vide "" est de longueur zéro.

⚠ Ne pas confondre la méthode **String.length()** avec la propriété **length** d'un tableau.

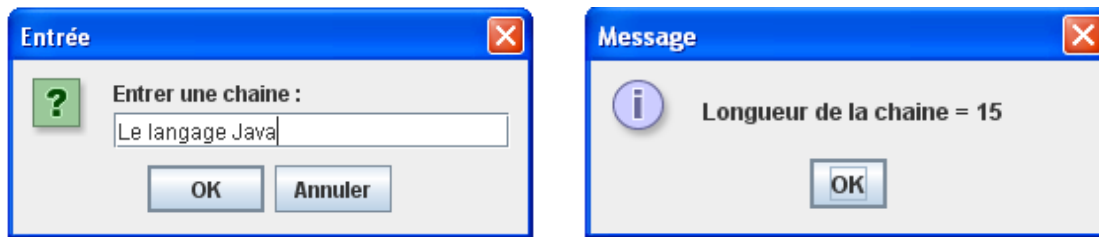
**Exercice :** Ecrire un programme en Java qui lit une chaîne de caractères (en mode graphique) puis affiche sa longueur.

#### Solution

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 String chn = JOptionPane.showInputDialog("Entrer une chaîne : ");
 JOptionPane.showMessageDialog(null, "Longueur de chn= "+chn.length());
 }
}
```



## Trace d'exécution



**Remarque :** Les caractères d'une chaîne `s` sont situés dans l'intervalle `[0..s.length()-1]`.

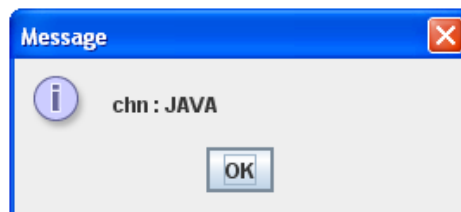
### I.2.2 Modification de la casse d'une chaîne

Les méthodes Java **`toUpperCase()`** et **`toLowerCase()`** permettent respectivement d'obtenir une chaîne tout en majuscules ou tout en minuscules.

#### Exemple

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 String chn = "Java";
 JOptionPane.showMessageDialog(null, "chn : "+ chn.toUpperCase());
 }
}
```

#### Output du programme



### I.2.3. Accès à un caractère dans une chaîne

La méthode **`charAt(int)`** permet d'accéder individuellement à chaque caractère d'une chaîne à partir de son index. Le premier caractère d'une chaîne non vide `s` est `s.charAt(0)`.

#### Exemple

```
String s = "une chaîne";
char c = s.charAt(1); // c = 'n'
```

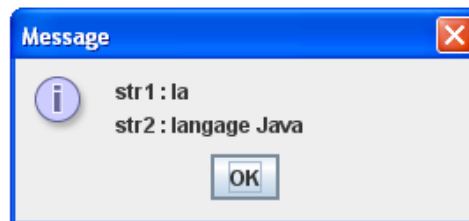
⚠ La tentative d'accès à un caractère en dehors de l'intervalle `[0 .. s.length()-1]` provoque la levée d'une exception *`StringIndexOutOfBoundsException`*.

### I.2.4. Extraction d'une sous-chaîne

La méthode **`substring(IndexDébut, IndexFin)`** permet d'extraire la chaîne comprise dans l'intervalle `[Indexdébut .. IndexFin-1]`. Si l'index de fin n'est pas précisé l'extraction se fait jusqu'à la fin de la chaîne.

Exemple

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 String chn = "le langage Java";
 String str1 = chn.substring(3,5);
 String str2 = chn.substring(3);
 JOptionPane.showMessageDialog(null,"str1 : "+ str1 + "\nstr2 : "+
 str2);
 }
}
```

Output du programme**I.2.5. Recherche d'une sous-chaîne**

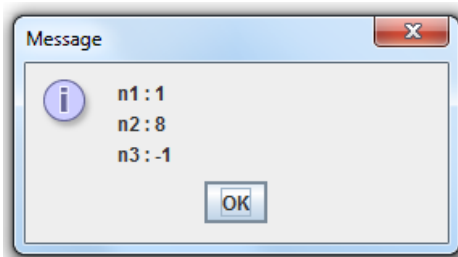
Les méthodes **indexOf( )** et **lastIndexOf( )** permettent de rechercher une sous-chaîne de caractères dans une chaîne. La 1<sup>ère</sup> méthode retourne l'indice de la 1<sup>ère</sup> occurrence de la sous-chaîne alors que la 2<sup>ème</sup> méthode retourne l'indice de la dernière occurrence.

Si la sous-chaîne n'existe pas dans la chaîne, la méthode renvoie (-1).

Exemple

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 String chn = "une chaîne";
 int n1 = chn.indexOf("ne");
 int n2 = chn.lastIndexOf("ne");
 int n3 = chn.indexOf("H");

 JOptionPane.showMessageDialog(null,"n1 : "+ n1 + "\nn2 : "+ n2
 +"\nn3 : "+ n3);
 }
}
```

Output du programme

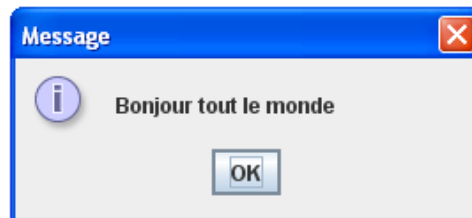
### I.3. Concaténation de chaînes de caractères

En Java, l'opérateur '+' permet de concaténer plusieurs chaînes de caractères. Il est également possible d'utiliser l'opérateur +=.

#### Exemple

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 String texte = "Bonjour"+" "+"tout";
 texte += " le monde";
 JOptionPane.showMessageDialog(null, texte);
 }
}
```

#### Output du programme

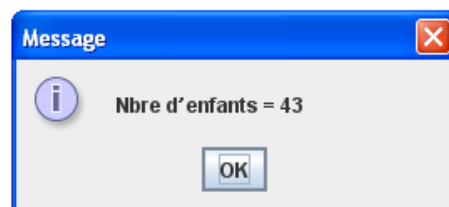


L'opérateur '+' sert aussi à concaténer des chaînes avec tous les types de base. La variable ou constante est alors convertie en chaîne puis ajoutée à la précédente. La condition préalable est d'avoir au moins une chaîne dans l'expression sinon le signe '+' est évalué comme opérateur mathématique. On dit que l'opérateur '+' connaît un *polymorphisme paramétrique* puisqu'il ne se comporte pas de la même façon en fonction du type de ses deux arguments.

#### Exemple

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 JOptionPane.showMessageDialog(null, "Nbre d'enfants = " + 4 + 3);
 }
}
```

#### Output du programme



Remarque: la concaténation de deux chaînes peut se faire en appelant la méthode « concat » comme dans l'exemple suivant :

```
String s = "long".concat("temp");
```

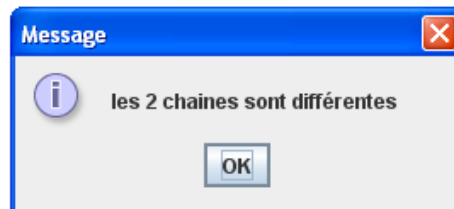
## I.4. Égalité de deux chaînes

Pour comparer deux chaînes de caractères, il ne faut pas utiliser l'opérateur « == » car dans ce cas, ce sont les deux références (c.-à-d. les adresses mémoires des 2 objets) qui sont comparées, mais il faut recourir à la méthode **equals()** (qui fait la comparaison caractère par caractère) .

### Exemple 1

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 String chn1 = new String("Java");
 String chn2 = new String("Java");
 if (chn1 == chn2)
 JOptionPane.showMessageDialog(null, "les 2 chaines sont identiques");
 else
 JOptionPane.showMessageDialog(null, "les 2 chaines sont différentes");
 }
}
```

### Output du programme

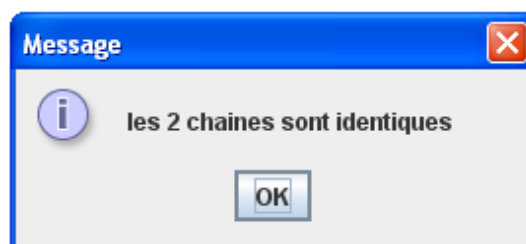


Les références chn1 et chn2 pointent sur deux objets différents bien qu'ils contiennent les mêmes caractères.

### Exemple 2

```
import javax.swing.JOptionPane;
public class Chaines {
 public static void main(String[] args) {
 String chn1 = new String("Java");
 String chn2 = new String("Java");
 if (chn1.equals(chn2))
 JOptionPane.showMessageDialog(null, "les 2 chaines sont identiques");
 else
 JOptionPane.showMessageDialog(null, "les 2 chaines sont différentes");
 }
}
```

### Output du programme



Remarque : Le code suivant :

```
String chn1 = "Java";
String chn2 = "Java";
```

permet de créer deux références qui pointent sur le même objet. Par conséquent l'expression (chn1 == chn2) renvoie la valeur true.

### I.5. la méthode toString()

---

Toutes les classes depuis « Object » (classe mère de toutes les classes) possèdent une méthode toString() qui renvoie un String :

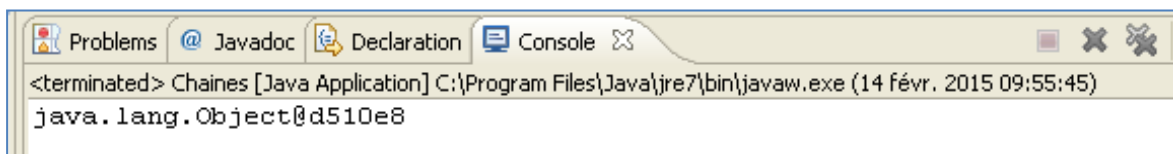
```
String toString()
```

Cette méthode sert à renvoyer une chaîne de caractère donnant des informations sur l'objet courant sous la forme d'une chaîne de caractères. Par exemple, la méthode toString() de la classe Object renvoie simplement le *hashcode* de l'objet.

#### Exemple

```
Object o = new Object();
System.out.println(o.toString());
```

#### Output



Il est ensuite possible de redéfinir cette méthode pour donner davantage d'informations.

La méthode println() de la classe « OutputStream » qui est très utilisée exécute toujours la méthode toString() et affiche la chaîne renvoyée.

donc faire :

```
Integer i = new Integer(2); //Integer : classe enveloppe (wrapper)
System.out.println(i);
```

est équivalent à :

```
System.out.println(i.toString());
```

### Exercice (QCM)

1. Parmi les réponses qui suivent, sélectionnez celles qui sont des classes Java.
  - a- boolean
  - b- char
  - c- Double
  - d- String
2. Quelle méthode permet de convertir une donnée de type primitif (int, double, boolean, ...) en String ?
  - a- toString()
  - b- toChar()
  - c- String.valueOf()
  - d- String.parseString()
3. Quelle méthode permet de convertir une chaîne de caractère en entier ?
  - a- valueOf()
  - b- int.parseInt()
  - c- Integer.parseInt()
4. L'indice du premier caractère d'une chaîne est :
  - a- 0
  - b- 1
  - c- 2
  - d- Ça dépend de la valeur spécifiée lors de la déclaration

5. Soit le tableau tabString déclaré comme suit :

```
String tabString[] = {"word", "wide", "web"};
```

Comment obtenir le nombre d'éléments contenus dans le tableau ?

- a- tabString.length;
  - b- tabString.length();
  - c- tabString.size();
  - d- tabString.capacity();
6. Soit le tableau tabString déclaré comme suit :
- ```
String tabString[] = {"word", "wide", "web"};
```

Comment obtenir la longueur du 1^{er} élément contenu dans le tableau ?

- a- tabString[0].length;
 - b- tabString[0].length();
 - c- tabString[1].length;
 - d- tabString[0].size();
7. Que va afficher l'instruction suivante ?

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

- a- true
- b- false
- c- une erreur de compilation

8. Que va afficher le fragment de code suivant ?

```
String s = "Bon".concat("jour");
System.out.println((s.startsWith("Bo")) && (s.endsWith("r")));
```

- a- true
- b- false
- c- une erreur de compilation

9. Que va afficher l'instruction suivante ?

```
System.out.println(" Bonjour ".trim().replace("jour","soir").substring(2,6));
```

- a- Bonsoir
- b- nsoir
- c- nsoi
- d- une erreur de compilation

10. Que va afficher le fragment de code suivant ?

```
System.out.print(3 + 3 + "3");
System.out.print(" and ");
System.out.println("3" + 3 + 3);
```

- a- 333 and 333
- b- 63 and 63
- c- 333 and 63
- d- 63 and 333

11. Que va afficher l'instruction suivante ?

```
System.out.println("aaa".replace("aa","b"));
```

- a- aaa
- b- ba
- c- ab
- d- bb

12. Soit la déclaration suivante :

```
String ville = "Sidi Bouzid";
```

Laquelle des déclarations suivantes est fausse ?

- a- `int pos = ville.indexOf('i');`
- b- `int pos = ville.indexOf('i',3);`
- c- `int pos = ville.indexOf("id");`
- d- `int pos = ville.lastIndexOf("id");`

13. Qu'est-ce qui est faux à propos de la méthode `toString()` ?

- a- toString() est une méthode de la super-classe Object
- b- la méthode toString() renvoie une description de l'objet courant sous forme de chaîne
- c- la méthode toString() ne peut pas être redéfinie.

14. Que va afficher le fragment de code suivant ?

```
String s1 = new String("Bonjour");  
StringBuffer s2 = new StringBuffer("Bonjour");  
System.out.println(s1.equals(s2));
```

- a- true
- b- false
- c- erreur de compilation

15. Que va afficher le fragment de code suivant ?

```
String s2 = new String("Java");  
System.out.println(s2.reverse());
```

- a- Java
- b- avaJ
- c- erreur de compilation

Exercices d'application

Exercice 1 : Créer une classe « **TestChaines** » qui contient 5 méthodes (toutes de type static):

- **public static String inverser(String chn)** : retourne l'inverse de la chaîne chn
- **public static boolean palind(String chn)** : vérifie si la chaîne chn est palindrome (se lit dans les 2 sens) ou non
- **public static int compter(char c, String chn)** : compte combien de fois le caractère c apparaît dans la chaîne chn
- **public static int compterMots(String chn)** : compte le nombre de mots contenus dans la chaîne chn (on suppose que les mots sont séparés par des espaces simples).
- **public static void main (String[] args)** : crée une chaîne puis teste les différentes méthodes de la classe.

Exercice 2 : Affichage indenté d'une chaîne

Créer une classe « **TestString** » réduite à une fonction « **main()** » qui demande la saisie d'une chaîne de caractères (en mode graphique) puis affiche cette dernière de façon indentée.

Exemple

```
Chaîne : toto
t
to
tot
toto
```

Exercice 3 : le mot le plus long

Ecrire un programme qui permet de lire une phrase puis affiche le mot le plus long dans cette phrase (vous pouvez utiliser la classe **StringTokenizer**).

Leçon N°5

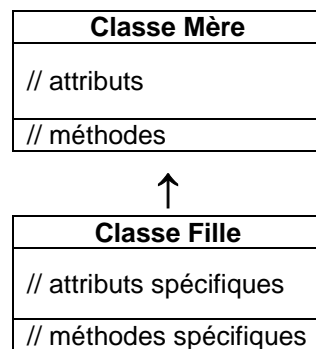
Héritage et polymorphisme

I. Notion d'héritage

L'héritage permet de donner à une nouvelle classe toutes les caractéristiques d'une classe existante. La classe dont elle hérite est appelée *classe mère* ou *classe de base*. La classe elle-même est appelée *classe fille* ou *classe dérivée*.

Les caractéristiques héritées sont les attributs et les méthodes de la classe de base (à l'exception des membres privés et des constructeurs).

En plus des membres hérités, une classe fille peut posséder des membres qui lui sont spécifiques.



II. Implémentation de l'héritage en Java

Pour implémenter une association d'héritage en Java, il faut faire suivre le nom de la classe dérivée du mot clé **extends** et du nom de classe de base.

La syntaxe générale est la suivante :

```

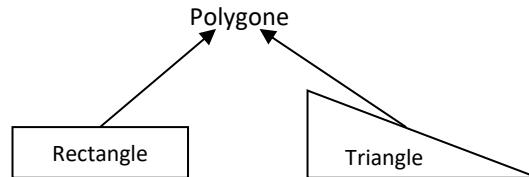
public class classe_mère {
    // attributs et méthodes de la classe mère
}
  
```

```

public class classe_fille extends classe_mère {
    // attributs et méthodes propres à la classe fille
}
  
```

Exemple

La classe « Polygone » englobe toutes les figures géométriques planes formées d'au moins trois segments appelés côtés comme le triangle et le rectangle. Ces deux formes possèdent des propriétés communes, telles que toutes les deux peuvent être décrites au moyen de seulement deux côtés : largeur et hauteur.



Ceci pourrait être représenté dans le monde des classes avec une classe mère « Polygone » à partir de laquelle seront dérivées deux classes filles « Rectangle » et « Triangle ». La classe « Polygone » contiendra les membres qui sont communs pour les deux formes à savoir la largeur et la hauteur. Alors que les classes filles contiendront chacune les caractéristiques spécifiques telles que les méthodes de calcul de la surface.

Programme

```

public class Polygone {
    protected double largeur, hauteur;          // en cm

    public Polygone(double l, double h)          // constructeur
    {
        largeur = l;
        hauteur = h;
    }
}
  
```

```

public class Rectangle extends Polygone {

    public Rectangle(double l, double h){
        super(l,h);
    }

    public double surface() {
        return (largeur * hauteur);
    }
}
  
```

```

public class Triangle extends Polygone {

    public Triangle(double l, double h){
        super(l,h);
    }

    public double surface () {
        return (largeur * hauteur)/2 ;
    }
}
  
```

```

public class TestPolygone {

    public static void main(String[] args){
        Rectangle r = new Rectangle(5,4);
        Triangle t = new Triangle(6,5);
        System.out.println("Surface du rectangle : "+r.surface()+" cm2");
        System.out.println("Surface du triangle : "+t.surface()+" cm2");
    }
}

```

Output du programme

```

Surface du rectangle : 20.0 cm2
Surface du triangle : 15.0 cm2

```

Remarques

1. Pour qu'un attribut de la classe mère puisse être utilisé dans une sous-classe, il faut que son mode d'accès soit public ou protected, ou, si les deux classes sont situées dans le même package, qu'il utilise le mode d'accès par défaut (package friendly).

Le tableau 1 fournit un récapitulatif des différents modes d'accès aux membres d'une classe selon les fonctions qui veulent accéder à ces membres.

Tableau 1 : Résumé des droits d'accès aux membres d'une classe

Mode d'accès	Emplacement des méthodes			
	Méthodes de la classe	Méthodes des classes dérivées	Méthodes des classes du même package	Méthodes externes
Private	Oui	Non	Non	Non
Public	Oui	Oui	Oui	Oui
Protected	Oui	Oui	Non	Non
Par défaut (package friendly)	Oui	Non (sauf si les classes sont dans le même package)	Oui	Non

2. Les constructeurs des classes Rectangle et Triangle ont fait appel au constructeur de la classe mère Polygone en utilisant le mot-clé **super**.

Exemple

```

Rectangle(double l, double h) {
    super(l,h);
}

```

L'appel au constructeur d'une classe supérieure doit toujours se situer dans un constructeur et toujours en tant que première instruction.

3. On dit qu'une méthode d'une sous-classe *redéfinit* une méthode de sa classe supérieure (@override), si elles ont la même signature mais que le traitement effectué est réécrit dans la sous-classe. Lors de la redéfinition d'une méthode, il est encore possible d'accéder à la méthode redéfinie dans la classe supérieure. Cet accès utilise également le mot-clé **super** comme préfixe à la méthode comme suit :

```

type Méthode(arguments) {
    ...
    super.Méthode(arguments) ;
    ...
}

```

4. Il est possible d'interdire la redéfinition d'une méthode en introduisant le mot-clé **final** au début de la signature de cette méthode. Il est également possible d'interdire l'héritage d'une classe en introduisant **final** au début de la déclaration de la classe (avant le mot-clé `class`).

Exercice (QCM)

1. La classe qui hérite d'une autre classe est aussi appelée la classe...

- a. Mère
- b. Fille
- c. Petite-fille

2. La portée `protected` empêche l'accès aux méthodes et attributs qui suivent depuis l'extérieur de la classe, sauf...

- a. dans les classes filles
- b. dans la classe mère
- c. dans la fonction `main()`

3. La classe B hérite de la classe A. Si A possède 3 méthodes et que B en possède 2 qui lui sont propres, combien de méthodes différentes un objet de type B pourra-t-il utiliser ?

- a. 2
- b. 3
- c. 5

4. En Java, peut-on surcharger une méthode redéfinie ?

- a. Oui
- b. Non

5. En Java, es-ce qu'une classe peut hériter de plusieurs super-classes ?

- a. Oui
- b. Non

Exercice

Une banque commerciale vous demande de lui développer un programme pour la gestion des comptes. On vous explique que vous devez gérer deux types de comptes : les comptes courants et les comptes d'épargne.

- Un compte courant a un numéro, un solde et un montant de découvert autorisé (le montant maximal que le client peut retirer lorsqu'il ne possède pas de solde suffisant).
- Un compte d'épargne a un numéro, un solde et un taux d'intérêt.
- Un compte est associé à une personne (civile ou morale) titulaire du compte, cette personne étant décrite par son nom. Une fois le compte créé, le titulaire du compte ne peut plus être modifié.

- Créditer un compte consiste à déposer un montant d'argent dans le compte. Ce montant sera ajouté au solde actuel.
- Débitier un compte consiste à retirer un montant d'argent (si le solde actuel et le découvert le permettent). Le découvert maximal autorisé peut varier d'un compte à un autre et est fixé arbitrairement par la banque à la création du compte. Il peut être ensuite modifié selon l'évolution des revenus du titulaire du compte.
- Effectuer un virement consiste à transférer un montant d'argent d'un compte bancaire à un autre (si le solde actuel et le découvert le permettent), ce qui revient débiter un compte au profit d'un autre compte qui sera crédité du même montant.
- Toutes les informations concernant un compte peuvent être consultées : numéro du compte, nom du titulaire, montant du découvert maximal autorisé, taux d'intérêts et solde actuel.

Travail demandé

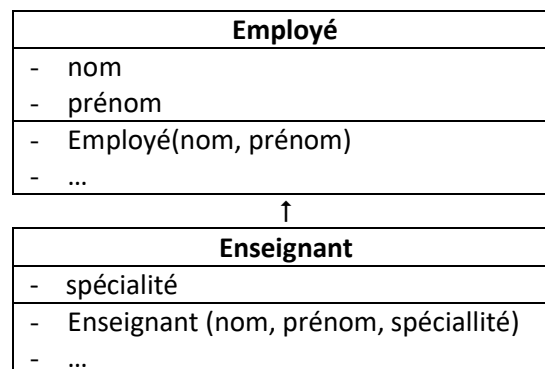
1. À partir du « cahier des charges » précédent, proposer un *diagramme de classes* qui modélise les différents types de comptes bancaires.
2. Réaliser une implémentation de ces classes en langage Java
3. Définir une classe « TestComptes » avec une fonction main() permettant de :
 - créer un compte courant c1, au nom de SALAH avec un solde initial de 1 000 D et un découvert de 300 D
 - créer un compte d'épargne c2, au nom de SALEM avec un solde initial de 5000 D et un taux d'intérêt de 8%.
 - retirer 300 D du compte c1.
 - retirer 600 D du compte c2.
 - déposer 500 D sur le compte c1.
 - virer 1000 D du compte c2 vers le compte c1.
 - afficher les caractéristiques des comptes c1 et c2.

III. Le polymorphisme

Le **polymorphisme** est la faculté attribuée à un objet d'être une instance de plusieurs classes. Un objet a une seule classe « réelle » qui est celle dont le constructeur a été appelé lors de la création (c'est-à-dire la classe figurant après le mot-clé `new`), mais il peut aussi être déclaré avec une classe supérieure à sa classe réelle. Cette propriété est très utile pour la création d'ensembles regroupant des objets de classes différentes (tableaux, collections, etc.).

Tout objet peut être vu comme une instance de sa classe réelle et aussi comme instance de tous ses ascendants (ses superclasses).

Exemple



```
Employé e1 = new Employé("Ben Mohahed", "Mohamed");
Enseignant e2 = new Enseignant("Ben Salah", "Salah", "Math");
e1 = e2; // upCasting implicite
Employé e3 = new Enseignant("Ben Amor", "Amor", "POO");
Enseignant e4 = (Enseignant) e3 ; // downCasting explicite
```

Remarques

1. Une **méthode polymorphe** est une méthode déclarée dans une super-classe puis redéfinie par une sous-classe. Ainsi, le polymorphisme peut être vu comme la capacité de choisir dynamiquement la méthode qui correspond au type réel de l'objet.

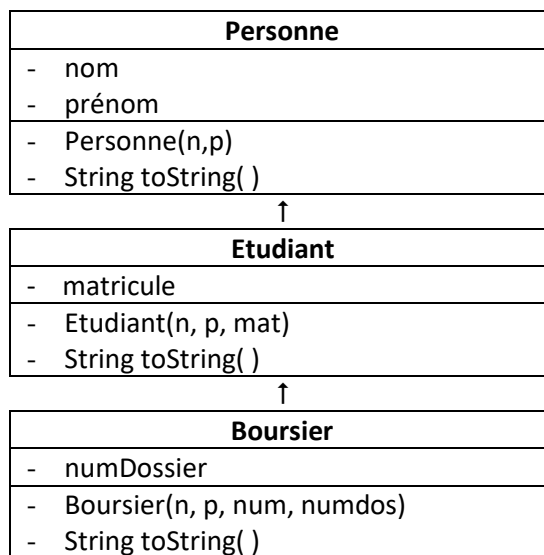
Autrement dit, le polymorphisme désigne l'aptitude de différents objets d'effectuer chacun le traitement adéquat suite à la réception d'un même message.

2. Il ne faut pas confondre les notions de *surcharge* (ou surdéfinition) et polymorphisme (ou liaison dynamique) qui ont finalement peu de points communs. Voici un tableau qui résume les différences :

	Surcharge	Polymorphisme
Héritage	nul besoin	nécessite une arborescence de classes
signature des méthodes	doivent différer	doivent être les mêmes (redéfinition)
résolu à	la compilation	l'exécution

Exemple

1. Implémenter les 3 classes suivantes :



La méthode toString() doit retourner, selon le cas, une chaîne sous l'une des formes suivantes :

- Nom = leNom, Prénom = lePrénom
- Nom = leNom, Prénom = lePrénom, Matricule = leMatricule
- Nom = leNom, Prénom = lePrénom, Matricule = leMatricule, NumDossier = leNuméro

2. Définir une classe « TestPersonnes » réduite à une méthode main() permettant de :

- Créer un tableau « tabPers » qui contient les trois personnes suivantes :

Catégorie	Données		
Personne	Nom : Ben Ali	Prénom : Ali	-
Etudiant	Nom : Ben Salah	Prénom : Salah	NCE : 15160001
Etudiant boursier	Nom : Ben Mohamed	Prénom : Mohamed	NCE : 14150002 Num Dossier : 10013

- Parcourir le tableau « tabPers » et afficher une description de toutes les personnes.

Solution

```
public class Personne {
    protected String nom;
    protected String prénom;

    public Personne(String nom, String prénom) {
        this.nom = nom;
        this.prénom = prénom;
    }

    public String toString() {
        return "Nom=" + nom + ", Prénom=" + prénom;
    }
}
```

```
public class Etudiant extends Personne {
```

```

protected int NCE;

public Etudiant(String nom, String prénom, int NCE){
    super(nom, prénom);
    this.NCE = NCE;
}

public String toString(){
    return super.toString()+" , Matricule=" + matricule;
}
}

```

```

public class Boursier extends Etudiant{

    private int numDoss;

    public Boursier(String nom, String prénom, int NCE, int numDoss){
        super(nom, prénom, NCE);
        this.numDoss = numDoss;
    }

    public String toString(){
        return super.toString()+" , NumDossier=" + numDoss;
    }
}

```

```

public class TestPers {

    public static void main(String[] args) {
        Personne tabPers[] = new Personne[3];
        tabPers[0] = new Personne("Ben Ali", "Ali");
        tabPers[1] = new Etudiant("Ben Salah", "Salah", 15160001);
        tabPers[2] = new Boursier("Ben Mohamed", "Mohamed", 1415002, 10013);

        for(int i = 0; i < tpers.length; i++){
            System.out.println(tpers[i].toString());
        }
    }
}

```

Output du programme

```

Nom=Ben Ali, Prénom=Ali
Nom=Ben Salah, Prénom=Salah, Matricule=15160001
Nom=Ben Mohamed, Prénom=Mohamed, Matricule=1415002, NumDoss=10013

```

Remarques

1. Une des propriétés induites par le polymorphisme est que l'interpréteur Java est capable de trouver le traitement à effectuer lors de l'appel d'une méthode sur un objet. Ainsi, pour plusieurs objets déclarés sous la même classe (mais n'ayant pas la même classe réelle), le traitement associé à une méthode donné peut être différent. Si cette méthode est redéfinie par la classe réelle d'un objet (ou par une classe située entre la classe réelle et la classe de déclaration), le traitement effectué est celui défini dans la classe la plus spécifique de l'objet et qui redéfinit la méthode.

Dans l'exemple précédent, la méthode `toString()` est redéfinie dans toutes les sous-classes de « Personne » et le traitement effectué est :

```

for (int i = 0 ; i < tpers.length ; i++) {
    System.out.println(tpers[i].toString());
}

```

2. On ne peut pas réduire le niveau de visibilité (le mode d'accès) pour une méthode redéfinie dans une sous-classe.

3. L'ensemble des classes prédéfinies de Java ainsi que celles définies par le programmeur forme une hiérarchie avec une racine unique. Cette racine est la classe `Object` dont hérite toute autre classe. Grâce à cette propriété, il est possible de créer des tableaux, des ensembles ou des collections regroupant des objets appartenant à la classe `Object` (donc de n'importe quelle classe).
4. L'opérateur **`instanceof`** peut être utilisé pour tester l'appartenance d'un objet à une classe comme dans l'exemple suivant :

```
for (int i = 0 ; i < tpers.length ; i++) {
    if (tpers[i] instanceof Personne)
        System.out.println(tpers[i].prénom + " est une personne");
    if (tpers[i] instanceof Etudiant)
        System.out.println(tpers[i].prénom + " est un étudiant");
    if (tpers[i] instanceof Boursier)
        System.out.println(tpers[i].prénom + " est un étudiant boursier");
}
```

L'exécution de ce code sur le tableau précédent affiche le texte suivant :

```
Ali est une personne
Salah est une personne
Salah est un étudiant
Mohamed est une personne
Mohamed est un étudiant
Mohamed est un étudiant boursier
```

5. La méthode **`getClass()`** permet de récupérer la classe *réelle* d'un objet quelconque.

Exercice : On veut développer une application devant servir à l'inventaire d'une bibliothèque. Elle devra traiter des documents de nature diverse : des livres, des dictionnaires, et autres types de documents qu'on ne connaît pas encore précisément mais qu'il faudra certainement ajouter un jour (articles, mémoires, ...). Tous les documents possèdent un **numéro d'enregistrement** et un **titre**. A chaque livre est associé, en plus, un **auteur** et un **nombre de pages**, les dictionnaires ont, eux, pour attributs supplémentaires une **langue** et un **nombre de tomes**. On veut manipuler tous les articles de la bibliothèque au travers de la même représentation : celle d'un document.

1. Dessiner un diagramme de classe qui modélise cette application
2. Définissez les classes **Document**, **Livre** et **Dictionnaire**. Définissez pour chacune un constructeur permettant d'initialiser toutes ses variables d'instances ainsi que la méthode **`toString()`** qui renvoie une chaîne de caractères décrivant un document, un livre ou un dictionnaire.
3. Définissez une classe **Bibliothèque** réduite à une méthode **`main()`** permettant de tester les classes précédentes à l'aide des documents suivants :
 - Document N°1, "Configurer son compte utilisateur";
 - Document N°2 (Livre), "La communication sous Unix", "J-M Rifflet", 799 pages
 - Document N°3 (Dictionnaire), "Man pages", "Anglais", 4 tomes.

Créer un vecteur `vDoc` et le remplir par les 3 documents définis précédemment, puis parcourir ce vecteur et afficher une description des différents documents.

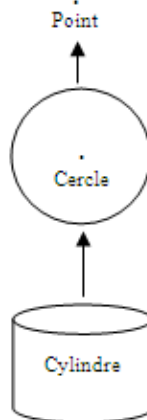
Exercices d'application

Exercice 1 : Soient les 4 classes C1, C2, C3 et C4 qui comportent les attributs suivants :

C1(A, B, C)
 C2(A, B, C, D)
 C3(A, B, C, E)
 C4(A, B, C, E, F)

Proposer une hiérarchie d'héritage optimale pour ces classes.

Exercice 2 : Soit le graphe d'héritage suivant :



Définir la classe **Point** qui comporte :

- Deux attributs de type entier (abscisse et ordonnée)
- Un constructeur sans arguments Point() qui initialise le point aux coordonnées de l'origine (0,0)
- Un constructeur avec arguments Point(abs,ord)
- Deux accesseurs getAbs() et getOrd() qui retournent respectivement l'abscisse et l'ordonnée du point
- Deux mutateurs setAbs(abs) et setOrd(ord) qui permettent de modifier respectivement l'abscisse et l'ordonnée du point
- Une méthode toString() qui retourne les coordonnées du point sous la forme :

[abs,ord]

1- Définir la classe **Cercle** qui est une sous-classe de la classe Point et qui possède (en plus des coordonnées du centre):

- Un attribut rayon de type réel
- Un constructeur sans arguments Cercle() qui initialise le rayon à 0
- Un constructeur avec arguments Cercle(abs, ord, rayon)
- Un accesseur getRayon()
- Un mutateur setRayon(rayon)
- Une méthode aire() qui retourne la surface du cercle
- Une méthode toString() qui retourne la description du cercle sous la forme :

Centre : [abs,ord] Rayon : rayon Aire(cercle) : aire

2- Définir la classe **Cylindre** qui est une sous-classe de la classe **Cercle** et qui comporte (en plus des membres hérités de ses superclasses):

- Un attribut hauteur de type réel
- Un constructeur sans arguments `Cylindre()` qui initialise la hauteur à 0
- Un constructeur avec arguments `Cylindre(abs, ord, rayon, haut)`
- Un accesseur `getHauteur()`
- Un mutateur `setHauteur(haut)`
- Une méthode `aire()` qui retourne la surface du cylindre selon la formule :

$$a = 2 * \text{aire du cercle} + 2 * \pi * \text{rayon} * \text{hauteur}$$

- Une méthode `volume()` qui retourne le volume du cylindre selon la formule :

$$v = \text{aire du cercle} * \text{hauteur}$$

- Une méthode `toString()` qui retourne la description du cylindre sous la forme :

Centre : [abs,ord] Rayon : rayon Aire(cercle) : aire Hauteur : haut Volume : vol

3- Définir la classe **Test** réduite à une méthode `main()` qui permet de :

- Créer un point `P(3,6)`
- Créer un cercle `C` de centre `P` et de rayon `r = 10 cm`
- Créer un `Cylindre Cyl` de centre `P`, de rayon `r` et de hauteur `h = 15 cm`
- Ranger les 3 objets dans un tableau `tabFormes[]` puis le parcourir et afficher une description de chacun d'eux.

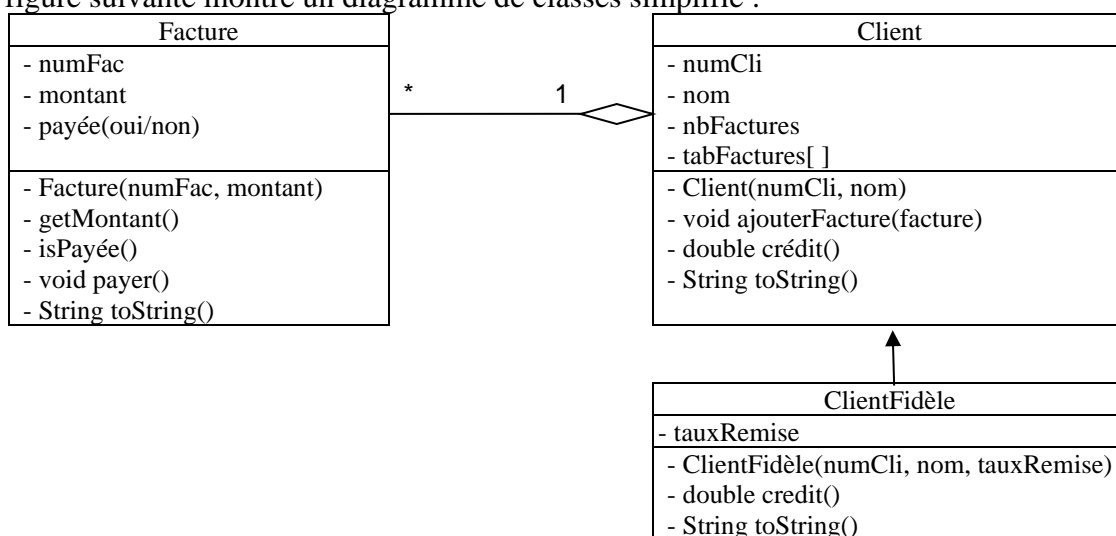
Exercice 3 : Une entreprise commerciale vous demande de lui développer un programme en Java pour gérer les factures de ses clients.

Une facture est caractérisée par un numéro unique, un montant (en Dinars) et un champ qui indique si la facture a été payée ou non.

Chaque client est caractérisé par un numéro unique, un nom et possède un certain nombre de factures rangées dans un tableau de taille maximale 20. Pour chaque objet client nouvellement créé, le nombre de factures doit être initialisé à zéro.

Les clients fidèles de l'entreprise bénéficient d'une remise sur le total des factures à payer. Le taux de cette remise varie d'un client fidèle à un autre.

La figure suivante montre un diagramme de classes simplifié :



Remarques

- Chaque facture nouvellement créée est supposée non encore payée
- La méthode payer() permet de changer l'état d'une facture suite à son règlement par le client
- La méthode crédit() fournit le montant total des factures impayées du client actuel. Si c'est un client fidèle, il faut déduire le montant de la remise.
- La méthode toString() retourne une description textuelle de la facture ou du client actuel (avec notamment son crédit)

Travail demandé

1. Définir la classe « Facture »
2. Définir la classe « Client »
3. Définir la classe « ClientFidèle»
4. Définir une classe « TestFactures » réduite à une fonction main() qui permet de :
 - a. Créer les 5 factures suivantes :

○ Facture F1	numéro : 1	Montant : 150 D	(*payée)
○ Facture F2	numéro : 2	Montant : 200 D	
○ Facture F3	numéro : 3	Montant : 150 D	
○ Facture F4	numéro : 4	Montant : 200 D	
○ Facture F5	numéro : 5	Montant : 300 D	
 - b. Créer un tableau « tabClients » qui contient les 3 clients suivants :

○ Client C1	numéro : 101	nom : Adel	Factures : {F1, F2}
○ Client C2	numéro : 102	nom : Ahmed	Factures : {F3, F4}
○ ClientFidèle C3	numéro : 103	nom : Ali	Factures : {F5}

TauxRemise : 10%
 - c. Parcourir le tableau « tabClients » et afficher une description des clients qui y figurent.

Exercice 4

1. Développer en java la classe **Robot** obéissant au schéma UML suivant :

Robot
#Code : long #type : String #etat : booléen #orientation : char
+Robot(code : long, type : String) +getCode() : long +getType() : String +getEtat() : boolean +getOrientation() : char +setEtat(etat: boolean) : void +setOrientation(sens : char) : void +tourner(sens:char) : void +affiche() : void

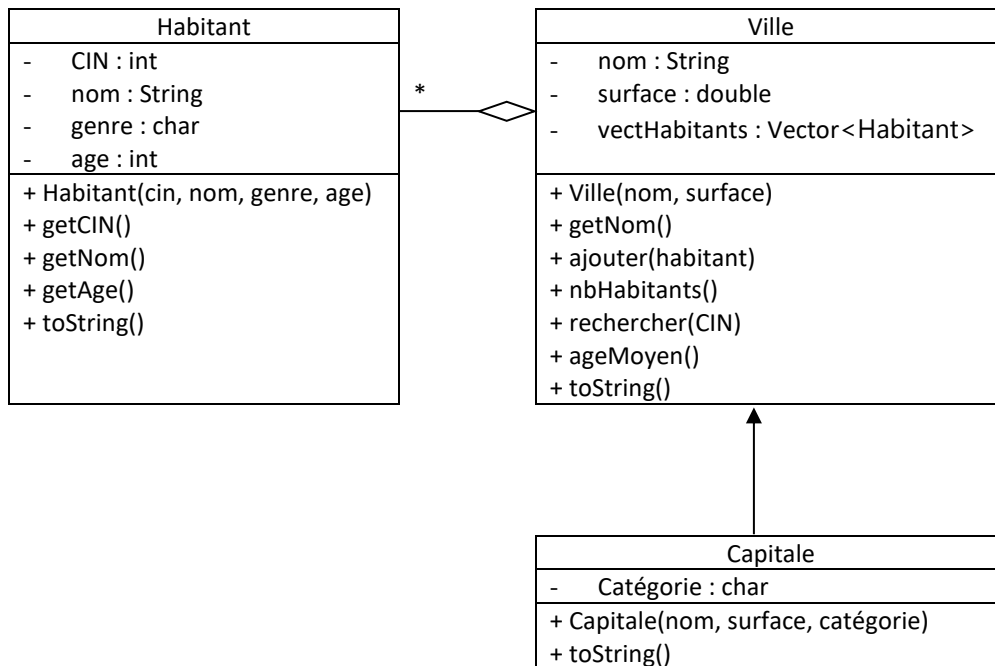
Remarques

- Le symbole # signifie membre protégé alors que le symbole + signifie membre public.
 - Un robot peut être en état de marche (true) ou en état d'arrêt (false)
 - L'attribut orientation ne peut prendre que l'une des valeurs suivantes : N (Nord), E (Est), S (Sud), O (Ouest).
 - La méthode **tourner ()** permet d'ajuster l'orientation du Robot.
 - Les nouveaux robots sont toujours orientés vers le nord en état d'arrêt.
 - La méthode **affiche()** permet d'afficher le code, le type, l'état et l'orientation actuelle du Robot.
2. Ecrire une classe **TestRobot** dont la fonction main comprendra la création d'un tableau de quatre robots, initialiser leurs attributs, les mettre en marche et les orienter vers des orientations différentes chacun de l'autre et enfin les afficher.
3. Soit une classe **RobotMobile** qui hérite de **Robot** et ayant en plus :
- les attributs entiers privés **abs** et **ord** qui définissent la position actuelle du robot
 - une méthode **void avancer(int x)** qui permet d'avancer le robot selon son orientation :
 - si on avance de x vers l'Est l'abscisse augmente de x,
 - si on avance de x vers le West l'abscisse diminue de x,
 - si on avance de x vers le nord l'ordonnée augmente de x,
 - si on avance de x vers le Sud l'ordonnée diminue de x,
 - une méthode **void affichePosition()** qui affiche la position actuelle (coordonnées).

Développer la classe RobotMobile (prévoyez un constructeur à quatre arguments (code, type, abs et ord) et redéfinissez la méthode **affiche()** en utilisant celle de la classe mère et la méthode **affichePosition()**).

4. Ecrire une classe **TestRobotMobile** dont la méthode main permet de :
- créer un RobotMobile (code : 6672, nom : Arlo), ce robot est supposé initialement positionné au point de coordonnées (0,0).
 - Mettre en marche le robot et lui appliquer la séquence d'actions suivante :
 - Avancer de 6 cm vers le Nord
 - Avancer de 4 cm vers le West
 - Avancer de 14 cm vers l'Est
 - Avancer de 8 cm vers le Sud
 - Arrêter le robot et en afficher une description complète avec notamment sa position finale et son orientation.

Exercice 6 : Soit le diagramme de classes suivant :



Travail demandé

- Définir la classe « Habitant ». Le genre d'une personne peut être M (pour Masculin) ou F (pour Féminin).
- Définir la classe « Ville ». Cette classe contient un vecteur d'habitants. Chaque ville nouvellement créée est supposée vide (sans aucun habitant).
 - La méthode rechercher (CIN) doit retourner toutes les informations disponibles sur la personne recherchée si elle habite la ville en cours ou le message « Habitant introuvable ».
 - La méthode ageMoyen() doit retourner l'âge moyen de tous les habitants de la ville actuelle.
 - La méthode toString() doit retourner le nom, le nombre d'habitants, la surface et l'âge moyen des habitants de la ville en cours.
- Définir la classe « Capitale ». l'attribut « catégorie » peut prendre l'une des valeurs 'P' (capitale politique), 'E' (capitale économique) ou 'T' (capitale politique et économique).
- Définir une classe « Test » réduite à une méthode « main() » permettant de :
 - Créer 6 habitants H1 ... H6, 2 villes V1 et V2 et une capitale C.
 - Affecter à chaque ville 2 habitants (dans l'ordre).
 - Créer un tableau tabVilles puis le charger par les 3 villes V1, V2 et C.
 - Parcourir le tableau tabVilles et afficher une description de toutes les villes qui y figurent.

Leçon N°6

Interfaces et classes abstraites

I. Les interfaces

Une **interface** est un type, au même titre qu'une classe, mais abstrait et qui donc ne peut pas être instancié (par appel à new plus constructeur). Une interface décrit un ensemble de *signatures de méthodes* qui doivent être implémentées dans toutes les classes qui *implémentent* l'interface et peut contenir des constantes.

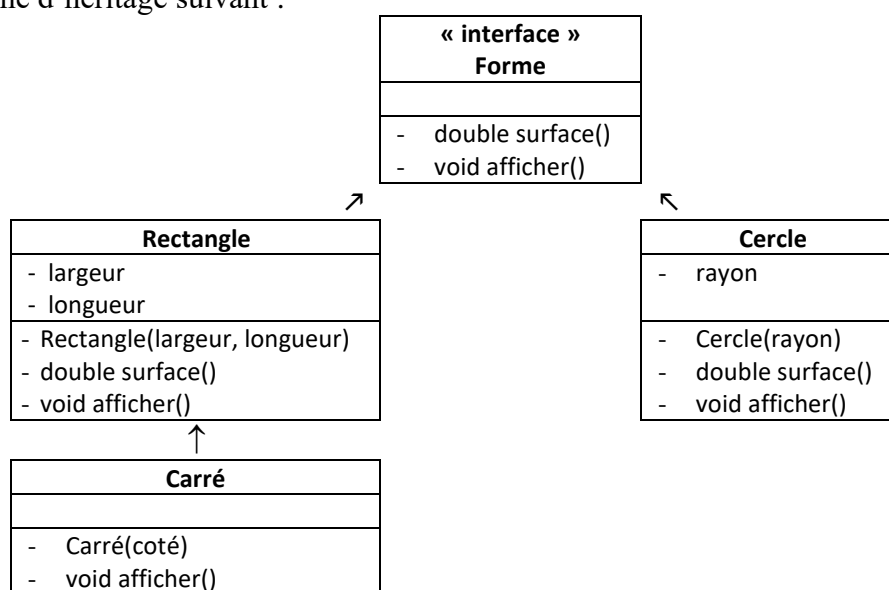
L'utilité du concept d'interface réside dans le regroupement de plusieurs classes, tel que chacune implémente un ensemble commun de méthodes, sous un même type.

Une interface possède donc les caractéristiques suivantes :

- elle ne contient que des signatures de méthodes (sans implémentation) et éventuellement des attributs constants (qui doivent être initialisés);
- elle ne peut pas être instanciée ;
- une classe (abstraite ou non) peut implémenter plusieurs interfaces. La liste des interfaces implémentées doit alors figurer après le mot-clé **implements** placé dans la déclaration de la classe, en séparant chaque interface par une virgule ;
- Une interface peut être implémentée par plusieurs classes. Ses méthodes sont alors polymorphes (définies de différentes manières) ;
- une interface peut hériter d'une ou plusieurs autres interfaces (avec le mot-clé **extends**).

Exemple

Soit le graphe d'héritage suivant :



1. implémenter l'interface « Forme »

2. implémenter la classe « Rectangle »
3. Implémenter la classe « Cercle »
4. Implémenter la sous-classe « Carré »
5. Définir une classe « TestFormes » réduite à une méthode main() qui permet de :
 - créer un tableau « tabFormes » pouvant contenir 3 objets de type « Forme »
 - remplir le tableau « tabFormes » par un rectangle de dimensions (10 cm x 20 cm), un cercle de rayon 10 cm et un carré de côté 15 cm.
 - parcourir le tableau et afficher une description des objets qu'il contient avec notamment leurs surfaces successives.

Le résultat attendu est le suivant :

```
Rectangle [longueur = 20.0, largeur = 10.0, surface = 200.0]
Cercle [rayon = 10.0, surface = 314.0]
Carré [Coté = 15.0, surface = 225.0]
```

Solution

```
public interface Forme {
    public double surface();
    public void afficher();
}
```

```
public class Rectangle implements Forme{
    protected double largeur;
    protected double longueur;

    public Rectangle(double largeur, double longueur) {
        this.largeur = largeur;
        this.longueur = longueur;
    }

    public double surface()
    {
        return (largeur * longueur);
    }

    public void afficher(){
        System.out.println("Rectangle [longueur = "+longueur+", largeur = "
            +largeur+", surface = "+surface()+"");
    }
}
```

```
public class Cercle implements Forme{
    private double rayon;

    public Cercle (double r) {
        this.rayon = r;
    }

    public double surface()
    {
        return Math.PI*rayon*rayon;
    }

    public void afficher(){
```

```

        System.out.println("Cercle [rayon = "+rayon+", surface = "
            +surface()+"");
    }
}

```

```

public class Carré extends Rectangle{

    public Carré(double coté){
        super(coté,coté);
    }

    public void afficher(){
        System.out.println("Carré [Coté = "+largeur+", surface = "
            +surface()+"");
    }
}

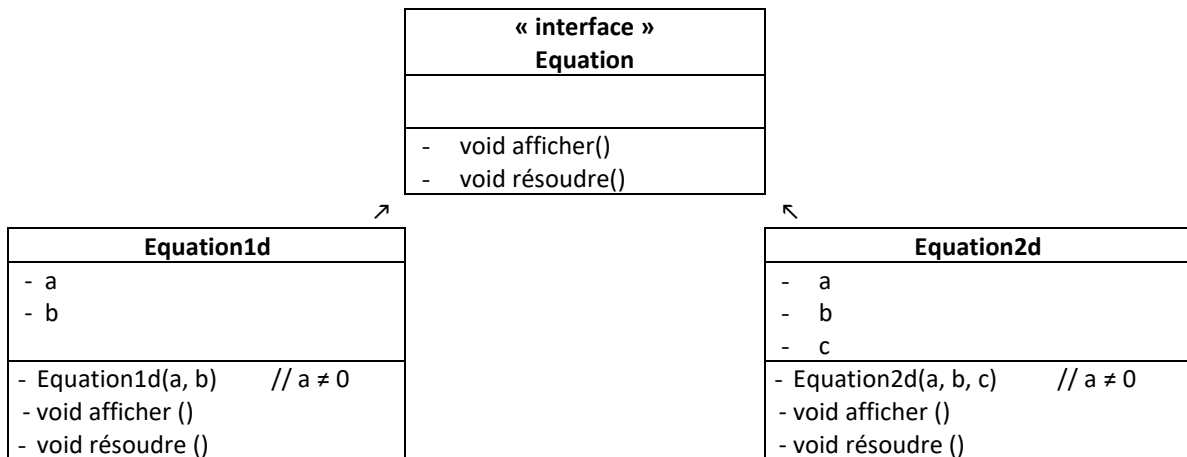
```

```

public class Test {
    public static void main(String[] args) {
        Forme tabFormes[] = new Forme[3];
        tabFormes[0] = new Rectangle(10,20);
        tabFormes[1] = new Cercle(10);
        tabFormes[2] = new Carré(15);
        for(Forme f:tabFormes){
            f.afficher();
        }
    }
}

```

Exercice : Soit la hiérarchie suivante :



1. implémenter l'interface Equation
2. implémenter la classe Equation1d
3. Implémenter la classe Equation2d
4. Définir une classe « TestEquations » réduite à une méthode main() qui permet de :

- créer un tableau tab pouvant contenir 2 objets de type Equation
- remplir le tableau avec les équations ($x+5=0$) et ($x^2+5x+1=0$)
- parcourir le tableau et afficher les équations qu'il contient avec leurs racines respectives.

Le résultat attendu est le suivant :

```
Equation : 1,00x + 5,00 = 0
Solution = -5,00
-----
Equation : 1,00x2 + 5,00x + 1,00 = 0
Racine1 = -0,21
Racine2 = -4,79
```

Remarque : Si une classe implémente une interface mais que le programmeur n'a pas écrit l'implémentation de toutes les méthodes de l'interface, une erreur de compilation se produira sauf si la classe est une *classe abstraite* (voir section suivante).

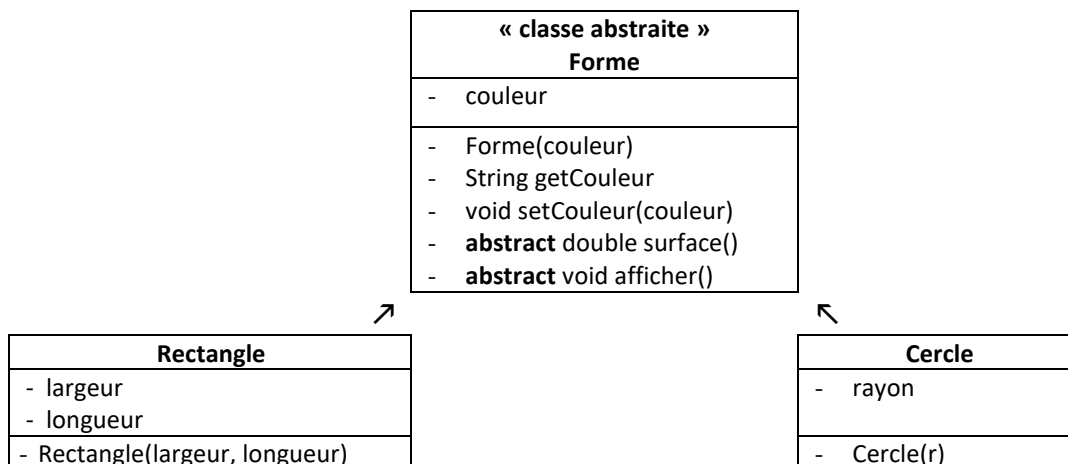
II. Les classes abstraites

Le concept de classe abstraite se situe entre celui de classe (concrète) et celui d'interface. C'est une classe qu'on ne peut pas directement instancier (créer des objets) car certaines de ses méthodes ne sont pas implémentées.

Une classe abstraite peut donc contenir des attributs, des méthodes implémentées et des signatures de méthodes à implémenter par les classes dérivées (*méthodes abstraites*). Une classe abstraite peut implémenter (partiellement ou totalement) des interfaces et peut hériter d'une autre classe (concrète ou abstraite).

Syntaxiquement, le mot-clé **abstract** est utilisé devant le mot-clé **class** pour déclarer une classe abstraite, ainsi que pour la déclaration de signatures de méthodes à implémenter.

Exemple : Soit le graphe d'héritage suivant :



Imaginons que l'on souhaite attribuer une couleur à tout objet représentant une forme. Comme une interface ne peut pas contenir des attributs variables, il faut déclarer « Forme » comme une classe abstraite comme suit :

```
public abstract class Forme {
    protected String couleur;

    public Forme(String couleur) {
        this.couleur = couleur ;
    }

    public String getCouleur() {
        return couleur;
    }

    public void setCouleur(String couleur) {
```

```

        this.couleur = couleur ;
    }

    public abstract double surface();           // méthode abstraite

    public abstract void affiche();           // méthode abstraite
}

```

Par la suite, il faut rétablir l'héritage des classes « Rectangle » et « Cercle » vers la classe abstraite « Forme » :

```

public class Rectangle extends Forme {
    ...
}

public class Cercle extends Forme {
    ...
}

```

Remarques

1. *Lorsqu'une classe hérite d'une classe abstraite, elle doit :
 - soit implémenter les méthodes abstraites de sa super-classe en les dotant d'un corps ;
 - soit être elle-même abstraite si au moins une des méthodes abstraites de sa super-classe reste abstraite.
2. Une classe abstraite peut ne pas contenir de méthodes abstraites. Mais dès qu'une classe contient une méthode abstraite ; elle doit être obligatoirement déclarée abstraite.
3. On ne peut pas créer des objets à partir d'une classe abstraite (avec l'opérateur new) ; mais on peut déclarer des variables (références) de ce type.

Exercice (QCM) : Cocher à chaque fois la/les réponse(s) correcte(s)

1. Quelle relation lie les classes et les interfaces
 - a. Une classe peut implémenter plusieurs interfaces mais ne peut étendre qu'une seule classe
 - b. Une classe peut implémenter plusieurs classes mais ne peut étendre qu'une seule interface
 - c. Une classe peut implémenter plusieurs classes et peut étendre plusieurs interfaces
 - d. Une classe peut implémenter une seule interface et étendre une seule classe
2. Pour empêcher une classe d'être étendue, quel est le mot réservé que l'on utilise dans la déclaration de classe ? (public ??? class MaClasse {})
3. Lequel des énoncés suivants peut être déclaré final ?
 - a. protected
 - b. final
 - c. abstract
 - d. static

- a. Classes
 - b. Attriuts membres de la classe
 - c. Méthodes membres de la classe
4. Une classe déclarée finale ne doit avoir aucune méthode abstraite
- a. Vrai
 - b. Faux
5. Le polymorphisme sert à :
- a. Rendre abstraite une classe concrète
 - b. Se passer des interfaces
 - c. Standardiser les relations entre objets de nature distincte
6. Lequel est un exemple de polymorphisme ?
- a. Les classes internes
 - b. Classes anonymes
 - c. La surcharge de méthode
 - d. La redéfinition de méthode
7. Java supporte-il l'héritage multiple ?
- a. Oui
 - b. Non
8. L'interface en Java est :
- a. Equivalente à une classe abstraite avec des données membres
 - b. Une classe à part entière
 - c. Une forme de classe abstraite sans données membres et sans code de traitement
9. Le mot clé « protected » associé à une variable ou méthode :
- a. Sert à interdire tout usage externe à la classe
 - b. Sert à donner le maximum de visibilité
 - c. Donne un niveau de visibilité aux classes filles
10. Quelle classe n'a pas de classe mère ?
- a. Orpheline
 - b. String
 - c. Object
 - d. Toute classe abstraite
11. Qu'est-ce qui est faux pour les interfaces ?

- a. Une interface déclare des méthodes sans les implémenter
- b. Une interface peut être implémentée par plusieurs classes
- c. Une interface peut hériter d'une interface
- d. Une interface peut être instanciée
- e. Une interface peut être le type d'une référence

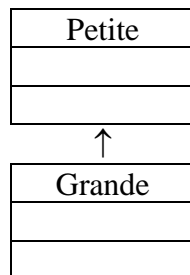
12. Etant donne que la classe Triangle étend la classe Figure, trouvez une ligne correcte parmi les suivantes

- a. `Triangle x = new Triangle(); Object y = (Object)x; Triangle z = y;`
- b. `Figure y = new Figure(); Triangle x = (Triangle)y; Figure z = x;`
- c. `Figure y = new Figure(); Triangle x = (Triangle)y; Figure z = (Figure)x;`
- d. `Triangle x = new Triangle(); Figure y = x; Triangle z = (Triangle)y;`

13. Laquelle des opérations ci-dessus est interdite en Java ?

- a. le upcasting explicite
- b. le upcasting implicite
- c. le downcasting explicite
- d. le downcasting implicite

14. Etant donné que la classe Grande étend la classe Petite, trouvez une ligne correcte parmi les suivantes :



- a. `Petite y = new Petite(); Grande x = (Grande)y; Petite z = x;`
- b. `Grande x = new Grande(); Petite y = x; Grande z = (Grande)y;`
- c. `Grande x = new Grande(); Petite y = x; Grande z = y;`
- d. `Petite y = new Petite(); Grande x = (Grande)y; Petite z = (Petite)x;`

15. Soit les 2 classes suivantes :

```

public class A{
    ...
    public void affiche(){
        System.out.println("Je suis un objet de la classe A") ;
    }
}
  
```

```

public class B extends A{
    ...
    public void affiche(){
        System.out.println("Je suis un objet de la classe B") ;
    }
}
  
```

Quel est le résultat d'exécution du programme suivant :

```

public class Test{
    public static void main(String [] args) {
        A b = new B() ;
        b.affiche() ;
    }
}

```

- a- Une erreur de compilation
- b- Je suis un objet de la classe A
- c- Je suis un objet de la classe B

16. Soit les 3 classes suivantes :

```

public class A{
    ...
    public void affiche(){
        System.out.println("Je suis un objet de la classe A") ;
    }
}

```

```

public class B extends A{
    ...
    // pas de redéfinition de la méthode affiche
}

```

```

public class C extends B{
    ...
    public void affiche(){
        super.affiche() ;
        System.out.println("Je suis un objet de la classe C") ;
    }
}

```

Quel est le résultat d'exécution du programme suivant :

```

public class Test{
    public static void main(String [] args) {
        C c = new C() ;
        c.affiche() ;
    }
}

```

- a- Une erreur de compilation
- b- Je suis un objet de la classe A
- c- Je suis un objet de la classe C
- d- Je suis un objet de la classe A
Je suis un objet de la classe C

17. Pour la classe B définie comme suit:

```
public class B
{
    public B()
    {
        System.out.print("Bonjour");
    }
    public B(int i)
    {
        this();
        System.out.println("Bonsoir "+i);
    }
}
```

Qu'affichera l'instruction suivante?

```
B monB=new B(2018);
```

- a. Bonsoir 2018
- b. BonjourBonsoir 2018
- c. erreur de compilation
- d. erreur d'exécution

18. Une méthode de classe est :

- a. Une méthode visible à tous les niveaux
- b. Une méthode déclarée « static »
- c. Une méthode accessible sans instance de classe
- d. Une forme de macro

19. Pour la classe C définie comme suit:

```
class C
{
    public static int i;
    public int j;
    public C()
    {
        i++; j=i;
    }
}
```

qu'affichera le code suivant?

```
C x = new C();      C y = new C();      C z = x;
System.out.println(z.i + " et " + z.j);
```

- a. 2 et 2
- b. 1 et 1
- c. 2 et 1
- d. 1 et 3

Exercices d'applications

Exercice 1

L'objectif de cet exercice est de modéliser à l'aide d'objets un ensemble de médias (disques, livres, etc.). Pour cela, nous disposons de la classe « Date » et de l'interface « Media » suivantes :

```
public class Date {
    private int jour, mois, annee;

    public Date (int jour, int mois, int annee) {
        this.jour = jour;
        this.mois = mois;
        this.annee = annee;
    }

    public String toString() {
        StringBuffer str = new StringBuffer() ;
        str.append(jour);
        str.append("/");
        str.append(mois);
        str.append("/");
        str.append(annee);
        return str.toString();
    }
}
```

```
public interface Média {
    public String getTitre();
    public String getAuteur();
    public double getPrix();
    public Date getDate() ;
}
```

1- Proposez une implémentation complète de la classe « Disque » qui implémente l'interface « Média » et qui possède la structure suivante :

Disque
<ul style="list-style-type: none"> - titre - auteur - prix - dateParution - nbPistes
<ul style="list-style-type: none"> - Disque(titre, auteur, prix, dateParution, nbPistes) - getNbPistes() - setNbPistes(nbPistes) - toString()

Remarque : la méthode « toString() » doit renvoyer une chaîne de la forme :

Disque[titre = « titre » auteur = « auteur » prix = « prix » dateParution =
« dateParution » nbPistes = « nbPistes »]

2- Proposez une implémentation complète de la classe « Livre » qui implémente l'interface « Media » et qui possède la structure suivante :

Livre
- titre - auteur - prix - dateParution - nbPages
- Livre(titre, auteur, prix, dateParution, nbPages) - getNbPages() - setNbPages(nbPages) - toString()

Remarque : la méthode « toString() » doit renvoyer une chaîne de la forme :

Livre[titre = « titre » auteur = « auteur » prix = « prix » dateParution =
« dateParution » nbPages = « nbPages »]

3- Etant donné la grande ressemblance entre les classes « Disque » et « Livre », on décide d'abandonner l'idée de l'interface au profit d'une classe abstraite « Media ».

- Proposer un diagramme de classes optimal
- Proposer une implémentation complète de la classe abstraite « Média »
- Proposer une implémentation complète de la nouvelle classe « Livre »

4- Créer une classe « Médiathèque » permettant de stocker un ensemble de médias dans un vecteur. Cette classe possède la structure suivante :

Médiathèque
- nom - capaciteMax - nbreMedias - vectMedias
- Mediatheque(nom, capaciteMax) - ajouter(Media m) - rechercher(String titre) - toString()

Remarques

- On suppose qu'une médiathèque nouvellement créée est toujours vide et que les médias seront ensuite ajoutés en faisant appel à la méthode « ajouter(Media m) »
- La méthode rechercher retourne la valeur « true » lorsqu'un titre figure dans la médiathèque sous n'importe quelle forme
- La méthode « toString() » doit renvoyer une chaîne de la forme :

```

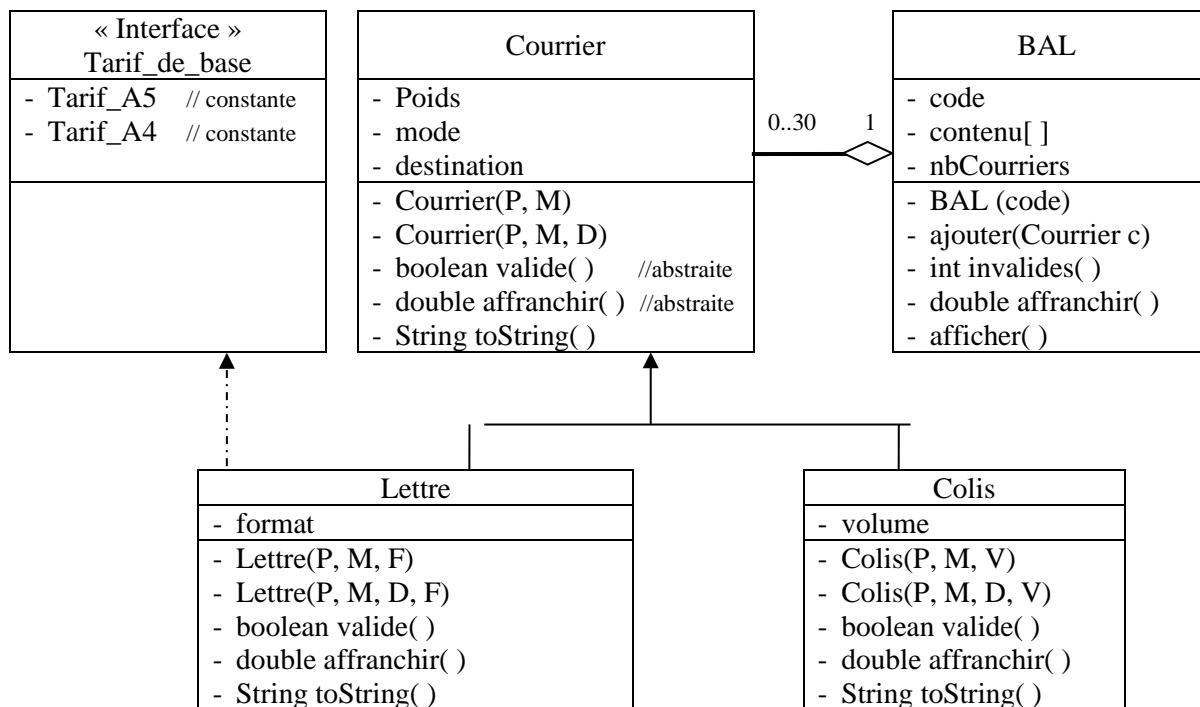
Médiathèque[Nom : « nom »    Nbre de médias : « nbreMedias »
Disque[Titre : « titre »    ...    ]
Livre[Titre : « titre »    ...    ]
...
]
```

5- Proposer le code d'une méthode « main() » permettant de :

- Créer une médiathèque « MédiaPlus »
- Ajouter à cette médiathèque deux disques et un livre
- Faire la recherche d'un disque dont on connaît le titre
- Afficher une description complète de la médiathèque.

Exercice 2

Il s'agit dans cet exercice de proposer une conception modélisant une boîte aux lettres en Java pour aider les employés de la poste à affranchir (timbrer) le courrier avec le montant approprié. Une boîte aux lettres recueille du courrier de différents types : des lettres, des colis, etc.



Quelque soit son type, un courrier est caractérisé par :

- son poids (en grammes)
- le mode d'expédition (Normal ou Express)
- son adresse de destination.

Une lettre est caractérisée en plus par son format ("A5" ou "A4").

Un colis est caractérisé en plus par son volume exprimé en litres (1 litre = 1 dm³).

Voici les règles utilisées pour affranchir le courrier :

- En mode d'expédition normal : le montant nécessaire pour affranchir une lettre dépend de son format et de son poids :

$$\text{montant} = \text{tarif de base} + 0.02 * \text{poids (grammes)}$$

- Le tarif de base est fixé actuellement à 0.350 D pour une lettre "A5" et 0.500 D pour une lettre "A4"
- Le montant nécessaire pour affranchir un colis dépend de son poids et de son volume :

$$\text{montant} = 0.2 * \text{volume (litres)} + 0.01 * \text{poids (grammes)}$$

- En mode d'expédition express : les montants précédents sont doublés, quelque soit le type de courrier ;
- Seul le courrier valide est affranchi. Le montant d'affranchissement de tout courrier invalide sera 0;
- Une lettre n'est pas valide si l'adresse de destination est vide (null);
- Un colis n'est pas valide si son adresse de destination est vide ou s'il dépasse un volume de 50 litres.

Travail demandé

Sachant que la méthode `toString()` retourne à chaque fois une description textuelle du courrier qui fournit le maximum d'information possible (avec une indication si le courrier est invalide).

1. Créer la classe « Courrier »
2. Créer l'interface « Tarif_de_base »
3. Créer la sous-classe « Lettre »
4. Créer la sous-classe « Colis »
5. Chaque boîte aux lettres est identifiée par un code unique et contient un ensemble de courriers de tout type (on suppose que le nombre total de courriers par boîte ne dépasse pas 30).

Créer la classe « BAL » sachant que :

- Chaque boîte créée est supposée initialement vide
- La méthode `ajouter()` permet d'ajouter un courrier à la boîte si elle n'est pas encore pleine
- La méthode `affranchir()` retourne le montant total d'affranchissement de tout le courrier valide présent dans la boîte aux lettres
- La méthode `afficher()` affiche une description complète de la boîte aux lettres et de son contenu (avec notamment le montant total d'affranchissement du courrier)
- La méthode `invalides()` retourne le nombre de courriers invalides présents dans la boîte.

6. Créer une classe « Poste » réduite à une méthode `main()` permettant de :

- Créer les 4 courriers suivants :
 - `lettre1 (50 gr, 'N', "Monastir", "A5");`
 - `lettre2 (100 gr, 'N', "A4");`
 - `colis1 (2000 gr, 'N', "Sfax", 15 litres);`
 - `Colis2 (2500 gr, 'N', "Nabeul", 70 litres);`
- Créer une boîte aux lettres que vous appelez « ZK01 »

- Ajouter les 4 courriers précédents à la boîte « ZK01 »
- Afficher une description complète de la boîte aux lettres « ZK01 » et de son contenu (avec notamment le nombre de courriers invalides et le montant total d'affranchissement du courrier).

Exercice 3

Le directeur d'une entreprise de produits chimiques souhaite gérer les salaires et les primes de ses employés au moyen d'un programme Java.

Un employé est caractérisé par son nom, son âge et sa date de recrutement dans l'entreprise (juste l'année).

Codez une classe abstraite Employé dotée des attributs nécessaires, d'une méthode abstraite calculSalaire() (ce calcul dépendra en effet de la catégorie de l'employé) et d'une méthode getNom() retournant une chaîne de caractère obtenue en concaténant la chaîne de caractères "L'employé " avec le nom. Dotez également votre classe d'un constructeur prenant en paramètre l'ensemble des attributs nécessaires.

Calcul du salaire

L'entreprise emploie trois catégories d'employés :

- Les vendeurs dont le salaire mensuel est égal à 20 % du *chiffre d'affaire* qu'ils réalisent mensuellement, plus 200 Dinars.
- Les techniciens dont le salaire vaut le *nombre d'unités* produites mensuellement multipliées par 1.5.
- Les manutentionnaires dont le salaire vaut leur *nombre d'heures* de travail mensuel multiplié par 5 Dinars.

Codez dans votre projet Java une hiérarchie de classes pour les employés en respectant les contraintes suivantes :

- La super-classe de la hiérarchie doit être la classe abstraite Employé.
- Les nouvelles classes doivent contenir les attributs qui leur sont spécifiques ainsi que le codage approprié des méthodes calculSalaire() et getNom() en changeant le mot "employé" par la catégorie de l'employé.
- Chaque sous-classe est dotée d'un constructeur prenant en argument l'ensemble des attributs nécessaires.

Employés à risques

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

Complétez votre projet en introduisant deux nouvelles sous-classes d'employés nommées TechARisque et ManutARisque et qui désigneront les employés des secteurs *production* et *manutention* travaillant avec des produits dangereux.

Ajouter également à votre programme une interface pour les *employés à risque* permettant de leur associer une *prime mensuelle* (fixée temporairement à 50 Dinars).

Collection d'employés

Satisfait de la hiérarchie proposée, le directeur de la société souhaite maintenant l'exploiter pour afficher le salaire de tous ses employés ainsi que le salaire moyen.

Ajoutez une classe `Personnel` contenant un vecteur pouvant contenir un nombre quelconque d'employés. Définissez ensuite les méthodes suivantes dans la classe `Personnel` :

- Un constructeur `personnel()`
- `void ajouterEmployé(Employé)` qui ajoute un employé au vecteur
- `void afficher()` qui affiche la liste des employés rangés dans le vecteur avec leurs salaires respectifs
- `double salaireMoyen()` qui retourne le salaire moyen des employés de l'entreprise.

Testez votre programme avec la méthode `main` suivante :

```
public class Salaires {

    public static void main(String[] args) {
        Personnel p = new Personnel();
        p.ajouterEmployé(new Vendeur("Ahmed", 45, 1995, 3000));
        p.ajouterEmployé(new Vendeur("Ali", 25, 2001, 2000));
        p.ajouterEmployé(new Technicien("Mohamed", 28, 1998, 1000));
        p.ajouterEmployé(new Manutentionnaire("Samir", 32, 1998, 45));
        p.ajouterEmployé(new TechARisque("Adel", 28, 2000, 1000));
        p.ajouterEmployé(new ManutARisque("Ridha", 30, 2001, 45));
        p.afficherSalaires();
        System.out.println("Le salaire moyen dans l'entreprise est de "
+        p.salaireMoyen() + " Dinars");
    }
}
```

Vous devriez obtenir un résultat semblable au suivant :

```
Le vendeur Ahmed gagne 800.0 Dinars
Le vendeur Ali gagne 600.0 Dinars
Le technicien Mohamed gagne 1500.0 Dinars
Le manutentionnaire Samir gagne 225.0 Dinars
Le technicien Adel gagne 1550.0 Dinars
Le manutentionnaire Ridha gagne 275.0 Dinars
Le salaire moyen dans l'entreprise est de 825.0 Dinars
```

Remarque : Il est conseillé de commencer par établir un diagramme de classes.

Exercice 4

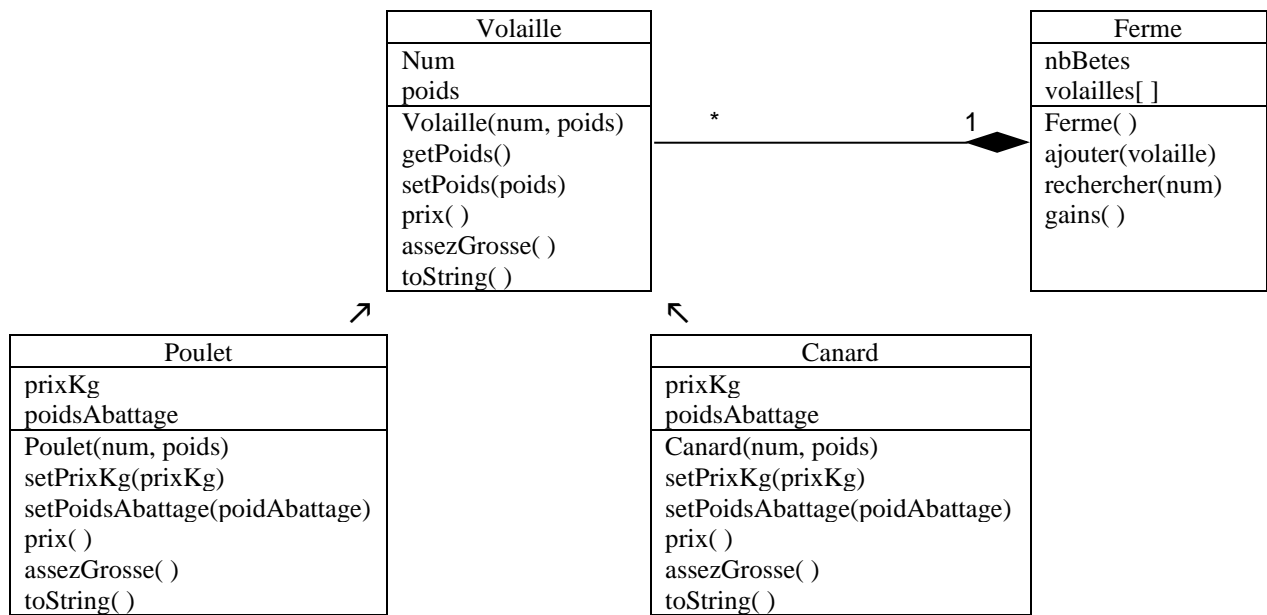
Un éleveur de volaille reçoit d'un fournisseur de jeunes poulets et de jeunes canards qu'il élève jusqu'à ce qu'ils aient la taille nécessaire à leur commercialisation (un certain poids exprimé en Kg).

Une volaille est caractérisée par un numéro d'identification reporté sur une bague qu'elle porte à sa patte droite et par son poids actuel constamment mis à jour. Les volailles arrivent à

l'élevage à l'âge de trois semaines. Elles sont baguées et enregistrées dans le système informatique.

Il y a deux sortes de volailles : les poulets et les canards. Le prix du poulet et du canard sont deux prix différents, exprimés en Dinars par kilo. En revanche, le prix est le même pour tous les individus de la même espèce. Ce prix varie chaque jour. Le poids auquel on abat les bêtes est également différent pour les poulets et les canards, mais c'est le même pour tous les poulets (respectivement, tous les canards).

La figure suivante montre un diagramme de classes simplifié.



Remarques

- Dans les classes « Poulet » et « Canard » la méthode `prix()` retourne le prix actuel de la bête (`prixKg*poids`).
- Dans les classes « Poulet » et « Canard » la méthode `assezGrosse()` retourne la valeur « true » dès que le poids actuel de la bête atteint le poids d'abattage.
- La méthode `toString()` retourne un texte qui a l'une des formes suivantes :

[Numéro : « num » Poids Actuel : « poids » Assez Petite]
 [Numéro : « num » Poids Actuel : « poids » Assez Grosse Prix : « prix »]

- Dans la classe « Ferme », le nombre de bêtes est initialement nul. Pour des raisons d'espace le nombre total de volailles ne peut en aucun cas dépasser 500.
- La méthode `rechercher(num)` vérifie si un volaille existe dans le tableau, si oui elle en affiche une description complète.
- La méthode `gains()` retourne la valeur totale en Dinars des volailles assez grosses élevées dans la ferme.

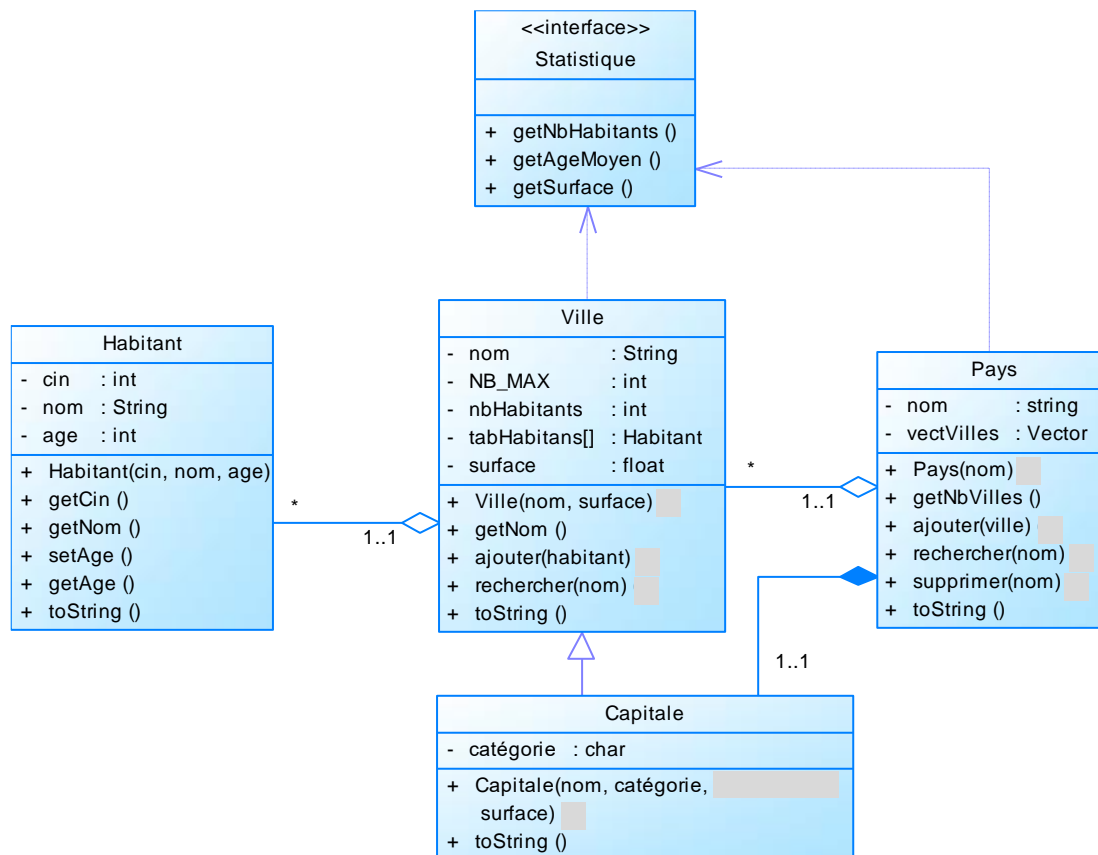
Travail demandé

1. Quel doit être le type de classe « Volaille » ? Justifier votre réponse.
2. Définir la classe « Volaille »
3. Définir la classe « Poulet »

4. Définir la classe « Canard »
5. Définir la classe « Ferme »
6. Définir une classe « TestFerme » réduite à une méthode main() permettant de :
 - Créer 2 poulets et 3 canards
 - Fixer le prix du poulet à 2D et celui du canard à 3D
 - Fixer le poids d'abattage des poulets à 1.2 kg et celui des canards à 1.5 kg
 - Créer une ferme « F » qui fait l'élevage des 5 bêtes précédentes
 - Rechercher une volaille quelconque à partir de son numéro
 - Afficher le gain espéré si le fermier décide de vendre les grosses volailles.

Exercice 5

Soit le diagramme de classe suivant :



1. Implémenter la classe « Habitant »
2. Implémenter l'interface « Statistique » sachant que :
 - `getNbHabitant()` retourne le nombre d'habitants d'une ville ou d'un pays
 - `getAgeMoyen()` permet de calculer l'âge moyen de tous les habitants d'une ville ou d'un pays. L'âge moyen pour un pays est la moyenne de tous les âges moyens des villes qui le composent
 - `getSurface()` la surface d'une ville ou d'un pays. La surface d'un pays est la somme des surfaces de toutes les villes qui le composent
3. Implémenter la classe « Ville ». Cette classe contient un tableau d'habitants dont la taille maximale est NB_MAX. La valeur de cette constante, fixée provisoirement à 500 000, est la même pour toutes villes.
 - La méthode « `rechercher (nom)` » doit retourner toutes les informations disponibles sur la personne recherchée ou le message « Habitant introuvable ».

- La méthode `toString()` doit retourner le nom, le nombre d'habitants, l'âge moyen, la surface et liste des habitants de la ville .
- 4. Implémenter la classe « Capitale ». l'attribut « catégorie » peut prendre l'une des valeurs 'P' (capitale politique), 'E' (capitale économique) ou 'T' (capitale politique et économique).
- 5. Implémenter la classe « Pays » sachant que :
 - La capitale politique du pays sera toujours mise en première position dans le vecteur des villes.
 - La méthode « rechercher (nom) » doit retourner l'indice de la ville ou -1 si la recherche est infructueuse.
 - La méthode `toString()` doit retourner le nom du pays, la surface, le nombre et l'âge moyen des habitants et une description détaillée des différentes villes (nom, nombre d'habitants, surface et liste des habitants).
- 6. Implémenter une classe « Test » réduite à une méthode « `main()` » permettant de :
 - Créer 5 habitants H1 ... H5, une capitale politique C, une ville V et un pays P composé de C et V.
 - Affecter les 3 premiers habitants à la ville V et les deux restants à la ville C.
 - Afficher une description détaillée du pays P.

Annexe : Principales méthodes de la classe « Vector »

Méthode	Rôle
<code>boolean isEmpty()</code>	Vérifie si le vecteur est vide
<code>int size()</code>	Retourner le nombre d'éléments de la liste
<code>boolean add(Object o)</code>	Ajouter l'objet o à la fin du vecteur.
<code>Object get(int index)</code>	Retourner l'élément à la position fournie en paramètre
<code>Object set(int index, Object o)</code>	Remplacer l'élément à la position fournie en paramètre
<code>boolean remove(Object o)</code>	supprimer la première occurrence de l'objet indiqué
<code>Object remove(int index)</code>	Supprimer l'élément à la position fournie en paramètre
<code>boolean removeAllElements()</code>	Vider le vecteur
<code>boolean contains(Object o)</code>	Retourne « true » si le vecteur contient l'objet o
<code>int indexOf(Object o)</code>	Retourner la première position dans la liste de l'élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé

Leçon N°7

La gestion des exceptions

Introduction

La notion d'exception est très importante en programmation. Une exception est une erreur qui se produit pendant l'exécution d'un programme et qui conduit le plus souvent à l'arrêt de l'exécution et l'affichage d'un message d'erreur sur la console.

Le fait de gérer les exceptions s'appelle aussi « la capture d'exception ». Le principe consiste à repérer un morceau de code (par exemple, une division par zéro) qui pourrait générer une exception, de capturer l'exception correspondante et enfin de la traiter, c'est-à-dire d'afficher un message personnalisé et de continuer l'exécution.

I. Le bloc try{...} catch{...}

Java contient une classe nommée **Exception** dans laquelle sont répertoriés différents cas d'erreur. La *division par zéro*, par exemple, en fait partie. Voici ce que donne l'exécution du code suivant :

```
public class Test{
    public static void main(String[] args) {
        int i = 20, j = 0;
        System.out.println(i/j);
        System.out.println("suite du programme");
    }
}
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at exceptions.Test.main(Test.java:5)
```

Remarquons que lorsque l'exception a été levée, le programme a été interrompu et que d'après le message affiché dans la console, le type de l'exception qui a été déclenchée est *ArithmeticException*. Voici comment capturer cette exception à l'aide d'un bloc try{...}/catch{...} et le résultat obtenu dans ce cas :

```
public class Test {
    public static void main(String[] args) {
        int i = 20, j = 0;
        try {
            System.out.println(i/j);
        }
        catch (ArithmeticException e){
            System.out.println("Attention : division par zéro !");
        }
        System.out.println("suite du programme");
    }
}
```

```
Attention : division par zéro !
suite du programme
```

Remarques

1. La capture de l'exception a permis d'éviter l'interruption de l'exécution du programme.
2. Les blocs try et catch doivent être contigus.
3. Le paramètre de la clause catch permet de connaître le type d'exception qui doit être capturé. Dans notre exemple, l'objet e peut servir, par exemple, pour afficher la nature de l'exception en écrivant :

```
System.out.println("Erreur ! "+e.getMessage());
ou e.printStackTrace();
```

II. Le bloc finally

Lorsqu'une ligne de code lève une exception, l'instruction dans le bloc try est interrompue et le programme se rend dans le bloc catch correspondant à l'exception levée. Si on souhaite effectuer une action, qu'une exception soit levée ou non (fermer un fichier, clore une connexion à une BD ou un socket (une connexion réseau), libérer une ressource...), il faut ajouter un bloc *finally* comme dans le cas suivant :

```
public class Test {
    public static void main(String[] args) {
        int i = 20, j = 0;
        try {
            System.out.println(i/j);
        }
        catch (ArithmeticException e){
            e.printStackTrace();
        }
        finally {
            System.out.println("fin du programme");
        }
    }
}
```

```
java.lang.ArithmeticException: / by zero
    at exceptions.Test.main(Test.java:6)
fin du programme
```

Le seul cas où le bloc finally n'est pas exécuté est lorsqu'il est précédé par une instruction *System.exit(arg)*. Cette dernière permet de sortir complètement du programme (un code retour différent de zéro indique une fin anormale).

III. Les exceptions personnalisées

Nous partons de l'exemple suivant pour expliquer la procédure de traitement des exceptions personnalisées. Considérons la classe « Point » munie d'un constructeur et de la méthode `afficher()` :

```
public class Point {
    private int x,y ; // coordonnées du point
    public Point (int x, int y) {
        this.x = x ;
        this.y = y ;
    }

    public void afficher () {
        System.out.println("Point["+x+ " , "+y+"]");
    }
}
```

Supposons dans notre exemple que nous ne manipulons que des points de coordonnées positives. Ce qui veut dire que lors de la création d'un point avec des coordonnées négatives, notre programme doit le rejeter tout en affichant un message d'erreur à l'utilisateur. Pour notre programme, la transmission au constructeur d'une valeur négative est considérée comme une **exception** que nous déclencherons à l'aide de l'instruction **throw**. À celle-ci, nous devons fournir un objet dont le type servira ultérieurement à identifier l'exception concernée. Nous créons donc (un peu artificiellement) une classe que nous nommerons *CoordException*. Java impose que cette classe dérive de la classe standard **Exception** (par convention le nom de toute exception se termine par « Exception »).

```
public class CoordException extends Exception {
    public CoordException(String message) {
        super(message);
    }
}
```

Pour lancer une exception de ce type au sein de notre constructeur, nous fournirons à l'instruction *throw* un objet de type *CoordException*, soit l'instruction :

```
throw new CoordException(arg);
```

En définitive, le constructeur de notre classe « Point » peut se présenter ainsi :

```
public Point (int x, int y) throws CoordException {
    if (x<0 || y<0) throw new CoordException("Coordonnées incorrectes");
    this.x = x ;
    this.y = y ;
}
```

Notons la présence de *throws CoordException* dans l'en-tête du constructeur qui précise que ce dernier est dangereux et qu'il est susceptible de déclencher une exception de type *CoordException*. Par conséquent, l'appel du constructeur ne se fera que dans un bloc try/catch.

En résumé, voici la définition complète de la classe « Point » avec un jeu de test :

```

public class Point {
    private int x,y ; // coordonnées du point
    public Point (int x, int y) throws CoordException {
        if (x<0 || y<0) throw new CoordException("Coordonnées incorrectes");
        this.x = x ;
        this.y = y ;
    }

    public void afficher () {
        System.out.println("(" + x + " , " + y + ")");
    }

    public static void main(String[] args) {
        try{
            Point P = new Point (2,3); P.afficher();
            Point Q = new Point(-3,5); Q.afficher();
        }
        catch (CoordException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

Résultats obtenus

```

(2,3)
Coordonnées incorrectes

```

IV. Gestion de plusieurs exceptions

IV.1 Exemple de gestion de deux Exceptions

Pour la même classe « Point », nous allons ajouter une méthode :

```
deplacer(int dx, int dy) thorws DeplException { ... }
```

qui fait déplacer un point donné de dx et de dy tout en prévoyant que ce déplacement doit garder les coordonnées du point positives. Dans le cas échéant, une exception *DeplException* se déclenchera pour empêcher le déplacement et avertit l'utilisateur que ce déplacement n'est pas autorisé.

Voici le code de la méthode `deplacer()` à ajouter dans la classe `Point` :

```

public void deplacer (int dx, int dy) throws DeplException {
    if (((x+dx)<0) || ((y+dy)<0))
        throw new DeplException("Erreur de déplacement") ;
    x += dx ; y += dy ; // si pas d'exception.
}

```

DeplException est une classe qui dérive également de la classe *Exception*. De ce fait, on la déclare comme suit :

```

package exceptions;
public class DeplException extends Exception {
    public DeplException(String message) {
        super(message);
    }
}

```

Evidement, dans la méthode `main()` nous allons ajouter un deuxième bloc `catch` (`DeplException`) sous l'autre bloc `catch` de détection de l'erreur de coordonnées « `CoordException` ».

La classe « `Point` » aura donc l'allure suivante :

```
package exceptions;
public class Point {
    private int x,y ; // coordonnées du point
    public Point (int x, int y) throws CoordException {
        if (x<0 || y<0) throw new CoordException("Coordonné(s) négative(s)");
        this.x = x ;
        this.y = y ;
    }

    public void déplacer (int dx, int dy) throws DeplException {
        if ((x+dx)<0) || ((y+dy)<0))
            throw new DeplException("Erreur de déplacement") ;
        x += dx ; y += dy ; // si pas d'exception.
    }

    public void afficher () {
        System.out.println("(" + x + " , " + y + ")");
    }

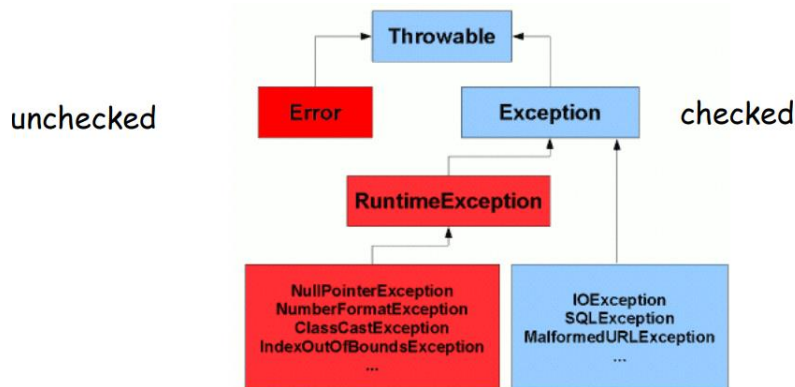
    public static void main(String[] args) {
        try{
            Point P = new Point (2,3); P.afficher();
            P.deplacer(-3,1);
            P.afficher();
            Point Q = new Point (-3,5); P.afficher();
        }
        catch (CoordException e) {
            System.out.println(e.getMessage());
        }
        catch (DeplException e){
            System.out.println(e.getMessage());
        }
    }
}
```

Résultats obtenus

(2,3)
Erreur de déplacement

Remarque : En présence de deux méthodes susceptibles de générer deux exceptions comme dans l'exemple précédent (`CoordException` et `DeplException`), nous devons obligatoirement traiter les deux dans la méthode `main()`, ce qui fait avoir les deux blocs `catch` après le bloc `try`. Bien entendu, comme la première exception déclenchée (`DeplException`) provoque la sortie du bloc `try` nous n'avons aucune chance de traiter la deuxième exception due à la création d'un point avec des coordonnées négatives. Le choix de l'ordre de traitement des exceptions est donc important (en général, il faut commencer par la plus spécifique).

V. Hiérarchie des exceptions



La classe *Throwable* est la classe mère de toutes les exceptions et erreurs : seules des instances de *Throwable* ou de ses classes dérivées peuvent être levées par l'instruction *throw* ou être argument d'un *catch*.

Une instance de *Throwable* peut avoir comme cause une autre instance de *Throwable* qui est à l'origine de la création de cette instance («chaînage des exceptions»).

En java, il existe des exceptions vérifiées ou contrôlées (checked) et des exceptions de types unchecked.

- Les exceptions surveillées (checked) : ce sont des cas exceptionnels, mais dont l'occurrence est prévue et peut être traitée (exemple: valeur en dehors des limites autorisées). Une méthode qui peut lancer une exception surveillée doit le déclarer (ou l'attraper par un catch)
- Les exceptions non surveillées (unchecked)
 - Runtime : par exemple un dépassement de tableau (peut être captée)
 - Error : il n'y a rien à faire dans ce cas (erreur interne)

Une *Error* est une exception qui indique des problèmes graves : une application ne résout pas ce genre de problème. Une méthode n'a pas à déclarer dans sa clause *throws* les *Error* qui ne seraient pas capturées.

Les exceptions levées par la machine virtuelle correspondent aux :

- Erreurs de compilation ou de lancement : *NoClassDefFoundError*, *ClassFormatError*
- problème d'entrée/sortie : *IOException*, *AWTException*
- problème de ressource : *OutOfMemoryError*, *StackOverflowError*
- des erreurs de programmation (runtime) : *NullPointerException*, *ArrayIndexOutOfBoundsException*, *ArithmeticException*, etc.

Exercice 1 (QCM)

1. Grâce à quel bloc d'instructions peut-on capturer des exceptions ?

- a. `try{...} catch{...}.`
- b. `try{...} catch{...}.`
- c. `try{...} catch{...}.`

2. Comment créer de nouveaux types d'exceptions ?

- a. En créant une classe héritée de la classe `exception`.
- b. En créant une classe héritée de la classe `Exception`.
- c. En créant une classe héritée de la classe `Error`.

3. Une classe personnalisée, `Err`, a été créée et elle est héritée de **Exception**.

Qu'est-ce qui ne va pas dans ce code ?

```
Public class Rectangle{
    private int Longueur = 0;
    private int largeur = 0;
    public rectangle(int L, int l) throw Err{
        if (L < 0 || l < 0)
            throws new Err();
        else
        {
            Longueur = L;
            largeur = l;
        }
    }
}
```

- a. Rien du tout.
- b. Les mots clé `throw` et `throws` ne sont pas à utiliser ici !
- c. Les mots clé `throw` et `throws` ont été intervertis.

4. Comment peut-on gérer plusieurs exceptions qui peuvent être déclenchées par le même morceau de code ?

- a. Avec plusieurs blocs `try{...}`.
- b. Grâce à plusieurs blocs `catch{...}`.
- c. On ne peut pas !

Exercice 2

Définir une classe `TestFato` réduite à une méthode `main` permettant de lire un entier puis afficher son factoriel à l'écran.

Les exceptions à prévoir sont :

- Cas d'un paramètre non entier
NB : la méthode `parseInt` lance dans ce cas une exception instance de la classe **`NumberFormatException`**.
- Cas d'un paramètre négatif.

Dans chacun de ces cas, l'erreur doit être signalée de façon précise à l'utilisateur.

Exercice 3 : Soit la classe Ville suivante :

Ville
- nom - nbHabitants - superficie
- Ville(nom, nbHabitants, superficie) - toString()

1. Créer la classe « Ville » en prévoyant deux exceptions :

- **NameException** : déclenchée si le nom de la ville comporte moins de 3 caractères
- **HabException** : déclenchée si le nombre d'habitants n'est pas positif

2. Créer une classe « TestVilles » réduite à une méthode main() qui crée 3 villes puis en affiche une description détaillée.

Exercice 4 : Soit la classe Temps suivante :

```
public class Temps {
    private int heures, minutes, secondes ;

    public Temps(int h, int m, int s) {
        heures = h ;
        minutes = m ;
        secondes = s ;
    }
    public static void main(String[] args ){
        Temps t = new Temps(24,12,67) ;
    }
}
```

- 1- Modifier le constructeur de cette classe de manière à ce qu'il lance une exception de type **TempsException** (qu'il ne traitera pas) si les heures, les minutes ou les secondes ne sont pas valides.
- 2- Modifier le code de la méthode main de manière à ce que l'exception **TempsException** (qui est une sous-classe de la classe Exception) soit traitée en affichant le message « **Temps invalide** » avec un arrêt du programme.

Exercice 5

1. Définir une classe **Pile** permettant de modéliser une pile contenant un nombre quelconque d'éléments de type entier et qui sera stockée sous forme d'un tableau.

Chaque pile contiendra les attributs suivants :

- **tailleMax** : taille maximale de la pile
- **indiceSommet** : indice du sommet actuel de la pile (contient la valeur -1 lorsque la pile est vide)
- **elements[]** : un tableau qui contient les éléments de la pile.

Les méthodes à prévoir sont :

- Un constructeur `Pile(n)` qui reçoit en argument la taille maximale de la pile
- `boolean pileVide()` qui retourne vrai si la pile est vide.
- `boolean pilePleine()` qui retourne vrai si le nombre d'éléments dans la pile a atteint la taille maximale.
- `void empiler(int e) throws PilePleineException`
Si la pile est pleine, cette méthode lance une exception, sinon elle met l'entier `e` à la fin de la pile
- `void dépiler() throws PileVideException`
Si la pile est vide, cette méthode lance une exception, sinon elle enlève le dernier élément de la pile (il suffit de décrémenter l'indice du sommet de la pile (suppression logique))
- `void afficher() throws PileVideException`
Si la pile est vide, cette méthode lance une exception, sinon elle affiche le contenu de la pile (bien sur dans l'ordre inverse de rangement dans le tableau).

2. Définir une classe **TestPile** réduite une méthode `main()` permettant de :

- Créer une pile `S` de taille maximale 10
- Empiler respectivement les éléments 10, 20, 30 et 40
- Dépiler le dernier élément
- Afficher le contenu de la pile.

Le résultat attendu est : 30 20 10

NB : Pour simplifier, on suppose que le traitement des exceptions se limite à l'affichage d'un message qui explique la cause de l'erreur sans arrêter l'exécution du programme.

Table des matières

	Page
Leçon N° 1 : Syntaxe du langage Java	1
Introduction	1
I. Notion de machine virtuelle	1
II. Structure d'un programme en Java	2
III. Les identificateurs	3
IV. Les types de données primitifs (ou de base)	4
V. Les variables et les constantes	4
V.1. Les variables	4
V.2. Les constantes	5
VI. Les expressions	5
VI.1. Les opérateurs usuels	5
VI.2. Les conversion de type (cast)	6
VII. Les structures de contrôle	7
VII.1. L'instruction if	7
VII.2. La structure switch	8
VII.3. La boucle while	9
VII.4. La boucle for	9
VII.5. La boucle do .. while	10
VIII. Les entrées/sorties	10
Exercices d'application	16
Leçon N°2 : Les tableaux en Java	17
I. Les tableaux simples	17
I.1. Définition	17
I.2. Accès aux éléments d'un tableau	18
II. Les tableaux à plusieurs dimensions	20
III. Les tableaux dynamiques (la classe Vector)	21
Leçon N°3 : Fondements de la Programmation Orientée Objet	25
I. Principe de base	25
II. Les avantages de la POO	26
III. Définition d'une classe	27
III.1. Déclaration des données membres	27
III.2. Déclaration des fonctions membres	28
III.3. La surcharge de méthode	29
IV. Les constructeurs	31
V. Les accesseurs et les mutateurs	31
VI. Les objets	33
VI.1. Instanciation	33
VI.2. Accès aux attributs et aux méthodes	33
VII. Les membres de classe	34
VIII. Le mot clé final	35
IX. Classe interne	36
Exercices d'application	38
Leçon 4 : Les chaînes de caractères	47
I. Les objets de la classe String	47
I.1. Instanciation d'un objet String	47
I.2. Principales méthodes de la classe String	48
I.2.1. Longueur d'une chaîne	48
I.2.2. Modification de la casse d'une chaîne	48
I.2.3. Accès à un caractère dans une chaîne	49
I.2.4. Extraction d'une sous-chaîne	49
I.2.5. Recherche d'une sous-chaîne	50

I.3. Concaténation de chaînes de caractères	51
I.4. Egalité de deux chaînes	52
I.5. La méthode toString()	53
Exercices d'application	57
Leçon N°5 : Héritage et polymorphisme	59
I. Notion d'héritage	59
II. Implémentation de l'héritage en java	59
III. Le polymorphisme	64
Exercices d'application	68
Leçon N°6 : Interfaces et classes abstraites	73
I. Les interfaces	73
II. Les classes abstraites	76
Exercices d'application	82
Leçon N°7 : La gestion des exceptions	91
Introduction	91
I. Le bloc try / catch	91
II. Le bloc finally	92
III. Les exceptions personnalisées	93
IV. Gestion de plusieurs exceptions	94
V. Hiérarchie des exceptions	96

Solution des exercices (Syntaxe du langage Java)

Exercice 1

```
import javax.swing.JOptionPane;
public class Equald {
    public static void main(String[] args) {
        String sa, sb;
        double a, b;
        sa = JOptionPane.showInputDialog("Entrer a :");
        a = Double.parseDouble(sa);
        sb = JOptionPane.showInputDialog("Entrer b :");
        b = Double.parseDouble(sb);
        if (a == 0)
            if (b == 0)
                JOptionPane.showMessageDialog(null, "Infinité de solutions");
            else
                JOptionPane.showMessageDialog(null, "Pas de solution");
        else
            JOptionPane.showMessageDialog(null, "x = "+(-b/a));
    }
}
```

Exercice 2

```
import javax.swing.JOptionPane;
public class Equa2d {
    public static void main(String[] args) {
        String sa, sb, sc;
        double a, b, c, delta;
        sa = JOptionPane.showInputDialog("Entrer a :");
        a = Double.parseDouble(sa);
        sb = JOptionPane.showInputDialog("Entrer b :");
        b = Double.parseDouble(sb);
        sc = JOptionPane.showInputDialog("Entrer c :");
        c = Double.parseDouble(sc);
        delta = b*b - 4 * a * c;
        if (delta < 0)
            JOptionPane.showMessageDialog(null, "Aucune solution");
        else
            if (delta == 0)
                JOptionPane.showMessageDialog(null, "x = "+(-b)/(2*a));
            else
                JOptionPane.showMessageDialog(null, "x1 = "+(-b-Math.sqrt(delta))/(2*a)+
                    "\nx2=" +(-b+Math.sqrt(delta))/(2*a));
    }
}
```

Exercice 3

```
import javax.swing.*;
public class PGCD {
    public static void main(String[] args) {
        String nombre1, nombre2;
        int a, b;
        nombre1 = JOptionPane.showInputDialog("Entrer le 1er nombre :");
        a = Integer.parseInt(nombre1);
        nombre2 = JOptionPane.showInputDialog("Entrer le 2eme nombre :");
        b = Integer.parseInt(nombre2);
        while (a != b) {
            if (a > b)
```

```

        a = a - b;
    else
        b = b - a;
    }
    JOptionPane.showMessageDialog(null, "PGCD = "+a);
}
}

```

Exercice 4

```

import javax.swing.JOptionPane;
public class Conversion {
    public static void main(String[] args) {
        String sc, sf, choix ;
        double c, f;
        int choix;
        do {
            choix = JOptionPane.showInputDialog("1. °C --> F\n2. F --> °C\n3. Fin");
            choix = Integer.parseInt(choix);
            switch(choix){
                case 1:
                    sc = JOptionPane.showInputDialog("Entrer la temp en °C :");
                    c = Double.parseDouble(sc);
                    f = c * (9.0/5) + 32;
                    JOptionPane.showMessageDialog(null, "Temp en °F = "+f);
                    break;
                case 2:
                    sf = JOptionPane.showInputDialog("Entrer la temp en °F :");
                    f = Double.parseDouble(sf);
                    c = (f - 32)*(5.0/9);
                    JOptionPane.showMessageDialog(null, "Temp en °C = "+c);
                    break;
            }
        } while (choix != 3);
    }
}

```

Solution des exercices (les tableaux)

```
import javax.swing.JOptionPane;
public class TabNotes {
    public static void main(String[] args) {
        double tabNotes[] = new double[5];
        String note;
        double SommeNotes = 0;
        int i;
        for (i = 0; i < tabNotes.length; i++)
        {
            do{
                note = JOptionPane.showInputDialog("Entrer une note :");
                tabNotes[i] = Double.parseDouble(note);
            } while ((tabNotes[i] < 0) || (tabNotes[i] > 20));
            SommeNotes += tabNotes[i];
        }
        double m = SommeNotes/5;
        JOptionPane.showMessageDialog(null, "moyenne =" + m);
        int nb = 0;
        for (double t:tabNotes){
            if (t > m)
                nb++;
        }
        JOptionPane.showMessageDialog(null, " Nbre de notes au-dessus de la
        moyenne = " + nb);
    }
}
```


Solution des exercices (fondements de la POO)

Exercice 1

```
public class Employe {
    private int matricule;
    private String nom;
    private double salBase;
    private double prime;
    private double retenue;

    // définition d'un constructeur
    public Employe (int matricule, String nom, double salBase, double
prime, double retenue){
        this.matricule = matricule;
        this.nom = nom;
        this.salBase = salBase;
        this.prime = prime;
        this.retenue = retenue;
    }

    public double calculSalaire(){
        return salBase + prime - retenue;
    }

    public void afficher(){
        System.out.println("Matricule = "+matricule);
        System.out.println("Nom = "+nom);
        System.out.println("Salaire Net = "+calculSalaire());
    }

    public static void main(String[] args) {
        Employe E = new Employe(2525,"Ali Ben Salah",450, 150, 100);
        E.afficher( );
    }
}
```

Output du programme

```
Matricule = 2525
Nom  = Ali Ben Salah
Salaire Net = 500.0
```

Exercice 2

```
public class Rectangle {

    private double longueur, largeur;    // en cm

    public Rectangle(double longueur, double largeur) {
        this.longueur = longueur;
        this.largeur = largeur;
    }

    public double perimetre(){
        return (longueur + largeur) * 2;
    }

    public double surface()      {
        return longueur * largeur;
    }
}
```

```

    void afficher(){
        System.out.println("longueur = "+longueur);
        System.out.println("largeur = "+largeur);
        System.out.println("Périmètre = "+perimetre()+" cm");
        System.out.println("Surface = "+surface()+" cm2");
    }

    public static void main(String[] args) {
        Rectangle R = new Rectangle(5,4);
        R.afficher();
    }
}

```

Output du programme

```

longueur = 5.0
largeur = 4.0
Périmètre = 18.0 cm
Surface = 20.0 cm2

```

Exercice 3

```

public class CompteBancaire {
    int numCompte;
    String nomClient;
    double solde;

    // définition d'un constructeur
    public CompteBancaire (int numCompte, String nomClient, double solde){
        this.numCompte = numCompte;
        this.nomClient = nomClient;
        this.solde = solde;
    }

    public void deposer(double montant){
        solde = solde + montant;
    }

    public void retirer(double montant){
        if (montant <= solde)
            solde = solde - montant;
        else
            System.out.println("Solde insuffisant ...");
    }

    public void virer(double montant, CompteBancaire C){
        if (montant <= solde){
            this.retirer(montant);
            C.deposer(montant);
        }
        else
            System.out.println("virement impossible ...");
    }

    public void consulter() {
        System.out.print("Numéro du compte = "+numCompte);
        System.out.print("\tNom du client = "+nomClient);
        System.out.println("\tSolde = "+solde+" Dinars");
    }

    public static void main(String[] args) {
        CompteBancaire A = new CompteBancaire(1041,"Ali B. Salah", 450);
    }
}

```

```

        CompteBancaire B = new CompteBancaire(1042,"Med B. Salah", 300);
        A.deposer(250);
        A.retirer(200);
        A.virer(100, B);
        A.consulter();
        B.consulter();
    }
}

```

Output du programme

```

Numéro du compte = 1041 Nom du client = Ali B. Salah Solde = 400.0 Dinars
Numéro du compte = 1042 Nom du client = Med B. Salah Solde = 400.0 Dinars

```

Exercice 4

```

import javax.swing.JOptionPane;
public class Date {
    private int j, m, a;

    private Date(int j, int m, int a) {
        this.j = j;
        this.m = m;
        this.a = a;
    }

    public int nbJours(){
        if ((m==4) || (m==6) || (m==9) || (m==11))
            return 30;
        else
            if ((m==1) || (m==3) || (m==5) || (m==7) || (m==8) || (m==10) || (m==12))
                return 31;
            else {
                if ((a%400 == 0) || ((a%4==0) && (a%100!=0)))
                    return 29;
                else
                    return 28;
            }
    }

    public boolean valide (){
        if ((j > 0) && (j <= nbJours()) && (m>=1) && (m<=12) && (a>0))
            return true;
        else
            return false;
    }

    public Date lendemain(){
        if (j < nbJours())
            j++;
        else{
            if (m == 12){
                m = 1;
                a++;
            }
            else
                m++;
            j = 1;
        }
        return new Date(j,m,a);
    }
}

```

```

    }

    public String toString(){
        return j+"/"+m+"/"+a;
    }

    public static void main(String[] args) {
        int j, m, a;
        String sj, sm, sa;
        sj = JOptionPane.showInputDialog("Entrer le jour :");
        j = Integer.parseInt(sj);
        sm = JOptionPane.showInputDialog("Entrer le mois :");
        m = Integer.parseInt(sm);
        sa = JOptionPane.showInputDialog("Entrer l'année :");
        a = Integer.parseInt(sa);
        Date D1 = new Date(j,m,a);
        if (D1.valide()) {
            Date D2 = D1.lendemain();
            JOptionPane.showMessageDialog(null,"Lendemain : "+D2.toString());
        }
        else
            JOptionPane.showMessageDialog(null,"date invalide");
    }
}

```

Exercise 5

```

public class Time {
    private int heures, minutes, secondes;

    public Time(){
        heures = 0;
        minutes = 0;
        secondes = 0;
    }

    public Time(int heures, int minutes, int secondes) {
        this.heures = heures;
        this.minutes = minutes;
        this.secondes = secondes;
    }

    public int getHeures(){
        return heures;
    }

    public int getMinutes(){
        return minutes;
    }

    public int getSecondes(){
        return secondes;
    }

    public void setHeures(int hr){
        heures = hr;
    }

    public void setMinutes(int min){
        minutes = min;
    }
}

```

```

public void setSecondes(int sec) {
    secondes = sec;
}

public void ajouterHeures(int hr) {
    heures = (heures + hr) % 24;
}

public void ajouterMinutes(int min) {
    minutes = minutes + min;
    if (minutes >= 60) {
        heures = (heures + minutes / 60) % 24;
        minutes = minutes % 60;
    }
}

public void ajouterSecondes(int sec) {
    secondes = secondes + sec;
    if (secondes >= 60) {
        minutes = minutes + secondes / 60;
        secondes = secondes % 60;
    }
    if (minutes >= 60) {
        heures = (heures + minutes / 60) % 24;
        minutes = minutes % 60;
    }
}

public String toString() {
    return heures+":"+minutes+":"+secondes;
}
}

```

```

public class TestTime {

    static Time somme(Time t1, Time t2) {
        Time t3 = t1;
        t3.ajouterHeures(t2.getHeures());
        t3.ajouterMinutes(t2.getMinutes());
        t3.ajouterSecondes(t2.getSecondes());
        return t3;
    }

    public static void main(String[] args) {
        Time t1 = new Time(22, 58, 50);
        Time t2 = new Time(0, 1, 2);
        t1.setSecondes(57);
        t1.ajouterHeures(1);
        Time t3 = somme(t1, t2);
        t3.ajouterSecondes(1);
        System.out.println("t3 = " + t3.toString());
    }
}

```

Output du programme

```
t3 = 0:0:0
```

Exercice 6

```

public class Adresse {
    int numéro;
    String rue;
}

```

```

    String ville;

    public Adresse(int numéro, String rue, String ville){
        this.numéro = numéro;
        this.rue = rue;
        this.ville = ville;
    }

    public String toString(){
        return "Adresse : "+numéro+", rue "+rue+", "+ville;
    }
}

```

```

public class Client {
    private int numCli;
    private String nomCli;
    private Adresse adrCli;
    private int telCli;
    public Client(int numCli, String nomCli, Adresse adrCli, int telCli) {
        this.numCli = numCli;
        this.nomCli = nomCli;
        this.adrCli = adrCli;
        this.telCli = telCli;
    }

    public String toString(){
        return "Num Client : "+ numCli+"\nNom Client :+nomCli+"\n"+adrCli.toString()
        +"\nTel : "+telCli;
    }
}

```

```

import javax.swing.JOptionPane;
public class Compte {
    private int numCompte;
    private Client client;
    private float solde;

    private Compte(int numCompte, Client client, float solde) {
        this.numCompte = numCompte;
        this.client = client;
        this.solde = solde;
    }

    public void consulter(){
        JOptionPane.showMessageDialog(null, "Num Compte = "+numCompte+
        "\n"+client.toString()+"\nSolde : "+solde + " D");
    }

    public static void main(String[] args) {
        Client ABS = new Client(1111,"Ali Ben Salah", new Adresse(17, "Ibn Rochd",
        "Sousse"), 94105665);
        Client KBA = new Client(2222,"Karim Ben Amor", new Adresse(18, "Ibn Sina",
        "Monastir"), 41646605);
        Compte C1 = new Compte(1041, ABS, 450);
        Compte C2 = new Compte(1042, KBA, 500);
        Compte C3 = new Compte(1043, KBA, 600);

        Compte tabCompte[] = new Compte[3];
        tabCompte[0]= C1;
        tabCompte[1]= C2;
        tabCompte[2]= C3;
        for (Compte c:tabCompte) {
            c.consulter();
        }
    }
}

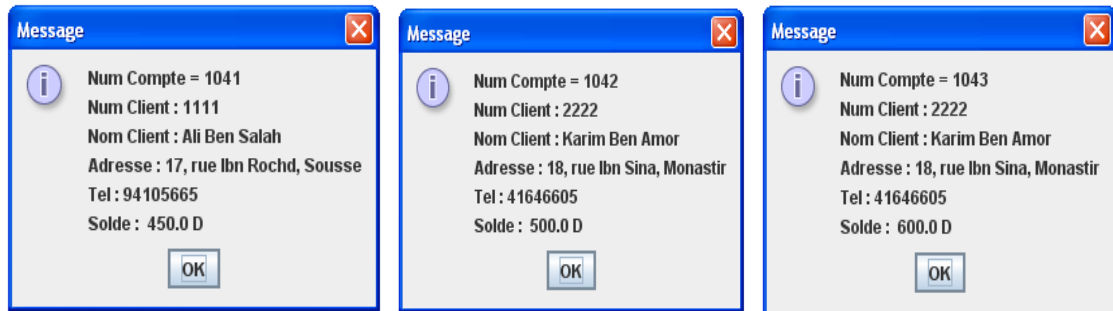
```

```

    }
}

```

Output du programme



Exercice 7

```

public class Complexe {
    private double réel;
    private double imag;

    public Complexe() {
        réel = 0;
        imag = 0;
    }

    public Complexe(double a, double b) {
        réel = a;
        imag = b;
    }

    public double getRéel() {
        return réel;
    }

    public double getImag() {
        return imag;
    }

    public double module() {
        return Math.sqrt(réel*réel+imag*imag);
    }

    public String toString() {
        return réel+" "+imag+"i";
    }
}

```

```

public class TestComplexe {

    static Complexe somme(Complexe u, Complexe v) {
        double réel = u.getRéel() + v.getRéel();
        double imag = u.getImag() + v.getImag();
        return new Complexe(réel, imag);
    }

    static Complexe produit(Complexe u, Complexe v) {
        double réel = u.getRéel()*v.getRéel() - u.getImag()*v.getImag();
        double imag = u.getRéel()*v.getImag() + u.getImag()*v.getRéel();
        return new Complexe(réel, imag);
    }
}

```

```

    }
    public static void main(String[] args){
        Complexe c1 = new Complexe(2,2);
        Complexe c2 = new Complexe(1,3);
        System.out.println("|C2|="+c2.module());
        Complexe c3 = somme(c1,c2);
        System.out.println("c3 = c1 + c2 = "+c3.toString());
        Complexe c4 = produit(c1,c2);
        System.out.println("c4 = c1 * c2 = "+c4.toString());
    }
}

```

Output du programme

```

|C2|=3.1622776601683795
c3 = c1 + c2 = 3.0+5.0i
c4 = c1 * c2 = -4.0+8.0i

```

Exercice 8

```

public class Projet {
    private String nomProj;
    private double budget;

    public Projet(String nomProj, double budget) {
        this.nomProj = nomProj;
        this.budget = budget;
    }

    public double getBudget() {
        return budget;
    }

    public String toString() {
        return "Projet [nomProj=" + nomProj + ", budget=" + budget + "]";
    }
}

```

```

public class Service {
    private String nomServ;
    private String responsable;
    private int nbrProj;
    private Projet projets[];

    public Service(String nomServ, String responsable) {
        this.nomServ = nomServ;
        this.responsable = responsable;
        this.nbrProj = 0;
        projets = new Projet[5];
    }

    public void ajouterProjet(Projet p){
        projets[nbrProj]=p;
        nbrProj++;
    }

    public double totalBudgets(){
        double tot = 0;
        for (int i = 0; i < nbrProj; i++)
            tot += projets[i].getBudget();
        return tot;
    }
}

```



```

    public String toString() {
        String s = "Service [nomServ=" + nomServ + ", responsable=" +
            responsable + ", nbrProj=" + nbrProj;
        for (int i = 0; i < nbrProj; i++){
            s+="\n"+projets[i].toString();
        }
        s+="\nBudget total="+totalBudgets()+"\n";
        return s;
    }
}

```

```

public class TestProj {
    public static void main(String[] args) {
        Projet P1 = new Projet("Projet 1", 100000);
        Projet P2 = new Projet("Projet 2", 200000);
        Projet P3 = new Projet("Projet 3", 300000);
        Projet P4 = new Projet("Projet 4", 400000);
        Projet P5 = new Projet("Projet 5", 500000);

        Service S1 = new Service("Technique", "Med");
        Service S2 = new Service("Production", "Salah");

        S1.ajouterProjet(P1); S1.ajouterProjet(P2);
        S2.ajouterProjet(P3); S2.ajouterProjet(P4); S2.ajouterProjet(P5);

        System.out.println(S1.toString());
        System.out.println();
        System.out.println(S2.toString());
    }
}

```

Output du programme

```

Service [nomServ=Technique, responsable=Med, nbrProj=2
Projet [nomProj=Projet 1, budget=100000.0]
Projet [nomProj=Projet 2, budget=200000.0]
Budget total=300000.0]

Service [nomServ=Production, responsable=Salah, nbrProj=3
Projet [nomProj=Projet 3, budget=300000.0]
Projet [nomProj=Projet 4, budget=400000.0]
Projet [nomProj=Projet 5, budget=500000.0]
Budget total=1200000.0]

```

Exercice 9

```

public class Etudiant {
    private int matricule;
    private String nom;
    private String prénom;
    private String adresse;

    public Etudiant(int matricule, String nom, String prénom){
        this.matricule = matricule;
        this.nom = nom;
        this.prénom = prénom;
    }
    public Etudiant(int matricule, String nom, String prénom, String adresse){
        this(matricule, nom, prénom);
        this.adresse = adresse;
    }
}

```

```

    public String toString(){
        return "l'étudiant " + prénom + " " + nom + " de matricule " +
            matricule + " qui habite à " + adresse;
    }
}

```

```

public class Livre {
    private int numLiv;
    private String titre;
    private String auteur;
    private boolean disponible;
    private Etudiant emprunteur;

    Livre(int numLiv, String titre, String auteur){
        this.numLiv = numLiv;
        this.titre = titre;
        this.auteur = auteur;
        this.disponible = true;
        this.emprunteur = null;
    }

    public String toString(){
        String message;
        message = "Le livre de numLiv "+numLiv+ " ayant pour titre
"+titre
        +" écrit par "+auteur;
        if (disponible)
            message = message + " est disponible";
        else
            message = message + " est emprunté par " +
emprunteur.toString();
        return message;
    }

    public int getnumLiv(){
        return numLiv;
    }

    public boolean getDisponible(){
        return disponible;
    }

    public void emprunter(Etudiant e){
        disponible = false;
        emprunteur = e;
    }

    public void retourner(){
        disponible = true;
        emprunteur = null;
    }
}

```

```

public class Bibliothèque {
    private String nomBib;
    private int nbLiv;
    private Livre tabLiv[];
    public Bibliothèque(String nom){
        nomBib = nom;
        nbLiv = 0;
        tabLiv = new Livre[1000];
    }
}

```

```

    public void ajouterLivre(Livre liv){
        tabLiv[nbLiv] = liv;
        nbLiv++;
    }

    public Livre[] getLivres(){
        return tabLiv;
    }

    public void rechercherLivre(int numLiv){
        boolean trouvé = false;
        int i=0;
        while ((i<nbLiv) && (!trouvé)){
            if (tabLiv[i].getnumLiv() == numLiv){
                System.out.println(tabLiv[i].toString());
                trouvé = true;
            }
            i++;
        }
        if (!trouvé)
            System.out.println("Livre introuvable...");
    }

    public int compter(){
        int nb = 0;
        for (int i=0; i<nbLiv; i++){
            if (tabLiv[i].getDisponible())
                nb++;
        }
        return nb;
    }
}

```

```

public class TestBib {
    public static void main(String[] args) {
        Etudiant E1 = new Etudiant(1011, "Ben Salah", "Ali", "Sousse");
        Etudiant E2 = new Etudiant(1011, "Ben Amor", "Walid");

        Livre L1 = new Livre(4921, "Penser en Java", "B. Eckel");
        Livre L2 = new Livre(4922, "BD Relationnelles", "G. Gardarin");
        Livre L3 = new Livre(4923, "UML2", "C. Soutou");

        Bibliothèque B = new Bibliothèque("Almaarif");

        B.ajouterLivre(L1);
        B.ajouterLivre(L2);
        B.ajouterLivre(L3);

        B.getLivres()[1].emprunter(E1);
        B.getLivres()[2].emprunter(E2);

        B.getLivres()[2].retourner();
        System.out.println("Nbre de livres disponibles : "+B.compter());
        B.rechercherLivre(4922);
    }
}

```

Output du programme

```
Nbre de livres disponibles : 2
```

Le livre de numLiv 4922 ayant pour titre BD Relationnelles écrit par G. Gardarin est emprunté par l'étudiant Ali Ben Salah de matricule 1011 qui habite à Sousse

Exercice 10

La lettre $c \in e[]$ ssi $e[c-97] = \text{true}$

```
public class EnsembleLettres {
    private boolean e[];

    public EnsembleLettres() {
        e = new boolean[26];
        for (int i = 0; i < 26; i++)
            e[i] = false;
    }

    public boolean[] getEns() {
        return e;
    }

    public void ajouter(char c) {
        e[c-97] = true;
    }

    public void enlever(char c) {
        e[c-97] = false;
    }

    public void afficher() {
        System.out.print("{");
        for (int i = 0; i < 26; i++)
            if (e[i])
                System.out.print((char) (97+i)+",");
        System.out.println("}");
    }

    public boolean estVide() {
        for (int i = 0; i < 26; i++)
            if (e[i])
                return false;
        return true;
    }

    public int cardinal() {
        int n = 0;
        for (int i = 0; i < 26; i++)
            if (e[i])
                n++;
        return n;
    }
}
```

```

public boolean appartient(char lettre){
    if (e[lettre-97])
        return true;
    else
        return false;
}
}

```

```

public class TestEnsLettres {

    public static EnsLettres intersection(EnsLettres e1, EnsLettres
e2){
        EnsLettres e3 = new EnsLettres();
        for (int i = 0; i < 26; i++)
            if ((e1.getEns()[i]) && (e2.getEns()[i]))
                e3.getEns()[i] = true;
        return e3;
    }

    public static EnsLettres union(EnsLettres e1, EnsLettres e2){
        EnsLettres e3 = new EnsLettres();
        for (int i = 0; i < 26; i++)
            if ((e1.getEns()[i]) || (e2.getEns()[i]))
                e3.getEns()[i] = true;
        return e3;
    }

    public static EnsLettres différence(EnsLettres e1, EnsLettres e2){
        EnsLettres e3 = new EnsLettres();
        for (int i = 0; i < 26; i++)
            if ((e1.getEns()[i]) && (!e2.getEns()[i]))
                e3.getEns()[i] = true;
        return e3;
    }

    public static boolean egaux(EnsLettres e1, EnsLettres e2){
        for (int i = 0; i < 26; i++)
            if (e1.getEns()[i] != e2.getEns()[i])
                return false;
        return true;
    }

    public static boolean estInclus(EnsLettres e1, EnsLettres e2){
        return différence(e1, e2).estVide();
    }

    public static void main(String[] args) {
        EnsLettres e1 = new EnsLettres();
        e1.ajouter('a');
        e1.ajouter('b');
        System.out.print("e1 = "); e1.afficher();

        EnsLettres e2 = new EnsLettres();
        e2.ajouter('b');
        e2.ajouter('c');
        System.out.print("e2 = "); e2.afficher();

        EnsLettres e3 = intersection(e1, e2);
        System.out.print("Intersection : "); e3.afficher();
    }
}

```

```

        EnsLettres e4 = union(e1,e2);
        System.out.print("Union : "); e4.afficher();

        EnsLettres e5 = difference(e1,e2);
        System.out.print("Différence : "); e5.afficher();

        System.out.println("e1 = e2 ? : "+egaux(e1,e2));
        System.out.println("e1 inclus dans e2 ? : 
"+estInclus(e1,e2));
    }
}

```

Output du programme

```

e1 = {a,b,}
e2 = {b,c,}
Intersection : {b,}
Union : {a,b,c,}
Différence : {a,}
e1 = e2 ? : false
e1 inclus dans e2 ? : false

```

Solution des exercices (Les chaines de caractères)

Solution

```

import javax.swing.JOptionPane;
public class TestChaines {

    public static String inverse(String s){
        String r = "";
        for (int i = 0; i < s.length();i++)
            r=s.charAt(i)+r;
        return r;
    }

    public static boolean palind(String s){
        String s_inv = inverse(s);
        if (s.equals(s_inv))
            return true;
        else
            return false;
    }

    public static int compter(char c, String s){
        int nb = 0;
        for (int i = 0; i < s.length(); i++)
            if (s.charAt(i) == c)
                nb++;
        return nb;
    }

    public static int compterMots(String s){
        if (s.length() > 0)
            return compter(' ',s)+1;
        else
            return 0;
    }

    public static void main(String[] args) {

```

```

        String chn = JOptionPane.showInputDialog("Entrer une chaine");
        if (palind(chn))
            JOptionPane.showMessageDialog(null, "Palindrome");
        else
            JOptionPane.showMessageDialog(null, "Non Palindrome");
        JOptionPane.showMessageDialog(null, "Nb de a : "+compter ('a', chn));
        JOptionPane.showMessageDialog(null, "Nb de mots : "+compterMots(chn));
    }
}

```

Exercice 2

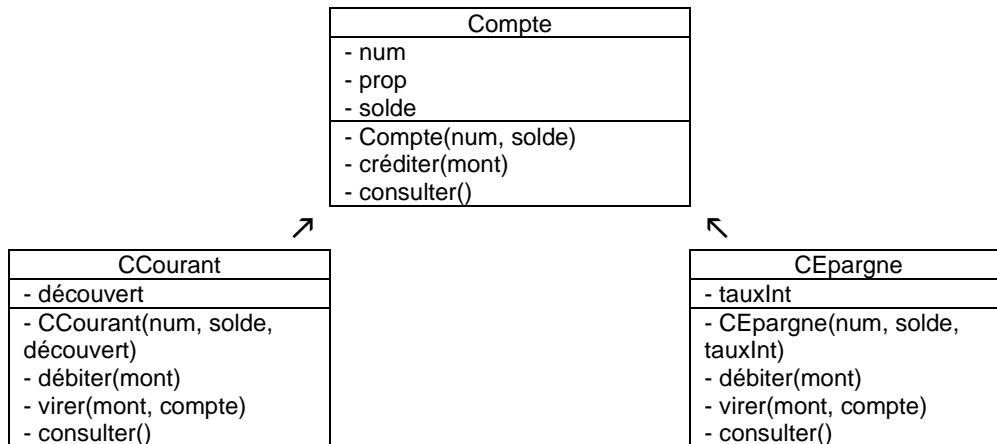
```

import javax.swing.JOptionPane;
public class AffichageIndenté {
    public static void main(String[] args) {
        String monTexte = JOptionPane.showInputDialog("Entrer une chaine : ");
        for (int i = 1; i <= monTexte.length(); i++)
            JOptionPane.showMessageDialog(null, monTexte.substring(0,i));
    }
}

```

Solution des exercices (héritage)

Solution (Exo comptes)



```

public class Compte {
    protected int num;
    protected String prop;
    protected double solde;

    public Compte (int num, String prop, double solde){
        this.num = num;
        this.prop = prop;
        this.solde = solde;
    }

    public void créditer(double montant){
        solde = solde + montant;
    }

    public void consulter() {
        System.out.println("Num compte = "+num);
        System.out.println("Propriétaire = "+prop);
        System.out.println("Solde = "+solde+" Dinars");
    }
}

```

```

public class CCourant extends Compte {
    private double découvert;

    public CCourant(int num, String prop, double solde, double découvert){
        super(num, prop, solde);
        this.découvert = découvert;
    }

    public void débiter(double montant) {
        if (montant <= (solde+découvert))
            solde = solde - montant;
        else
            System.out.println("Solde insuffisant ... ");
    }

    public void virer(double montant, Compte autre){
        if (montant < (solde+découvert))
        {
            this.débiter(montant);
            autre.créditer(montant);
        }
        else

```



```

        System.out.println("Solde insuffisant ... ");
    }
    public void consulter() {
        super.consulter();
        System.out.println("découvert = "+découvert+" Dinars");
    }
}

```

```

public class CEpargne extends Compte {
    private double tauxInt;

    public CEpargne(int num, String prop, double solde, double tauxInt){
        super(num, prop, solde);
        this.tauxInt = tauxInt;
    }

    public void débiter(double montant) {
        if (montant <= solde)
            solde = solde - montant;
        else
            System.out.println("Solde insuffisant ... ");
    }

    public void virer(double montant, Compte autre){
        if (montant < solde)
        {
            this.débiter(montant);
            autre.créditer(montant);
        }
        else
            System.out.println("Solde insuffisant ... ");
    }

    public double calculInteret(){
        return solde * tauxInt;
    }

    public void consulter() {
        super.consulter();
        System.out.println("taux d'interet : "+tauxInt);
        System.out.println("Monatant des interets : "+calculInteret()+"
Dinars");
    }
}

```

```

public class TestComptes {

    public static void main(String[] args) {
        CCourant c1 = new CCourant(1,"Salah", 1000, 300);
        CEpargne c2 = new CEpargne(2,"Salem", 5000, 0.08);
        c1.débiter(300);
        c2.débiter(600);
        c1.créditer(500);
        c2.virer(1000,c1);
        c1.consulter();
        System.out.println();
        c2.consulter();
    }
}

```

Output du programme

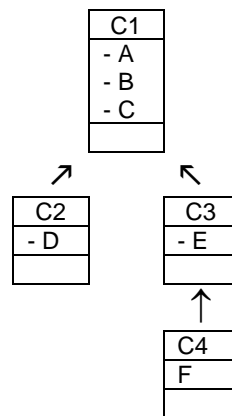
```

Num compte = 1
Propriétaire = Salah
Solde = 2200.0 Dinars
découvert = 300.0 Dinars

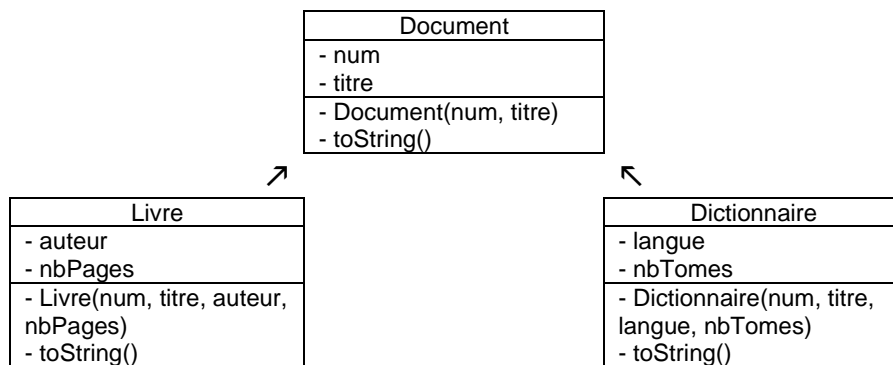
Num compte = 2
Propriétaire = Salem
Solde = 3400.0 Dinars
taux d'interet : 0.08
Monatant des interets : 272.0 Dinars

```

Exercice 1



Exercice 2



```

public class Document {
    protected int num;
    protected String titre;

    public Document (int num, String titre){
        this.num = num;
        this.titre = titre;
    }

    public String toString(){
        return "Num : "+num+"\tTitre : "+titre;
    }
}

```

```

public class Livre extends Document {
    private String auteur;
    private int nbPages;

    public Livre(int num, String titre, String auteur, int nbPages){
        super(num, titre);
        this.auteur = auteur;
        this.nbPages = nbPages;
    }

    public String toString(){
        return super.toString()+"\tAuteur : "+auteur+"\tNb. pages : "+nbPages;
    }
}

```

```

public class Dictionnaire extends Document{
    private String langue;
    private int nbTomes;

    public Dictionnaire(int num, String titre, String langue, int nbTomes){
        super(num, titre);
        this.langue = langue;
        this.nbTomes = nbTomes;
    }

    public String toString(){
        return super.toString()+"\tLangue : "+langue+"\tNbre de tomes : "+nbTomes;
    }
}

```

```

public class Bibliothèque {
    public static void main(String[] args) {
        Document Doc = new Document(1,"Configurer son compte utilisateur");
        Livre Liv = new Livre(2,"La communication sous Unix","J-M Rifflet",799);
        Dictionnaire Dic = new Dictionnaire(3,"Man Pages", "Anglais", 4);
        Document tdoc[] = new Document[3];
        tdoc[0] = Doc;
        tdoc[1] = Liv;
        tdoc[2] = Dic;
        for (int i = 0; i < tdoc.length; i++)
            System.out.println(tdoc[i].toString());
    }
}

```

Output du programme

Num : 1	Titre : Configurer son compte utilisateur	
Num : 2	Titre : La communication sous Unix	Auteur : J-M Rifflet Nb. pages : 799
Num : 3	Titre : Man Pages	Langue : Anglais Nbre de tomes : 4

Exercice 3

```

public class Point {
    protected double abs, ord;

    public Point() {
        this.abs = 0;
        this.ord = 0;
    }

    public Point(double abs, double ord) {

```

```

        this.abs = abs;
        this.ord = ord;
    }

    public double getAbs() {
        return abs;
    }
    public void setAbs(double abs) {
        this.abs = abs;
    }
    public double getOrd() {
        return ord;
    }
    public void setOrd(double ord) {
        this.ord = ord;
    }
    public String toString() {
        return "[" + abs + ", " + ord + "]";
    }
}

```

```

public class Cercle extends Point {
    protected double rayon;

    public Cercle() {
        super();
        this.rayon = 0;
    }

    public Cercle(double abs, double ord, double rayon) {
        super(abs, ord);
        this.rayon = rayon;
    }

    public double getRayon() {
        return rayon;
    }

    public void setRayon(double rayon) {
        this.rayon = rayon;
    }

    public double aire() {
        return Math.PI * rayon*rayon;
    }

    public String toString(){
        return super.toString() + ", rayon = "+rayon+" cm"+",
        Aire = "+aire()+" cm2";
    }
}

```

```

public class Cylindre extends Cercle{

    protected double hauteur;

    public Cylindre() {
        super();
        this.hauteur = 0;
    }
}

```

```

    public Cylindre(double abs, double ord, double rayon,
double      hauteur) {
        super(abs, ord, rayon);
        this.hauteur = hauteur;
    }
    public double getHauteur() {
        return hauteur;
    }

    public void setHauteur(double hauteur) {
        this.hauteur = hauteur;
    }

    public double aire() {
        return 2*super.aire()+2 * Math.PI * rayon*hauteur;
    }

    public double volume() {
        return super.aire()*hauteur;
    }

    public String toString(){
        return super.toString() + ", hauteur = "+hauteur+"
        cm"+"", volume = "+volume()+" cm3";
    }
}

```

```

public class Test {
    public static void main(String[] args){
        Point P = new Point(3,6);
        Cercle C = new Cercle(3,6,10);
        Cylindre Cyl = new Cylindre(3,6,10,15);
        Point tabFormes[] = new Point[3];
        tabFormes[0]=P;
        tabFormes[1]=C;
        tabFormes[2]=Cyl;
        for (Point f:tabFormes)
            System.out.println(f.toString());
    }
}

```

Output du programme

```

[3.0,6.0]
[3.0,6.0], rayon = 10.0 cm, Aire = 314.1592653589793 cm2
[3.0,6.0], rayon = 10.0 cm, Aire = 1570.7963267948967 cm2, hauteur = 15.0 cm, volume
= 4712.3889 cm3

```

Exercice 4

```

public class Facture {
    private int numFac;
    private double montant;
    private boolean payée = false;    //initialisation

    Facture (int num, double mont) {
        numFac = num;
        montant = mont;
    }

    public void payer(){

```

```

        payée = true;
    }
    public boolean getPayée() {
        return payée;
    }

    public double getMontant() {
        return montant;
    }

    public String toString() {
        return "Num: "+numFac+"\tMontant: "+montant+"\tPayée :"+payée;
    }
}

```

```

public class Client {
    protected int numCli;
    protected String nom;
    protected int nbreFactures;
    Facture factures[];

    Client(int num, String nom) {
        numCli = num;
        this.nom = nom;
        nbreFactures = 0;
        factures = new Facture[20];
    }

    public void ajouterFacture(Facture fac) {
        factures[nbreFactures] = fac;
        nbreFactures++;
    }

    public double Crédit() {
        double total = 0;
        for (int i = 0; i < nbreFactures; i++)
            if (factures[i].getPayée() == false)
                total = total + factures[i].getMontant();
        return total;
    }

    public String toString() {
        return "NumCli: "+numCli+"\tNom: "+nom+"\tNbre de Factures : "+nbreFactures+"\tCrédit :"+Crédit();
    }
}

```

```

public class ClientFidèle extends Client {
    private double TauxRemise;

    ClientFidèle(int num, String nom, double tx) {
        super(num, nom);
        TauxRemise = tx;
    }

    public double Crédit() {
        double total = 0;
        for (int i = 0; i < nbreFactures; i++)
            if (factures[i].getPayée() == false)

```

```

        total = total + factures[i].getMontant();
        return total*(1-TauxRemise);
    }

    public String toString(){
        return "NumCli: "+numCli+"\tNom: "+nom+"\tNbre de Factures : "+
            nbreFactures+"\tTaux de remise : "+TauxRemise+"\tCrédit
        : "+Crédit();
    }
}

```

```

public class TestFactures {

    public static void main(String[] args) {
        Facture F1 = new Facture(1, 150);
        F1.payer();
        Facture F2 = new Facture(2, 200);
        Facture F3 = new Facture(3, 150);
        Facture F4 = new Facture(4, 200);
        Facture F5 = new Facture(5, 300);
        Client TabClient[] = new Client[2];
        TabClient[0] = new Client(101, "Ahmed");
        TabClient[0].ajouterFacture(F1);
        TabClient[0].ajouterFacture(F2);
        TabClient[1] = new ClientFidèle(102, "Ali", 0.1);
        TabClient[1].ajouterFacture(F3);
        TabClient[1].ajouterFacture(F4);
        TabClient[1].ajouterFacture(F5);

        for (int i = 0; i<TabClient.length;i++)
            System.out.println(TabClient[i].toString());
    }
}

```

Output du programme

```

NumCli: 101 Nom: Ahmed  Nbre de Factures :2      Crédit :200.0
NumCli: 102 Nom: Ali    Nbre de Factures :3      Taux de remise :0.1
Crédit :585.0

```

Solution des exercices (interaces et classes abstraites)

Exercise 1

```
public interface Tarif_de_base {
    final double tarif_A5 = 0.250;
    final double tarif_A4 = 0.350;
}

public abstract class Courrier implements Tarif_de_base {
    protected double poids;
    protected char mode;    // N ou E
    protected String destination;

    public Courrier(double poids, char mode){
        this.poids = poids;
        this.mode = mode;
    }

    public Courrier(double poids, char mode, String destination){
        this.poids = poids;
        this.mode = mode;
        this.destination = destination;
    }

    public abstract double affranchir();

    public abstract boolean valide();

    public String toString(){
        return ("Valide = "+valide()+"\tPoids = "+poids +"\tMode = "
            +mode + "\tDestination : "+destination);
    }
}
```

```
public class Lettre extends Courrier {
    private String format;

    public Lettre(double poids, char mode, String format){
        super(poids,mode);
        this.format = format;
    }

    public Lettre(double poids, char mode, String destination, String format){
        super(poids,mode, destination);
        this.format = format;
    }

    public boolean valide(){
        return (destination != null);
    }

    public double affranchir(){
        double montant = 0;
        if (valide()){
            montant = 0.250 + 0.02 * poids;
            if (mode == 'E')
                montant *= 2;
        }
        return montant;
    }

    public String toString(){
        return super.toString() + "\tFormat = "+format+"\tMontant = "
            +affranchir();
    }
}
```



```
}

```

```
public class Colis extends Courrier {
    private double volume;

    public Colis(double poids, char mode, double volume){
        super(poids,mode);
        this.volume = volume;
    }

    public Colis(double poids, char mode, String destination, double volume){
        super(poids,mode,destination);
        this.volume = volume;
    }

    public boolean valide(){
        return ((destination != null)&(volume <= 50));
    }

    public double affranchir(){
        double montant = 0;
        if (valide()){
            montant = 0.02 * volume + 0.01 * poids;
            if (mode == 'E')
                montant *= 2;
        }
        return montant;
    }

    public String toString(){
        return super.toString() + "\tVolume = "+volume+"\tMontant = "
            +affranchir();
    }
}
```

```
public class BAL {
    private String code;
    private Courrier contenu[];
    int nbCourriers;

    public BAL(String code){
        this.code = code;
        contenu = new Courrier[30];
        nbCourriers = 0;
    }

    public void ajouter(Courrier c){
        if (nbCourriers <= 29){
            contenu[nbCourriers] = c;
            nbCourriers++;
        }
    }

    public int invalides(){
        int nb = 0;
        for (int i = 0; i< nbCourriers; i++){
            if (!contenu[i].valide())
                nb++;
        }
        return nb;
    }

    public double affranchir(){
        double total = 0;
        for (int i = 0; i< nbCourriers; i++){

```

```

        total = total + contenu[i].affranchir();
    }
    return total;
}

public void afficher(){
    System.out.println("Code :"+code +"\\tNb courriers :"+nbCourriers);
    for (int i = 0; i< nbCourriers; i++){
        System.out.println(contenu[i].toString());
    }
    System.out.println("Nbre de courriers invalides :"+invalides());
    System.out.println("Montant total d'affranchissement :"+
        affranchir());
}
}

```

```

public class Poste {
    public static void main(String[] args) {
        Lettre lettre1 = new Lettre(50, 'N', "Monastir", "A4");
        Lettre lettre2 = new Lettre(100, 'N', "A3");
        Colis colis1 = new Colis(2000, 'N', "Sfax", 15);
        Colis colis2 = new Colis(2500, 'N', "Nabeul", 70);
        BAL ZK01 = new BAL("ZK01");
        ZK01.ajouter(lettre1);          ZK01.ajouter(lettre2);
        ZK01.ajouter(colis1);          ZK01.ajouter(colis2);
        ZK01.afficher();
    }
}

```

Output du programme

```

Code :ZK01  Nb courriers : 4

Valide = true      Poids = 50.0      Mode = N      Destination : Monastir
Format = A4 Montant = 1.25

Valide = false     Poids = 100.0     Mode = N      Destination : null
Format = A3 Montant = 0.0

Valide = true      Poids = 2000.0    Mode = N      Destination : Sfax
Volume = 15.0      Montant = 23

Valide = false     Poids = 2500.0    Mode = N      Destination : Nabeul
Volume = 70.0      Montant= 0.0

Nbre de courriers invalides :2
Montant total d'affranchissement :24.25

```

Exercice 2

```

public abstract class Employé {
    protected String nom;
    protected int age;
    protected int dateRecrut;

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

    public abstract double calculerSalaire();

    public String getNom()

```

```

    {
        return "L'employé " + nom;
    }
}

```

```

public class Vendeur extends Employé{
    private double chiffAff;

    public Vendeur(String nom, int age, int dateRecrut, double
chiffAff) {
        super(nom, age, dateRecrut);
        this.chiffAff = chiffAff;
    }

    public double calculerSalaire(){
        return chiffAff*0.2+100;
    }

    public String getNom()
    {
        return "Le vendeur " + nom;
    }
}

```

```

public class Technicien extends Employé{
    private int nbUnités;

    public Technicien(String nom, int age, int dateRecrut, int nbUnités)
{
        super(nom, age, dateRecrut);
        this.nbUnités = nbUnités;
    }

    public double calculerSalaire(){
        return nbUnités*1.5;
    }

    public String getNom()
    {
        return "Le technicien " + nom;
    }
}

```

```

public class Manutentionnaire extends Employé{
    private int nbHeurs;

    public Manutentionnaire(String nom, int age, int dateRecrut, int nbHeurs) {
        super(nom, age, dateRecrut);
        this.nbHeurs = nbHeurs;
    }

    public double calculerSalaire(){
        return nbHeurs*5;
    }

    public String getNom()
    {
        return "Le manutentionnaire " + nom;
    }
}

```

```
public interface Risque {
    final double prime = 50;
}
```

```
public class TechARisque extends Technicien implements Risque{

    public TechARisque(String nom, int age, int dateRecrut,int nbUnités) {
        super(nom, age, dateRecrut, nbUnités);
    }

    public double calculerSalaire(){
        return super.calculerSalaire()+prime;
    }
}
```

```
public class ManutARisque extends Manutentionnaire implements Risque{
    public ManutARisque(String nom, int age, int dateRecrut,int nbHeures) {
        super(nom, age, dateRecrut, nbHeures);
    }

    public double calculerSalaire(){
        return super.calculerSalaire()+prime;
    }
}
```

```
public class Personnel {
    Employé tab[];
    int nbreEmployés;

    public Personnel()
    {
        tab = new Employé[50];
        nbreEmployés = 0;
    }

    void ajouterEmployé(Employé e){
        tab[nbreEmployés] = e;
        nbreEmployés++;
    }

    void afficherSalaires(){
        for(int i = 0; i<nbreEmployés;i++){
            System.out.print(tab[i].getNom());
            System.out.println(" gagne "+tab[i].calculerSalaire()+ "Dinars");
        }
    }

    double salaireMoyen(){
        double total = 0;
        for(int i = 0; i<nbreEmployés;i++){
            total += tab[i].calculerSalaire();
        }

        return total/nbreEmployés;
    }
}
```

```
public class Salaires {

    public static void main(String[] args) {
        Personnel p = new Personnel();
        p.ajouterEmployé(new Vendeur("Ahmed", 45, 1995, 3000));
        p.ajouterEmployé(new Représentant("Ali", 25, 2001, 2000));
        p.ajouterEmployé(new Technicien("Mohamed", 28, 1998, 1000));
    }
}
```

```

        p.ajouterEmployé(new Manutentionnaire("Samir", 32, 1998, 45));
        p.ajouterEmployé(new TechARisque("Adel", 28, 2000, 1000));
        p.ajouterEmployé(new ManutARisque("Ridha", 30, 2001, 45));
        p.afficherSalaires();
        System.out.println("Le salaire moyen dans l'entreprise est de "
            + p.salaireMoyen() + " Dinars");
    }
}

```

Exercice 3

```

public abstract class Volaille {
    protected int num;
    protected double poids;

    public Volaille(int num, double poids) {
        this.num = num;
        this.poids = poids;
    }

    public double getPoids() {
        return poids;
    }

    public void setPoids(double poids) {
        this.poids = poids;
    }

    public abstract double prix();

    public abstract boolean assezGrosse();

    public String toString() {
        return "num=" + num + ", poids actuel =" + poids;
    }
}

```

```

public class Poulet extends Volaille{
    private static double prixKg;
    private static double poidsAbattage;

    public Poulet(int num, double poids){
        super(num, poids);
    }

    public static void setPrixKg(double prixKg) {
        Poulet.prixKg = prixKg;
    }

    public static void setPoidsAbattage(double poidsAbattage) {
        Poulet.poidsAbattage = poidsAbattage;
    }

    public double prix(){
        return poids * prixKg;
    }

    public boolean assezGrosse(){
        return (poids >= poidsAbattage);
    }

    public String toString(){
        String s = "["+super.toString();
        if (!this.assezGrosse())

```

```

        s = s + "\tAssez Petite]";
    else
        s = s + "\tAssez Grosse"+"\tPrix :"+prix()+"]";
    return s;
}

```

```

public class Canard extends Volaille {
    private static double prixKg;
    private static double poidsAbattage;

    public Canard(int num, double poids){
        super(num, poids);
    }

    public static void setPrixKg(double prixKg) {
        Canard.prixKg = prixKg;
    }

    public static void setPoidsAbattage(double poidsAbattage) {
        Canard.poidsAbattage = poidsAbattage;
    }

    public double prix(){
        return poids * prixKg;
    }

    public boolean assezGrosse(){
        return (poids >= poidsAbattage);
    }

    public String toString(){
        String s = "["+super.toString();
        if (!this.assezGrosse())
            s = s + "\tAssez Petite]";
        else
            s = s + "\tAssez Grosse"+"\tPrix :"+prix()+"]";
        return s;
    }
}

```

```

public class Ferme {
    private int nbBetes;
    private Volaille[] volailles;

    public Ferme(){
        nbBetes = 0;
        volailles = new Volaille[50];
    }

    public void ajouter(Volaille v){
        volailles[nbBetes] = v;
        nbBetes++;
    }

    public void rechercher(int num){
        boolean trouve = false;
        int i = 0;
        while ((!trouve) && (i < nbBetes)){
            if (volailles[i].num == num) {
                System.out.println(volailles[i].toString());
            }
        }
    }
}

```

```

        trouve = true;
    }
    i++;
}

if (!trouve) {
    System.out.println("volaille inexistante");
}

}

public double gains(){
    double tot = 0;
    for (int i = 0; i < nbBetes; i++){
        if (volailles[i].assezGrosse()) {
            tot+=volailles[i].prix();
        }
    }
    return tot;
}
}

```

```

public class TestFerre {

    public static void main(String[] args) {
        Poulet P1 = new Poulet(1,1.0);
        Poulet P2 = new Poulet(2,2.5);
        Canard C1 = new Canard(3,1.0);
        Canard C2 = new Canard(4,3.5);
        Canard C3 = new Canard(5, 3.0);
        Poulet.setPrixKg(2);
        Canard.setPrixKg(3);
        Poulet.setPoidsAbattage(1.2);
        Canard.setPoidsAbattage(1.5);
        Ferme F = new Ferme();
        F.ajouter(P1); F.ajouter(P2); F.ajouter(C1); F.ajouter(C2);
        F.ajouter(C3);
        F.rechercher(2);
        F.rechercher(4);
        F.rechercher(5);
        System.out.println("Gains espéré = " + F.gains() + "
Dinars");
    }
}

```

Output du programme

```

[num=2, poids actuel =2.5    Assez Grosse    Prix :5.0]
[num=4, poids actuel =3.5    Assez Grosse    Prix :10.5]
[num=5, poids actuel =3.0    Assez Grosse    Prix :9.0]
Gains espéré = 24.5 Dinars

```

Exercice 4

```

public class Disque implements Media{
    private String titre;
    private String auteur;
    private double prix;
    private Date dateParution;
    private int nbPistes;
}

```

```

public Disque(String titre, String auteur, double prix, Date
dateParution,int nbPistes) {
    this.titre = titre;
    this.auteur = auteur;
    this.prix = prix;
    this.dateParution = dateParution;
    this.nbPistes = nbPistes;
}

public int getNbPistes() {
    return nbPistes;
}

public void setNbPistes(int nbPistes) {
    this.nbPistes = nbPistes;
}

public String toString() {
    return "Disque [titre=" + titre + ", auteur=" + auteur + ",
prix="+ prix + ", dateParution=" + dateParution + ",
nbPistes="
    + nbPistes + "];"
}

public String getTitre() {
    return titre;
}

public String getAuteur() {
    return auteur;
}

public double getPrix() {
    return prix;
}

public Date getDate() {
    return dateParution;
}
}

```

```

public class Livre implements Media{
    private String titre;
    private String auteur;
    private double prix;
    private Date dateParution;
    private int nbPages;

    public Livre(String titre, String auteur, double prix, Date
dateParution,int nbPages) {
        this.titre = titre;
        this.auteur = auteur;
        this.prix = prix;
        this.dateParution = dateParution;
        this.nbPages = nbPages;
    }

    public int getNbPages() {
        return nbPages;
    }

    public void setNbPages(int nbPages) {
        this.nbPages = nbPages;
    }
}

```



```

    }

    public String toString() {
        return "Livre [titre=" + titre + ", auteur=" + auteur + ",
nbPages="
        + nbPages + "]";
    }

    public String getTitre() {
        return titre;
    }

    public String getAuteur() {
        return auteur;
    }

    public double getPrix() {
        return prix;
    }

    public Date getDate() {
        return dateParution;
    }
}

```

```

public abstract class Media2 {
    protected String titre;
    protected String auteur;
    protected double prix;
    protected Date dateParution;

    public Media2(String titre, String auteur, double prix, Date
dateParution) {
        this.titre = titre;
        this.auteur = auteur;
        this.prix = prix;
        this.dateParution = dateParution;
    }

    public String getTitre() {
        return titre;
    }

    public String getAuteur() {
        return auteur;
    }

    public double getPrix() {
        return prix;
    }

    public Date getDate() {
        return dateParution;
    }

    public abstract String toString();
}

```

```

public class Livre extends Media2{
    private int nbPages;

```

```

public Livre(String titre, String auteur, double prix, Date
dateParution,int nbPages) {
    super(titre, auteur, prix, dateParution);
    this.nbPages = nbPages;
}

public int getNbPages() {
    return nbPages;
}

public void setNbPages(int nbPages) {
    this.nbPages = nbPages;
}

public String toString() {
    return "Livre [titre=" + titre + ", auteur=" + auteur + ",
prix=" + prix + ", dateParution=" + dateParution + ", nbPages="
+ nbPages + "]";
}
}

```

```

public class Mediatheque {
    private String nom;
    private int capaciteMax;
    private int nbreMedias;
    private Media medias[];

    public Mediatheque(String nom, int capaciteMax) {
        this.nom = nom;
        this.capaciteMax = capaciteMax;
        this.nbreMedias = 0;
        this.medias = new Media[capaciteMax];
    }

    public void ajouter(Media m){
        medias[nbreMedias]=m;
        nbreMedias++;
    }

    public boolean rechercher(String titre){
        for(int i=0; i < nbreMedias; i++){
            if(medias[i].getTitre() == titre){
                return true;
            }
        }
        return false;
    }

    public String toString() {
        String st = "Mediatheque [nom=" + nom + ", nbreMedias=" +
nbreMedias;
        for(int i=0; i < nbreMedias; i++){
            st = st + "\n" + medias[i].toString();
        }
        return st + "]";
    }

    public static void main(String[] args) {
        Mediatheque mediaPlus = new Mediatheque("mediaPlus",100);
    }
}

```

```
mediaPlus.ajouter(new Livre("titt", "Auut", 99.99, new
Date(17,12,2014),300));
mediaPlus.ajouter(new Disque("titt", "Auut", 99.99, new
Date(17,12,2014),10));
System.out.println(mediaPlus.rechercher("titt"));
System.out.println(mediaPlus.toString());
}
}
```

Bibliographie

Ouvrages

1. Claude Delannoy, « Programmer en java », Eyrolles, 2012
2. Chiraz Ben Othmane Zribi, « Initiation à la programmation orientée objet en Java », Centre de publication Universitaire, 2003
3. E. Reed doke, John W. Satzingzr, Susan R. Williams, « Object-Oriented Application Development using Java», Course Technology, 2002
4. Bruce Eckel, «Penser en Java», Prentice Hall, 2002
5. Mala Gupta, OCA Java SE 7, Certification Guide, 2013

Sites web

<http://www.jmdoudoux.fr/java/dej/chap-techniques-base.htm>

<http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classeSystem.html>

<http://www.florat.net/tutorial-java>

<http://www.tutorialspoint.com/java>

<http://julien.sopena.fr/enseignements/L3-PRO-JAVA/cours/>

Avant propos

Ce cours intitulé « Programmation Orientée Objet I » est destiné aux étudiants de la deuxième année Technologies de l'informatique (premier semestre) option DSI, MDW, RSI et SEM pour un volume de 3 heures de cours intégré par semaine ; soit un volume total de 42 heures. Un atelier de 3 heures hebdomadaires est dispensé parallèlement à ce cours.

L'objectif de ce cours est de permettre aux étudiant d'acquérir les connaissances nécessaires pour écrire correctement des programmes orientés objet.

Le présent rapport comporte sept leçons. Les deux premières leçons exposent la syntaxe du langage Java avec plusieurs exemples d'application. Une attention particulière a été attribuée aux structures de tableaux.

A partir de la troisième leçon, nous commençons la présentation des concepts de base de l'approche objet (classe, objet, encapsulation, constructeur, accesseur, ...).

La quatrième leçon montre comment instancier des objets à partir d'une classe prédéfinie puis manipuler ces objets à travers les méthodes publiques offertes par cette classe : il s'agit de la classe **String** qui permet de définir et manipuler des chaînes de caractères sous forme d'objets. Deux autres classes qui permettent de définir des chaînes un peu particulières sont également introduites : il s'agit des classes `StringBuffer` et `StringTokenizer`.

La cinquième leçon montre comment tirer profit du concept d'héritage afin de construire une classe fille à partir d'une classe mère déjà définie.

La sixième leçon expose deux structures qui ressemblent aux classes mais qui ne sont pas instanciables : il s'agit des interfaces et des classes abstraites.

La dernière leçon explique comment détecter et traiter les erreurs (anomalies) qui peuvent apparaître lors de l'exécution d'un programme grâce au mécanisme de gestion des exceptions.

Le support comporte plusieurs exercices permettant de consolider les connaissances acquises par les étudiants et d'évaluer le degré d'assimilation des concepts présentés en cours. Les solutions proposées aux exercices ne sont pas uniques et dans plusieurs cas elles ne sont pas optimales car nous avons toujours privilégié l'apport pédagogique et la simplicité.

Exemple

<pre>class Cercle { private double rayon ; public static void main(String[] args) { Cercle C = new Cercle(); // correct } }</pre>	<pre>class Cercle { private double rayon ; public Cercle(double r) { //constructeur rayon = r; } public static void main(String[] args) { Cercle C = new Cercle(); // Erreur } }</pre>
--	--

Exercice 4 : Définir une classe **Point** qui possède le schéma suivant :

Classe Point
<ul style="list-style-type: none">- int x // abscisse- int y // ordonnée
<ul style="list-style-type: none">- Point () // constructeur par défaut qui initialise les coordonnées à (0,0)- Point(int abs, int ord) // 2^{ème} constructeur qui initialise les coordonnées à (abs,ord)- Deplacer(int dx, int dy) // déplace un point d'une distance horizontale dx et d'une distance verticale dy (les didtances dx et dy peuvent être positives ou négatives)- toString() // retourne une chaine de caractères contennat les coordonnées actuelles du point sous la forme (x,y)- public static void main(String[] args) : méthode principale dans laquelle on crée un point A de coordonnées (2,3), on le déplace de 2 cm sur l'horizontale et 1 cm sur la verticale puis on affiche les nouvelles coordonnées de ce point.

Introduction

Nous aborderons dans ce chapitre les objets de la classe String, StringBuffer et StringTokenizer.

II. Les objets de la classe StringBuffer

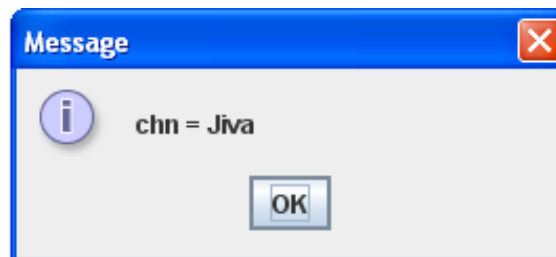
II.1. Particularités

Un objet de type StringBuffer est une chaîne de caractères modifiable qui présente certains avantages de manipulation et de performance. La classe StringBuffer possède en effet des méthodes permettant d'effectuer des modifications sur l'objet lui-même. Par exemple, ce qui était impossible pour le String (modifier un caractère de la chaîne) est possible pour le StringBuffer grâce à sa méthode setCharAt().

Exemple

```
import javax.swing.JOptionPane;
public class Chaines {
    public static void main(String[] args) {
        StringBuffer chn = new StringBuffer("Java");
        chn.setCharAt(1, 'i');
        JOptionPane.showMessageDialog(null, "chn = " + chn);
    }
}
```

Output



II.2. Principales méthodes de la classe StringBuffer

Méthode	Description
char charAt(int index)	Renvoie le caractère à la position index. L'index du premier caractère d'une chaîne est 0.
void setCharAt(int index, char car)	Mettre le caractère à l'index spécifié de la chaîne actuelle au caractère donné en argument.
StringBuffer deleteCharAt(int index)	Supprime le caractère qui se trouve à la position index
void delete(int Debut, int Fin)	Supprime les caractères de la chaîne de Debut à Fin-1
StringBuffer insert(int Index, String str)	Insère la chaîne str à l'index précisé.
public StringBuffer append(Object obj)	Concatène la représentation textuelle de l'objet obj à la fin de la chaîne actuelle
StringBuffer replace(int Debut, int Fin, String str)	Remplace la partie de la chaîne comprise entre Debut et Fin-1 par la chaîne str.
String substring(int Depart, int Fin)	Ne conserve de la chaîne que les caractères de l'index Debut à l'index Fin-1. Autrement dit, Fin-Debut vaut la longueur de la nouvelle chaîne.
StringBuffer reverse()	Inverse la valeur de l'objet StringBuffer qui a appelé la méthode.
String toString()	Renvoie l'objet de type String correspondant à cette chaîne.

Exemple 1 : Que va afficher le programme suivant :

```
public class StringBuf {
    public static void main (String[] args){
        StringBuffer chn = new StringBuffer("algo");
        chn.replace(0, 3, "Styl");
        System.out.println(chn.reverse());
    }
}
```

Output

olytS

Exemple 2 : Que fait la méthode « invert() » définie ci-dessous ?

```
class TestStringBuffer{
    public static String invert(String source) {
        int len = source.length();
        StringBuffer dest = new StringBuffer(len);
        for (int i = (len-1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

III. La classe StringTokenizer

Cette classe du package `java.util` sert à couper une chaîne en sous-chaînes de façon rapide. Le constructeur le plus courant est:

`StringTokenizer(String input, String délimiteur)`

- N'importe quelle chaîne peut servir de séparateur (des espaces, des slashes, tirets...). L'espace est le séparateur par défaut.
- La méthode **countTokens()** retourne le nombre d'éléments restant à lire dans la chaîne source.
- La méthode **nextToken()** renvoie un `String` contenant la sous-chaîne suivante.
- La méthode **hasMoreTokens()** renvoie « true » s'il reste des sous-chaînes à lire.

Exemple

```
import java.util.StringTokenizer;

public class StringTok {
    public static void main(String[] args) {
        String monTexte="Le langage Java";
        StringTokenizer st=new StringTokenizer(monTexte, " ");
        System.out.println("Nbre de tokens : "+st.countTokens());
        StringBuffer sortie = new StringBuffer();
        while(st.hasMoreTokens()){
            sortie.append(st.nextToken()+"\n");
        }
        System.out.println(sortie);
    }
}
```

Output

```
Nbre de tokens : 3
Le
langage
Java
```

16. Etant donné la signature de la méthode `replace()`

```
public String replace(CharSequence target, CharSequence  
replacement) {  
... }
```

CharSequence est une interface implémentée par quelles classes concrètes ?

- a- **String**
- b- StringBoxer
- c- **StringBuffer**
- d- **StringBuilder**

17. Que va afficher le fragment de code suivant ?

```
StringTokenizer s3 = new StringTokenizer("world wide web");  
System.out.println(s3.nextToken().nextToken());
```

- a- world
- b- wide
- c- web
- d- **erreur de compilation**

18. Que va afficher le fragment de code suivant ?

```
StringBuffer s3 = new StringBuffer("word wide web");  
System.out.println(s3.countTokens());
```

- a- 0
- b- 3
- c- **erreur de compilation**

19. Laquelle des méthodes suivantes n'est pas définie pour la classe StringTokenizer ?

- a- countTokens()
- b- hasMoreTokens()
- c- nextToken()
- d- toString()

20. Laquelle des méthodes suivantes n'est pas définie pour la classe StringBuffer ?

- a- setCharAt(index, ch)
 - b- append(str)
 - c- insert(offset, str)
 - d- lastIndexOf(str)
-

Exercice 2 : Les StringTokenizer

Utilisez la classe StringTokenizer à partir d'une chaîne valant "#1=toto#2=titi#3=tata" avec les séparateurs appropriés pour reconstituer et afficher les valeurs de toto, titi et tata sous la forme:

1:toto
2:titi
3:tata

en utilisant les méthodes de la classe StringTokenizer.

Exercice 2

```
import java.util.StringTokenizer;

public class Recherche {
    public static void main(String[] args) {
        StringTokenizer st=new StringTokenizer("#1=toto#2=titi#3=tata","#");
        while(st.hasMoreTokens())
            System.out.println(st.nextToken().replace('=',':'));
    }
}
```

2. La surcharge qui correspond à la possibilité de définir des comportements différents pour la même méthode selon les arguments passés en paramètres peut être considérée comme une autre forme de polymorphisme.

Exercice 10 : On désire écrire une application éducative pour apprendre la notion d'ensemble à des enfants et les initier aux notions d'appartenance, d'intersection, d'union et d'inclusion. Dans ce cadre, on souhaite faire manipuler aux enfants des ensembles de lettres (lettres de 'a' à 'z') sans répétition (une lettre ne peut être présente qu'une seule fois).

Un ensemble de lettres est représenté par un tableau `e` de 26 booléens, `e[i]` ayant la valeur *true* si la *i*^{ème} lettre de l'alphabet appartient à l'ensemble et *false* sinon.

Par exemple, l'ensemble `e = {'a','c','f','j','w','z'}` sera représenté par le tableau ci-dessous :

	a	b	c	d	e	f	g	H	i	J	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
e	V	F	V	F	F	V	F	F	F	V	F	F	F	F	F	F	F	F	F	F	F	F	V	F	F	V
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

1. Comment vérifier si une lettre `c` quelconque existe dans le tableau ou non (on vous rappelle que l'unicode de 'a' est 97).
2. Etant donnée cette représentation pour les ensembles de lettres, définir une classe `EnsLettres`, sachant que les opérations définies sur de tels ensembles sont les suivantes :
 - `EnsLettres()` qui crée un ensemble initialement vide (toutes les cases sont à *false*)
 - `getEns()` qui retourne le tableau `e`
 - `ajouter(lettre)` qui ajoute une lettre à l'ensemble actuel (en mettant la case correspondante dans le tableau à *true*)
 - `enlever(lettre)` qui enlève une lettre de l'ensemble actuel (en mettant la case correspondante dans le tableau à *false*)
 - `afficher()` qui affiche l'ensemble sur la console sous la forme suivante :

{lettre1, lettre2, ...}

Si l'ensemble est vide, seules les accolades seront affichées.

- `cardinal()` qui renvoie le nombre d'éléments de l'ensemble.
 - `estVide()` qui teste si l'ensemble est vide et renvoie la valeur *true* dans ce cas et *false* sinon.
 - `appartient(lettre)` qui teste si une lettre quelconque est présente ou non dans l'ensemble.
3. Définir une classe `TestEnsLettres` qui contient les méthodes statiques suivantes :
 - `intersection(e1,e2)` qui retourne l'intersection de deux ensembles de lettres
 - `union(e1,e2)` qui retourne l'union de deux ensembles de lettres
 - `différence(e1,e2)` qui retourne la différence entre l'ensemble `e1` et l'ensemble `e2`
 - `Egaux(e1,e2)` qui teste si deux ensembles de lettres sont égaux
 - `estInclus(e1,e2)` qui teste si l'ensemble `e1` est inclus dans `e2`.
 - une méthode `main()` dans laquelle vous créez deux ensembles `e1` et `e2` et vous testez les différentes méthodes que vous avez écrites.
-

