Université Mohammed Premier - Faculté des Sciences
Département d'Informatique
Oujda

Programmation Orientée Objet
EN JAVA

Pr. Abdelhak **LAKHOUAJA**

SMI S5 2016/2017

fso.umpoujda.com

بسم الله الرحمن الرحيم

Programmation Orientée Objets avec Java

Pr. Abdelhak LAKHOUAJA

Département de Mathématiques et Informatique
Faculté des Sciences
Oujda

a.lakhouaja@ump.ma

http://lakhouaja.oujda-nlp-team.net/

SMI-S5
Année universitaire : 2016/2017

Chapitre 1
Introduction

Introduction

Les langages orientés objets prennent en charge les quatre caractéristiques importantes suivantes :

1. Encapsulation
2. Abstraction
3. Héritage
4. Polymorphisme

La plateforme Java

Les composantes de la plateforme Java sont :

- Le langage de programmation.
- La machine virtuelle (The Java Virtual Machine - JVM).
- La librairie standard.

Langage Java

Java est compilé et interprété :

- Le code source Java (se terminant par `.java`) est compilé en un fichier Java bytecode (se terminant par `.class`)
- Le fichier Java bytecode est interprété (exécuté) par la JVM
- La compilation et l'exécution peuvent se faire sur différentes machines
- Le fichier bytecode est portable. Le même fichier peut être exécuté sur différentes machines (hétérogènes).

Premier programme en Java

Un exemple qui permet d'afficher le message `Bonjour - SMI-S5` :

```
public class Bonjour {
    public static void main(String[] args) {
        System.out.println("Bonjour - SMI-S5");
    }
}
```

- Le programme doit être enregistré (**obligatoirement**) dans un fichier portant le même nom que celui de la classe **Bonjour.java**.
- Pour compiler le programme précédent, il faut tout d'abord installer l'environnement de développement **JDK** (**J**ava **D**eveloppment **K**it).

Installation sous Linux

Installer openjdk-7, en tapant la commande :

```
sudo apt-get install openjdk-7-jdk
```

Ou bien, télécharger la version de jdk (jdk-7uxy-linux-i586.tar.gz ou jdk-7uxy-linux-x64.tar.gz) correspondant à votre architecture (32 ou 64 bits) de :

```
http://www.oracle.com/technetwork/java/javase/
downloads/jdk7-downloads-1880260.html
```

Décompresser le fichier téléchargé en tapant la commande :

- Systèmes 32 bits : `tar xfz jdk-7uxy-linux-i586.tar.gz`
- Systèmes 64 bits : `tar xfz jdk-7uxy-linux-x64.tar.gz`

Remplacer xy par le numéro de mise-à-jour, par exemple 55.

Installation sous Linux

Ajouter dans le fichier .bashrc (gedit ~/.bashrc) la ligne suivante :

```
PATH=~/jdk1.7.0_xy/bin:$PATH
```

Faites attention aux majuscules ! Linux (comme Java et C) est sensible à la casse (A≠a).

Installation sous Windows

Télécharger la version de jdk (jdk-7uxy-windows-i586.exe ou jdk-7uxy-windows-x64.exe) correspondant à votre architecture (32 ou 64 bits) de :

`http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`

Exécuter le fichier téléchargé et ajouter `C:\Program Files\Java\jdk1.7.0\_xy\bin` au chemin, en modifiant la variable **path**. La valeur de **path** doit ressembler à ce qui suit :

```
C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Java\jdk1.7.0_xy\bin
```


Installation sous Windows

Pour modifier la valeur de path :

1. cliquer sur **démarrer** puis **Panneau de configuration** puis **Système et sécurité** puis **Paramètres système avancés**
2. cliquer sur **Variables d'environnement** puis chercher dans **variables système**, **Path** et modifier son contenu.

Compilation

Dans une console, déplacez vous dans le répertoire ou se trouve votre fichier et tapez la commande suivante :

```
javac Bonjour.java
```

Après la compilation et si votre programme ne comporte aucune erreur, le fichier `Bonjour.class` sera généré.

Exécution

Il faut exécuté le fichier .class en tapant la commande (sans extension) :

```
java Bonjour
```

Après l'exécution, le message suivant sera affiché.

```
Bonjour - SMI-S5
```

Déclaration

System.out.println("Bonjour - SMI-S5") ;

- **System** : est une classe
- **out** : est un objet dans la classe System
- **println()** : est une méthode (fonction) dans l'objet out. Les méthodes sont toujours suivi de ().
- les points séparent les classes, le objets et les méthodes.
- chaque instruction doit se terminer par " ;"
- Bonjour - SMI-S5 : est une chaîne de caractères.

Première classe

Dans Java, toutes les déclarations et les instructions doivent être faites à l'intérieure d'une classe.

```
public class Bonjour
```

veut dire que vous avez déclaré une classe qui s'appelle **`Bonjour`**.

Le nom d'une classe (**identifiant**) doit respecter les contraintes suivantes :

- l'identifiant doit commencer par une lettre (arabe, latin, ou autre), par _ ou par $. Le nom d'une classe ne peut pas commencer par un chiffre.
- un identifiant ne doit contenir que des lettres, des chiffres, _, et $.
- un identifiant ne peut être un mot réservé.
- un identifiant ne peut être un des mots suivants : `true`, `false` ou `null` . Ce ne sont pas des mots réservés mais des types primitifs et ne peuvent pas être utilisés par conséquent.

Mots réservés

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

Même si **const** et **goto** sont des mots réservés, ils ne sont pas utilisés dans les programmes Java et n'ont aucune fonction.

Recommandations concernant les noms des classes

En java, il est, par convention, souhaitable de commencer les noms des classes par une lettre majuscule et utiliser les majuscules au début des autres noms pour agrandir la lisibilité des programmes.

Quelques noms de classes valides

Nom de la classe	**description**
Etudiant	Commence par une majuscule
PrixProduit	Commence par une majuscule et le le deuxième mot commence par une majuscule
AnnéeScolaire2014	Commence par une majuscule et ne contient pas d'espace

Quelques noms de classes non recommandées

Nom de la classe	description
etudiant	ne commence pas par une majuscule
ETUDIANT	le nom en entier est en majuscule
Prix_Produit	_ n'est pas utilisé pour indiqué un nouveau mot
annéescolaire2014	ne commence par une majuscule ainsi que le deuxième, ce qui le rend difficile à lire

Quelques noms de classes non valides

Nom de la classe	description
Etudiant#	contient #
double	mot réservé
Prix Produit	contient un espace
2014annéescolaire	commence par un chiffre

Méthode main

Pour être exécuté, un programme Java doit contenir la méthode spéciale **main()**, qui est l'équivalent de **main()** du langage C.

- **`String[] args`**, de la méthode **main** permet de récupérer les arguments transmis au programme au moment de son exécution.
- **`String`** est une classe. Les crochets ([]) indiquent que **args** est un tableau (voir plus loin pour plus d'informations sur l'utilisation des tableaux).
- Le mot clés **void**, désigne le type de retour de la méthode **main()**. Il indique que **main()** ne retourne aucune valeur lors de son appel.
- Le mot clés **static** indique que la méthode est accessible et utilisable même si aucun objet de la classe n'existe.
- Le mot clés **public** sert à définir les droits d'accès. Il est obligatoire dans l'instruction **public static void main(String[] args)** et peut être omis dans la ligne **public class Bonjour**.

Commentaires

Les commentaires peuvent s'écrire sur une seule ligne ou sur plusieurs ligne, comme dans l'exemple suivant :

```
/*
Premier programme en Java
contient une seule classe avec une seule methode
*/
public class Bonjour {
    //methode principale
    public static void main(String[] args) {
        System.out.println("Bonjour - SMI-S5");
    }
}
```

Commentaires pour la documentation Java

Les commentaires qui commencent par /** et se terminent par */ servent pour générer une documentation automatique.

Exemple :

```
/**
 * C'est une classe de test
 */
public class TestCommentaires {
    /**
     * Methode main (principale)
     *
     * @param args arguments
     */
    public static void main(String[] args) {
        System.out.println("Bonjour - SMI-S5 ");
    }
}
```


Commentaires pour la documentation Java

Pour générer la documentation sous format HTML, il faut taper la commande :

javadoc TestCommentaires.java

Données primitifs

Java est un langage (presque) purement orienté objets, puisqu'il permet la déclaration de types de données primitifs.

Lors de la déclaration d'une variable sans qu'aucune valeur ne lui soit affecté, la variable sera non initialisée.

Par exemple, lors de déclaration de la variable **age** de type **int** :

```
int age;
```

si vous essayez de l'utiliser dans une expression ou de l'afficher, vous allez recevoir une erreur lors de la compilation (contrairement au langage **C**) indiquant que la variable n'est pas initialisée (The local variable age may not have been initialized).

Types numériques

En java, il existe 4 types entiers et 2 types réels :

Type	**Valeur Minimale**	**Valeur Maximale**	**Taille en octets**
byte	-128	127	1
short	$-32\,768$	32 767	2
int	$-2\,147\,483\,648$	2 147 483 647	4
long	$-9\,223\,372\,036\,854\,775\,808$	$9\,223\,372\,036\,854\,775\,807$	8
float	$-3.4 * 10^{38}$	$3.4 * 10^{38}$	4
double	$-1.7 * 10^{308}$	$1.7 * 10^{308}$	8

Types numériques

Remarques :

1. Par défaut, une valeur comme 3.14 est de type **double**, donc pour l'affecter à une variable de type **float**, il faut la faire suivre par la lettre **f** (majuscule ou minuscule).
   `float pi=3.14F, min=10.1f;`
2. Par défaut les entiers sont de type **int**. Pour affecter une valeur inférieure à $-2\ 147\ 483\ 648$ ou supérieure à $2\ 147\ 483\ 647$ à un **long**, il faut la faire suivre par la lettre **l** (majuscule ou minuscule).
   `long test=4147483647L;`

Caractères

On utilise le type **char** pour déclarer un caractère. Par exemple :

```
char c='a';
char etoile='*';
```

Les caractères sont codés en utilisant **l'unicode**. Ils occupent 2 octets en mémoire. Ils permettent de représenter **65 536** caractères, ce qui permet de représenter (presque) la plupart des symboles utilisés dans le monde.

Séquences d'échappement

Séquence d'échappement	Description
\n	nouvelle ligne (new line)
\r	retour à la ligne
\b	retour d'un espace à gauche
\\	\ (back slash)
\t	tabulation horizontale
\'	apostrophe
\"	guillemet

Type boolean

Il permet de représenter des variables qui contiennent les valeurs vrai (**true**) et faux (**false**).

```
double a=10, b=20;
boolean comp;
comp=a>b; //retourne false
comp=a<=b; //retourne true
```

Constantes

Pour déclarer une constante, il faut utiliser le mot clés **final**. Par convention, les constantes sont écrites en majuscule.

Exemple :

```
final int MAX = 100;
final double PI = 3.14;
...
final int MAX_2 = MAX * MAX;
```


Expressions et opérateurs

Comme pour le langage C, Java possède les opérateurs :

- arithmétiques usuels (+, -, *, /, %)
- de comparaison (<, <=, >, >=, ==, !=).

On retrouve aussi les expressions arithmétiques, les comparaisons et les boucles usuelles du langage C.

D'autres façons propres au langage Java seront vues dans les chapitres suivants.

Exemple 1 :

```
public class Expressions {
    public static void main(String[] args) {
        double a=10, b=20, max, min;

        max = b;
        min = a;
        if (a > b) {
            max = a;
            min = b;
        }
        //Equivalent a printf du langage C
        System.out.printf("max = %f\tmin =%f\n", max
            , min);

    }
}
```


Exemple 2 :

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

        //Carrees des nombres impairs de 1 a 30
        for(int i=1; i<=30; i+=2)
            System.out.printf("%d^2 = %d\n",i,i*i);
    }
}
```

Exercices

Exercice 1

Parmi les identifiants suivants, quels sont ceux qui sont valides ?

a - nomEtudiant
b - prenom Etudiant
c - static
d - BONJOUR
e - 23code
f - code13
g - serie#
h - Test_Comp
i - goto

Exercice 2

Donnez le résultat des expressions suivantes :

a - 15 / 2 (7)
b - 15 % 2 (1)
c - 14 % 2 (0)
d - 31 % 7 (3)
e - 5 + 6 * 3 (23)
f - 25 + 3 / 2 (26)

Exercice 3

Donnez le résultat des expressions booléennes suivantes :

a - 4 <= 9 (true)
b - 12 >= 12 (true)
c - 3 + 4 == 8 (false)
d - 7 < 9 – 2 (false)
e - 5 != 5 (false)
f - 15 != 3 * 5 (true)
g - 9 != -9 (true)
h - 3 + 5 * 2 == 16 (false)

Exercice 4

Si j = 5 et k = 6, alors la valeur de j++ == k est :

a - 5
b - 6
c - true
d - false

Solution

false : j++ == k $\Leftrightarrow \begin{cases} j == k \\ j++ \end{cases}$

Exercice 5

Quel est le résultat de la sortie du code suivant ?

```
for(int i = 0; i < 3; ++i)
    for(int j = 0; j < 2; ++j)
        System.out.print(i + " " + j + " ");
```

a - 0 0 0 1 1 0 1 1 2 0 2 1
b - 0 1 0 2 0 3 1 1 1 2 1 3
c - 0 1 0 2 1 1 1 2
d - 0 0 0 1 0 2 1 0 1 1 1 2 2 0 2 1 2 2

Solution

0 0 0 1 1 0 1 1 2 0 2 1

Chapitre 2
Classes et objets

Introduction

Comme mentionné au chapitre précédent, Java est un langage (presque) purement orientée objets. Tout doit être à l'intérieure d'une classe.

Déclaration d'une classe

- Une classe est crée en utilisant le mot clés **class**.
- Elle peut contenir des méthodes (fonctions), des attributs (variables).

Pour illustrer ceci, nous allons créer le prototype de la classe **Etudiant** :

```
class Etudiant {
    //Declarations
    //attributs et methodes
}
```


Remarques :

on peut définir plusieurs classes dans un même fichier à condition qu'une seule classe soit précédée du mot clés **public** et que le fichier porte le même nom que la classe publique ;

une classe peut exister dans un fichier séparé (qui porte le même nom suivi de **.java**) ;

pour que la machine virtuelle puisse accéder à une classe contenant la méthode main, il faut que cette classe soit publique.

Définition des attributs

Nous supposons qu'un étudiant est caractérisé par son nom, son prénom, son cne et sa moyenne.

```
class Etudiant {
    private String nom;
    private String prenom;
    private String cne;
    private double moyenne
}
```

Par convention, les noms des attributs et des méthodes doivent être en minuscule. Le premier mot doit commencer en minuscule et les autres mots doivent commencer en majuscule (**ceciEstUneVariable**, **ceciEstUneMethode**).

Remarque

La présence du mot clés **private** (privé) indique que les variables ne seront pas accessibles de l'extérieure de la classe où elles sont définies. C'est possible de déclarer les variables non privé, mais c'est déconseillé.

Définition des méthodes

Dans la classe étudiant, nous allons définir trois méthodes :

- **initialiser()** : qui permet d'initialiser les informations concernant un étudiant.
- **afficher()** : qui permet d'afficher les informations concernant un étudiant.
- **getMoyenne** : qui permet de retourner la moyenne d'un étudiant.

Exemple

```
class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;
    public void initialiser(String x, String y,
        String z, double m) {
         nom = x;
         ...
    }
    public void afficher() {
         System.out.println("Nom : "+nom);
         ...
    }
    public double getMoyenne() {
         return moyenne;
    }
}
```

Remarques

- On peut définir plusieurs méthodes à l'intérieure d'une classe.
- La présence du mot clés **public** (publique) indique que les méthodes sont accessibles de l'extérieure de la classe. C'est possible de déclarer des méthodes privés.
- Il existe d'autres modes d'accès aux variables et aux méthodes qu'on verra plus loin.

Utilisation des classes

Après déclaration d'une classe, elle peut être utilisée pour déclarer un objet (variable de type classe) à l'intérieure de n'importe quelle méthode. Pour utiliser la classe étudiant :

```
Etudiant et;
```

Contrairement aux types primitifs, la déclaration précédente ne réserve pas de place mémoire pour l'objet de type **Etudiant** mais seulement une référence à un objet de type **Etudiant**. Pour réserver de la mémoire, il faut utiliser le mot clés **new** de la façon suivante :

```
et = new Etudiant();
```

Utilisation des classes

Au lieu de deux instructions, vous pouvez utiliser une seule instruction :

```
Etudiant et = new Etudiant();
```

A présent, on peut appliquer n'importe quelle méthode à l'objet **et**. par exemple, pour initialiser les attributs de **et**, on procède de la façon suivante :

```
et.initialiser("Oujdi","Mohammed","A8899",12.5);
```

Dans l'exemple suivant, on va utiliser la classe **ExempleEtudiant**, pour tester la classe **Etudiant**.

Exemple :

```
public class ExempleEtudiant {
    public static void main(String[] args) {
         double moy;
        Etudiant et = new Etudiant();
        et.initialiser("Oujdi","Ali","A8899",12.5);
        et.afficher();
        et.initialiser("Berkani","Lina","A7788",13);
        moy = et.getMoyenne();
        System.out.println("Moyenne : "+moy);
    }
}
class Etudiant {
    ...
}
```

Exécution :

Le résultat de l'exécution du programme précédent est le suivant :

```
Nom : Oujdi
Prenom : Ali
CNE : A8899
Moyenne : 12.5
Moyenne : 13.0
```


Initialisation des objets

Lors de la création d'un objet, tous les attributs sont initialisés par défaut. Dans les sections suivantes on verra comment initialiser les attributs lors de la création d'un objet.

Type	boolean	char	byte	short	int	long
valeur par défaut	false	`'\u0000'`	(byte)0	(short)0	0	0L

Type	float	double	objet
valeur par défaut	0.0f	0.0	null

Portée des attributs

Les attributs sont accessibles à l'intérieure de toutes les méthodes de la classe. Il n'est pas nécessaire de les passer comme arguments.

A l'extérieure des classes, les attributs privés (**private**) ne sont pas accessibles. Pour l'exemple de la classe **Etudiant**, une instruction de type :

```
Etudiant et = new Etudiant();
moy = et.moyenne;
```

aboutit à une erreur de compilation (`The field Etudiant.moyenne is not visible`).

Surcharge des méthodes

On parle de surcharge, lorsque plusieurs méthodes possèdent le même nom. Ces différentes méthodes **ne doivent pas** avoir le même nombre d'arguments ou des arguments de **même** types. On parle de **signature** de la méthode.

Exemple :

On va ajouter à la classe **Etudiant** trois méthodes qui portent le même nom. Une méthode qui contient :

1. trois arguments de types **double** ;
2. deux arguments de types **double** ;
3. deux arguments de types **float**.

Exemple

```
class Etudiant {
    ...
    //calcul de la moyenne de trois nombres
    public double calculMoy(double m1, double m2,
       double m3) {
        ...
    }
    //calcul de la moyenne de deux nombres (doubles)
    public double calculMoy(double m1, double m2) {
        ...
    }
    //calcul de la moyenne de deux nombres (float)
    public double calculMoy(float m1, float m2) {
        ...
    }
}
```


Utilisation de la classe Etudiant

Dans l'exemple suivant, on va utiliser la classe **ExempleEtudiant**, pour tester la classe **Etudiant** modifiée.

Exemple

```
public class ExempleEtudiant {
    public static void main(String[] args) {
        double moy;
        Etudiant et = new Etudiant();
        et.initialiser("Oujdi","Ali","A8899",12.5);
        //Appel de calculMoy(double, double, double)
        moy = et.calculMoy(10.5,12,13.5);
        //Appel de calculMoy(double, double)
        moy = et.calculMoy(11.5,13);
        //Appel de calculMoy(float, float)
        moy = et.calculMoy(10.5f,12f);
        //Appel de calculMoy(double, double)
        //13.5 est de type double
        moy = et.calculMoy(11.5f,13.5);
    }
}
```


Conflit

Considérons la classe **Etudiant** qui contient deux méthodes. Chaque méthode contient :

1. deux arguments, un de type **double** et l'autre de type **float** ;
2. deux arguments, un de type **float** et l'autre de type **double**.

```
class Etudiant {
    ...
    public double calculMoy(double m1, float m2) {
        ...
    }

    public double calculMoy(float m1, double m2) {
        ...
    }
}
```

Exemple d'utilisation

```
public class ExempleEtudiant {
    public static void main(String[] args) {
        double moy;
        Etudiant et = new Etudiant();
        et.initialiser("Oujdi","Ali","A8899",12.5);

        //Appel de calculMoy(double m1, float m2)
        moy = et.calculMoy(11.5,13f);

        //Appel de calculMoy(float m1, double m2)
        moy = et.calculMoy(10.5f,12.0);
```


Exemple d'utilisation

```
        //11.5 et 13.5 sont de type double
        //Erreur de compilation
        moy = et.calculMoy(11.5,13.5);

        //11.5f et 13.5f sont de type float
        //Erreur de compilation
        moy = et.calculMoy(11.5f,13.5f);
 }
}
```

Exemple d'utilisation

L'exemple précédent conduit à des erreurs de compilation :

- `moy = et.calculMoy(11.5,13.5)` aboutit à l'erreur de compilation : `The method calculMoy(double, float) in the type Etudiant is not applicable for the arguments (double, double);`
- `moy = et.calculMoy(11.5f,13.5f)` aboutit à l'erreur de compilation : `The method calculMoy(double, float) is ambiguous for the type Etudiant.`

Lecture à partir du clavier

Pour lire à partir du clavier, il existe la classe **Scanner**. Pour utiliser cette classe, il faut la rendre visible au compilateur en l'important en ajoutant la ligne :

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

nextShort()	permet de lire un **short**
nextByte()	permet de lire un **byte**
nextInt()	permet de lire un **int**
nextLong()	permet de lire un **long**. Il n'est pas nécessaire d'ajouter L après l'entier saisi.
nextFloat()	permet de lire un **float**. Il n'est pas nécessaire d'ajouter F après le réel saisi.
nextDouble()	permet de lire un **double**
nextLine()	permet de lire une ligne et la retourne comme un **String**
next()	permet de lire la donnée suivante comme **String**

Exemple :

```
import java.util.Scanner;
public class TestScanner
{
    public static void main(String[] args)
    {
        String nom;
        int age;
        double note1,note2,moyenne;

        /*clavier est un objet qui va permettre la
           saisie clavier
        vous pouvez utiliser un autre nom (keyb,
           input, ...)*/
        Scanner clavier = new Scanner(System.in);
```


Exemple :

```
        System.out.print("Saisir votre nom : ");
        nom = clavier.nextLine();
        System.out.print("Saisir votre age : ");
        age = clavier.nextInt();
        System.out.print("Saisir vos notes : ");
        note1 = clavier.nextDouble();
        note2 = clavier.nextDouble();
        moyenne = (note1+note2)/2;
        System.out.println("Votre nom est " + nom +
           ", vous avez " + age + " ans et vous avez
            obtenu "+ moyenne);
        clavier.close(); //fermer le Scanner
    }
}
```

Problèmes liées à l'utilisation de **nextLine()**

Reprenons l'exemple précédent et au lieu de commencer par la saisie du nom, on commence par la saisie de l'age.

```
Scanner clavier = new Scanner(System.in);
System.out.print("Saisir votre age : ");
age = clavier.nextInt();
System.out.print("Saisir votre nom : ");
nom = clavier.nextLine();
System.out.print("Saisir vos notes : ");
note1 = clavier.nextDouble();
note2 = clavier.nextDouble();
moyenne = (note1+note2)/2;
System.out.println("Votre nom est " + nom +
   ", vous avez " + age + " ans et vous avez
    obtenu "+ moyenne);
```


L'exécution du précédent programme est la suivante :

```
 Saisir votre age : 23
Saisir votre nom : Saisir vos notes : 12
13
Votre nom est , vous avez 23 ans et vous avez obtenu
12.5
```

Lors de la saisie de l'age, on a validé par **Entrée**. La touche **Entrée** a été stocké dans **nom** !

Pour éviter ce problème, il faut mettre `clavier.nextLine()` avant nom = clavier.nextLine();.

```
public class TestScanner
{
    public static void main(String[] args)
    {
        ...
        age = clavier.nextInt();
        System.out.print("Saisir votre nom : ");
        clavier.nextLine();
        nom = clavier.nextLine();
        ...
    }
}
```

Chapitre 3
Constructeurs

Introduction

On a vu dans le chapitre 2, que pour initialiser les attributs de la classe **Etudiant**, on a définit une méthode **initialiser()**.

```
class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    //Initialisation
    public void initialiser(String x, String y,
       String z, double m) {
        nom = x;
        prenom = y;
        cne = z;
        moyenne = m;
    }
    ...
}
```


Introduction

Cette façon de faire n'est pas conseillé pour les 2 raisons suivantes :

1. pour chaque objet créé, on doit l'initialisé en appelant la méthode d'initialisation ;
2. si on oublie d'initialiser l'objet, il sera initialisé par défaut, ce qui peut poser des problèmes lors de l'exécution du programme. Du fait que le programme sera compilé sans erreurs.

Pour remédier à ces inconvénients, on utilise les **constructeurs**.

Définition

Un **constructeur** est une méthode, sans type de retour, qui porte le même nom que la classe. Il est invoqué lors de la déclaration d'un objet.

Une classe peut avoir plusieurs constructeurs (**surcharge**), du moment que le nombre d'arguments et leurs types n'est pas le même.

Exemple

```
class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    //Constructeur
    public Etudiant(String x, String y, String
        z, double m) {
        nom = x;
        prenom = y;
        cne = z;
        moyenne = m;
    }
    ...
}
```

Exemple

Pour créer un objet et l'initialiser, on remplace les deux instructions :

```
Etudiant et = new Etudiant();

et.initialiser("Oujdi","Ali","A8899",12.5);
```

par l'instruction :

Etudiant et = new Etudiant("Oujdi","Ali","A8899",12.5);

Utilisation de **this**

Dans les arguments du constructeur **Etudiant**, on a utilisé : x, y, z et m. On peut utiliser les mêmes noms que les attributs privés de la classe en faisant appel au mot clés **this**.

Exemple

```
class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    //Constructeur
    public Etudiant(String nom, String prenom,
        String cne,
            double moyenne) {
        this.nom = nom;
        this.prenom = prenom;
        this.cne = cne;
        this.moyenne = moyenne;
    }
    ...
}
```


Surcharge des constructeurs

`this.nom`, `this.prenom`, `this.cne` et `this.moyenne` correspondent aux attributs de la classe.

On va modifier la classe **Etudiant** pour qu'il possède deux constructeurs :

1. un à trois arguments ;
2. l'autre à quatre arguments.

Exemple 1

```
class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    //Constructeur 1
    public Etudiant(String nom, String prenom,
        String cne) {
        this.nom = nom;
        this.prenom = prenom;
        this.cne = cne;
    }
```


Exemple 1

```
    //Constructeur 2
    public Etudiant(String nom, String prenom,
        String cne, double moyenne) {
        this.nom = nom;
        this.prenom = prenom;
        this.cne = cne;
        this.moyenne = moyenne;
    }
    ...
}
```

Exemple 1

```
Etudiant et = new Etudiant("Oujdi","Ali","A8899"
    ,12.5);

//l'attribut moyenne est initialisé par défaut (0.0)
Etudiant et1 = new Etudiant("Berkani","Lina","A7799"
    );
```


Exemple 2

Dans le constructeur 2 de l'exemple 1, trois instructions ont été répétées. Pour éviter cette répétition, on utilise l'instruction :

this (arguments)

L'exemple 1 devient :

Exemple 2

```
class Etudiant {
    ...
    //Constructeur 1
    public Etudiant(String nom, String prenom,
       String cne) {
        this.nom = nom;
        this.prenom = prenom;
        this.cne = cne;
    }
    //Constructeur 2
    public Etudiant(String nom, String prenom,
       String cne, double moyenne) {
        //Appel du constructeur 1
        this(nom, prenom, cne);
        this.moyenne = moyenne;
    }
```


Remarque

L'instruction

this (arguments)

doit être la première instruction du constructeur. Si elle est mise ailleurs, le compilateur génère une erreur.

Constructeur par défaut

Le constructeur par défaut est un constructeur qui n'a pas d'arguments.

Exemple

```
public class ExempleEtudiant {
    public static void main(String[] args) {
        //Utilisation du constructeur par défaut
        Etudiant et = new Etudiant();
    }
}
class Etudiant {
    ...
    //Constructeur par defaut
    public Etudiant() {
        nom = "";
        prenom = "";
        cne = "";
        moyenne = 0.0;
    }
    //Autres constructeurs
```

Remarques

1. Si aucun constructeur n'est utilisé, le compilateur initialise les attributs aux valeurs par défaut.
2. Dans les exemples 1 et 2 de la section «surcharge des constructeurs», l'instruction
   ```
   Etudiant et = new Etudiant();
   ```
   n'est pas permise parce que les deux constructeurs ont des arguments.
3. Un constructeur ne peut pas être appelé comme les autres méthodes. L'instruction
   ```
   et.Etudiant("Oujdi","Mohammed","A8899");
   ```
   n'est pas permise.

Constructeur de copie

Java offre un moyen de créer la copie d'une instance en utilisant le constructeur de copie. Ce constructeur permet d'initialiser une instance en copiant les attributs d'une autre instance du même type.

Exemple

```
public class ExempleEtudiant {
    Etudiant et1 = new Etudiant("Oujdi","Ali","A88");
    Etudiant et2 = new Etudiant(et1);
}
class Etudiant {
    ...
    //Constructeur de copie
    public Etudiant(Etudiant autreEt) {
        nom = autreEt.nom;
        prenom = autreEt.prenom;
        cne = autreEt.cne;
        moyenne = autreEt.moyenne;
    }
    ...
}
```


Remarque

et1 et **et2** sont différents mais ont les mêmes valeurs pour leurs attributs.

Chapitre 4
Généralités

Attributs statiques

Les variables statiques sont appelés « variables de classe ». Elles sont partagées par toutes les instances de la classe. Pour chaque instance de la classe, il n'y a qu'une seule copie d'une variable statique par classe. Il n'y a pas de création d'une nouvelle place mémoire lors de l'utilisation de « new ».

Pour déclarer une variable statique, il faut faut utiliser le mot clés static .

Exemple :

```
class Jeu {
    static int meilleurScore = 0;
    int score = 0;
    void calculScore() {
        score += 10;
        if (meilleurScore < score)    meilleurScore
           = score;
    }
}
```


Exemple :

```
public static void main(String[] args) {
  System.out.println(Jeu.meilleurScore);//Affiche 0
  Jeu.meilleurScore++;
  System.out.println(Jeu.meilleurScore);//Affiche 1
  Jeu j = new Jeu();
  j.calculScore();
  System.out.println(Jeu.meilleurScore);//Affiche 10
  //ou bien
  System.out.println(j.meilleurScore); //Affiche 10
  j.calculScore();
  System.out.println(Jeu.meilleurScore);//Affiche 20
 }
```

Constantes final et static

Une constante commune à toutes les instances d'une classe peut être déclarée en « final static ».

```
class A{
    final static double PI=3.1415927;
    static final  double Pi=3.1415927;
}
```

La classe « **Math** » fourni les constantes statiques **Math.PI** (égale à 3.14159265358979323846) et **Math.E** (égale à 2.7182818284590452354).

Constantes final et static

Dans la classe « **Math** », la déclaration de **PI** est comme suit :

public final static double PI = 3.14159265358979323846;

PI est :

- public, elle est accessible par tous ;
- final , elle ne peut pas être changée ;
- static , une seule copie existe et elle est accessible sans déclarer d'objet Math.

Méthodes statiques

Une méthode peut être déclaré statique en la faisant précédé du mot clés static. Elle peut être appelée directement sans créer d'objet pour cette méthode. Elle est appelée « méthode de classe ».

Il y a des restrictions sur l'utilisation des méthodes statiques :

Restrictions

- elles ne peuvent appeler que les méthodes statiques ;
- elles ne peuvent utiliser que les attributs statiques ;
- elles ne peuvent pas faire référence à this et à super.

Exemple :

```
class Calcul {
    private int somme;
    static int factorielle(int n) {
        //ne peut pas utiliser somme
        if (n <= 0)
            return 1;
        else
            return n * factorielle(n - 1);
    }
}
```

Exemple :

```
public class MethodesClasses {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int n;
        System.out.print("Saisir 1 entier :  ");
        n = clavier.nextInt();
        System.out.print("Factorielle :  " + n +
        " est  :" + Calcul.factorielle(n));
    }
}
```


Classe Math

La classe « **Math** » fourni les méthodes statiques **sin**, **cos**, **pow**, ...

Fin de vie des objets

Un objet (ou une variable) est en fin de vie lorsqu'il n'est plus utilisé. Il est hors porté.

Exemple 1 :

```
{
    int x=12; // x est accessible
    {
        int q;
        q=x+100; // x et q tous les deux sont
            accessibles
    }
    x=6;
    // x est accessible
    q=x+2; // Erreur: q est hors de portee
}
```

Attention

Ceci n'est pas permis en Java

```
{
    int x=12;
    {
        //illegale en Java (Duplicate local variable
            x)
        //valable en C, C++
        int x=96;
    }
}
```


Exemple 2 :

```
public class FinVie {
    public static void main(String[] args) {
        afficherUnEtudiant();
    }

    static void afficherUnEtudiant() {
        Etudiant et = new Etudiant("Oujdi", "Ali", "
            A20");
        System.out.println(et);
    }
}
```

La référence associé à **et** n'est plus utilisée par contre l'objet référencé par **et** existe toujours mais reste inaccessible.

Ramasse miettes (Garbage collector)

Contrairement au langage C où on la fonction **free** qui permet de libérer la mémoire occupée par un pointeur, en Java, il n'y a pas de méthode qui permet de libérer la mémoire occupée par un objet non référencé.

Par contre il existe un processus qui est lancé automatiquement (de façon régulière) de l'exécution d'un programme Java et récupère la mémoire non utilisé. Ce processus s'appelle le **ramasse miettes** (Garbage collector en anglais).

L'utilisateur peut appeler le ramasse miette en appelant la méthode System.gc();.

Exemple :

```
public class FinVie {
    public static void main(String[] args) {
        ...
        afficherUnEtudiant();
        System.gc();
        ...
    }

    static void afficherUnEtudiant() {
        Etudiant et = new Etudiant("Oujdi", "Ali", "
            A20");
        System.out.println(et);
    }
}
```

Chapitre 5
Héritage

Introduction

Comme pour les pour autres langages orientés objets, Java permet la notion d'héritage, qui permet de créer de nouvelles classes à partir d'autres classes existantes. L'héritage permet de réutiliser des classes déjà définies en adaptant les attributs et les méthodes (par ajout et/ou par modification).

Une classe qui hérite d'une classe existante est appelée classe **dérivée**. Elle est aussi appelée **sous-classe** ou **classe-fille**.

La classe, dont hérite d'autres classes, est appelée classe **super-classe**. Elle est aussi appelée **classe-mère** ou **classe-parente**.

Syntaxe

```
class SousClasse extends SuperClass
```


Remarques :

- Java ne permet pas l'héritage multiple. C'est-à-dire, une classe ne peut pas hériter de plusieurs classes. Elle ne peut hériter que d'une seule classe.
- Une classe peut hériter d'une classe dérivée. Considérons la classe **A** qui est la **super-classe** de **B** et **B** qui est la **super-classe** de **C**.
  $\longrightarrow$ **A** est la **super-super-classe** de **C**.

classe A (super-classe de B)
↓
classe B (super-classe de C)
↓
classe C

classe A (super-super-classe de C)
↓
classe C

Exemple introductif

Considérons les deux classes : **Etudiant** et **Professeur**. Pour les deux classes :

1. les attributs **nom** et **prenom** sont en commun ;
2. les méthodes **afficher()** et **setNom()** sont en commun ;
3. la classe **Etudiant** contient l'attribut **cne** et la classe **Professeur** contient l'attribut **cin**.

Exemple : Classe Etudiant

```
class Etudiant {
    private String nom, prenom, cne;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
}
```

Exemple : Classe Professeur

```
class Professeur {
    private String nom, prenom, cin;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
    void setCin(String cin){
        this.cin = cin;
    }
}
```


Utilisation de l'héritage

Un étudiant et un professeur sont des personnes. Définissons une nouvelle classe **Personne** :

```
class Personne {
    private String nom, prenom;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+
            prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
}
```

Les deux classes peuvent être modifiées en utilisant la classe **Personne** . Elle deviennent comme le montre le listing suivant :

```
class Etudiant extends Personne {
    private String cne;
    void setCne(String cne){
        this.cne = cne;
    }
}

class Professeur extends Personne{
    private String cin;

    void setCin(String cin){
        this.cin = cin;
    }
}
```


Accès aux attributs

L'accès aux attributs privés (**private**) d'une super-classe n'est pas permis de façon directe. Supposons qu'on veut définir, dans classe **Etudiant**, une méthode **getNom()** qui retourne le nom alors, l'instruction suivante n'est pas permise puisque le champ **Personne.nom** est non visible (`The field Personne.nom is not visible`).

```
class Etudiant extends Personne {
    private String cne;
    String getNom() {
        return nom; //non permise
    }
}
```

Accès aux attributs

Pour accéder à un attribut d'une **super-classe**, il faut soit :

- rendre l'attribut publique, ce qui implique que ce dernier est accessible par toutes les autres classes ;
- définir, dans la classe **Personne**, des méthodes qui permettent d'accéder aux attributs privés (**getters** et **setters**) ;
- déclarer l'attribut comme protégé en utilisant le mot clés **protected**.

Exemple

La classe **Etudiant** peut accéder à l'attribut **nom** puisqu'il est protégé.

```
class Personne {
    protected String nom;
    ...
}
class Etudiant extends Personne {
    ...
    String getNom() {
        return nom;
    }
}
```

Remarques :

1. Un attribut protégé est accessible par toutes les sous-classes et par toutes les classes du même paquetage (on verra plus loin la notion de **package**) ce qui casse l'encapsulation.
2. Le mode protégé n'est pas très utilisé en Java.

Héritage hiérarchique

Comme mentionné dans l'introduction, une classe peut être la **super-super-classe** d'une autre classe. Reprenons l'exemple concernant l'héritage et ajoutons la classe **EtudiantEtranger**. Un étudiant étranger est lui aussi un étudiant dont on veut lui ajouter la nationalité .

```
class Personne {
  ...
}
class Etudiant extends Personne {
    ...
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    ...
}
```

Définitions

Masquage (shadowing)

un attribut d'une sous-classe qui porte le même nom qu'un autre attribut de la super-classe.

Peu utilisé en pratique par-ce-qu'il est source d'Ambiguïté .

Redéfinition (overriding)

comme pour le cas de surcharge à l'intérieure d'une classe, une méthode déjà définie dans une super-classe peut avoir une nouvelle définition dans une sous-classe.

Remarques :

1. Il ne faut pas confondre surcharge et redé nition !
2. On verra plus de détails concernant la redé nition dans le chapitre concernant le polymorphisme.

Exemple : masquage

```
public class Masquage {
    public static void main(String[] args){
        Masquage masq = new Masquage();
        masq.affiche();   //Affichera 10
        masq.variableLocal();//Affichera 20
    }
    private int a = 10;
    public void affiche()
    {
        System.out.println("a = " + a);
    }
    public void variableLocal()
    {
        int a = 20; //variable locale
        System.out.println("a = " + a);
    }
}
```


Redéfinition : Exemple 1

```
class A
{
  public void f(int a, int b)
  {
    //instructions
  }
  //Autres methodes et attributs
}
class B extends A
{
  public void f(int a, int b)
  {
    //la methode redifinie f() de la
       super-classe
  }
  //Autres methodes et attributs
}
```

Redéfinition : Exemple 2

Reprenons la classe **Personne** et ajoutons à cette classe la méthode **afficher()** qui permet d'afficher le nom et le prénom.

Dans les classes **Etudiant**, **EtudiantEtranger**, et **Professeur**, la méthode **afficher()** peut être définie avec le même nom et sera utilisé pour afficher les informations propres à chaque classe.

Pour ne pas répéter les instructions se trouvant dans la méthode de base, il faut utiliser le mot clé **super()**.

Redéfinition : Exemple 2

```
class Personne {
    private String nom, prenom;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    ...
}

class Etudiant extends Personne {
    private String cne;
    void afficher() {
        super.afficher();
        System.out.println("CNE :"+cne);
    }
}
```

Redéfinition : Exemple 2 (suite)

```
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    void afficher() {
        super.afficher();
        System.out.println("Nationalite :"+
            nationalite);
    }
    ...
}

class Professeur extends Personne{
    private String cin;
    void afficher() {
        super.afficher();
        System.out.println("CIN :"+cin);
    }
}
```


Remarques :

1. La méthode **super.afficher()** doit être la première instruction dans la méthode **afficher()**.
2. La méthode **super.afficher()** de la classe **EtudiantEtranger**, fait appel à **afficher()** de la classe **Etudiant**.
3. Si la classe **Etudiant** n'avait pas la méthode **afficher()**, alors, par transitivité, la méthode **super.afficher()** de la classe **EtudiantEtranger**, fait appel à **afficher()** de la classe **Personne**.
4. Il n'y a pas de : **super.super**.

Héritage et constructeurs

Une sous-classe n'hérite pas des constructeurs de la super-classe.

Exemple 1

Reprenons à la classe **Personne** et ajoutons à cette classe un seul constructeur.

Si aucun constructeur n'est défini dans les classes **Etudiant** et **Professeur**, il y aura erreur de compilation.

```
class Personne {
    private String nom, prenom;
    //Constructeur
    public Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    ...
}
class Etudiant extends Personne {
    ...
    //Pas de constructeur
    ...
}
class Professeur extends Personne{
    ...
    //Pas de constructeur
    ...
```


Exemple 2

```
class Personne {
    private String nom, prenom;
    //Constructeur
    public Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    ...
}
```

```
class Etudiant extends Personne {
    private String cne;
    //Constructeur
    public Etudiant(String nom, String prenom,
       String cne) {
        super(nom, prenom);
        this.cne = cne;
    }
    ...
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    //Constructeur
    public EtudiantEtranger(String nom, String
       prenom, String cne, String nationalite) {
        super(nom, prenom,cne);
        this.nationalite = nationalite;
}   }
```


Dans cette exemple, le constructeur de la classe **EtudiantEtranger** fait appel au constructeur de la classe **Etudiant** qui à son tour fait appel au constructeur de la classe **Personne**.

Exemple 3

```
class Rectangle{
    private double largeur;
    private double hauteur;
    public Rectangle(double l, double h) {
        largeur = l;
        hauteur = h;
    }
    ...
}

class Carre extends Rectangle {
    public Carre(double taille) {
        super(taille, taille);
    }
    ...
}
```


Remarques :

1. **super** doit être la première instruction dans le constructeur et ne doit pas être appelé 2 fois.
2. Il n'est pas nécessaire d'appeler **super** lorsque la super-classe admet un constructeur par défaut. Cette t che sera réalisée par le compilateur.
3. Les arguments de super doivent être ceux d'un des constructeur de la super-classe.
4. Aucune autre méthode ne peut appeler **super(...)**.
5. Il n'y a pas de : **super.super**.

Opérateur « instanceof »

Types primitifs

Pour les types primitifs, les instructions suivantes sont vraies :

```
int i;
float x;
double y;
...
x = i;
y = x;
```

Un **int** est un **float** et un **float** est un **double** (conversion implicite).

Types primitifs

Par contre, les instructions suivantes sont fausses :

```
i = x;
x = y;
```

Un **float** n'est pas un **int** et un **double** n'est pas un **float**.

Pour les utiliser, il faut faire une conversion explicite (faire un **cast**) :

```
i = (int) x;
x = (float) y;
```


Objets

si **B** est sous-classe de **A**, alors on peut écrire :

```
//a est de type « A », mais l'objet référencé par a est de type « B ».
A a = new B(...);
A a1;
B b = new B();
a1 = b; // a1 de type « A », référence un objet de type « B »
```

Par contre, on ne peut pas avoir :

```
A a=new A();
B b;
b=a;
// erreur : on ne peut pas convertir du type « A » vers le type « B »
```

Cas des tableaux (Voir chapitre 7)

Reprenons l'exemple des classes **Personne**, **Etudiant**, **EtudiantEtranger** et **Professeur**. Les instructions suivantes sont vraies :

```
Etudiant e = new Etudiant();
EtudiantEtranger eEtr = new EtudiantEtranger();
Professeur prof = new Professeur();

Personne[] P = new Personne[3];
P[0] = e;
P[1] = eEtr;
P[2] = prof;

for (int i=0; i<3; i++)
    P[i].info();
```


Cas des tableaux

La méthode **info()** est ajoutée aux différentes classes pour indiquer dans quelle classe on se trouve :

```
void info(){
        System.out.println("Classe ...");
}
```

On reviendra plus en détails sur l'utilisation des tableaux dans le chapitre concernant les tableaux.

instanceof

Si « B » est une sous classe de « A » alors l'instruction :

b instanceof A ; retourne true.

```
Etudiant e = new Etudiant();
boolean b = e instanceof Personne; //b --> true

EtudiantEtranger eEtr =new EtudiantEtranger();
b = eEtr instanceof Personne;  //b --> true

Personne personne = new Etudiant();
b = personne  instanceof Etudiant; //b --> true

personne = new Personne();
b = personne  instanceof Etudiant; //b --> false
```


Cas de la classe **Object**

L'instruction :

personne instanceof Object;

retourne « true » car toutes les classes héritent, par défaut, de la classe Object

Chapitre 6
Polymorphisme et abstraction

Introduction

Le mot polymorphisme vient du grecque : **poly** (pour plusieurs) et **morph** (forme). Il veut dire qu'une même chose peut avoir différentes formes. Nous avons déjà vue cette notion avec la redéfinition des méthodes. Une même méthode peut avoir différentes définitions suivant la classe où elle se trouve.

Exemple introductif

Soient les classes **Personne**, **Etudiant** et **EtudiantEtranger** définies dans les chapitres précédents :

```
class Personne {
  ...
}
class Etudiant extends Personne {
    ...
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    ...
}
```


Exemple introductif

Puisque un étudiant étranger est lui aussi un étudiant, au lieu de définir 2 tableaux :

```
Etudiant[] etudiants = new Etudiant[30];
EtudiantEtranger[] etudiantsEtrangers = new EtudiantEtranger[10];
```

on pourra définir un seul tableau comme suit :

```
Etudiant[] etudiants = new Etudiant[40];
```

Exemple introductif

Avant d'utiliser le tableau précédent, considérons la déclaration :

Personne personne;

Nous avons vu que les instructions suivantes sont toutes valides :

personne = new Personne();

personne = new Etudiant();

personne = new EtudiantEtranger();

De la même façon on peut initialiser le tableau de la façon suivante :

```
for(int int i=0; i<10; i++)
    etudiants[i] = new Etudiant();

etudiants[10] = new EtudiantEtranger();
etudiants[11] = new EtudiantEtranger();
...
```


Liaison dynamique

Considérons la classe **B** qui hérite de la classe **A** :

```
class A{
  public void message(){
     System.out.println("Je suis dans la classe A");
  }
}

class B extends A{
  public void message(){
    System.out.println("Je suis dans la classe B");
  }
  public void f(){
    System.out.println("Methode f de la classe B");
  }
}
```

Liaison dynamique

La méthode « message() » a été redéfinie dans la classe **B** et la méthode « f() » a été ajoutée dans B.

Considérons les instructions :

```
public static void main(String[] args) {
    A a = new A();
    B b = new B();
    a.message();
    b.message();
    b.f();
    a = new B();
    a.message();
}
```


Liaison dynamique

Dans l'exécution on aura le résultat suivant :

```
Je suis dans la classe A
Je suis dans la classe B
Methode f() de la classe B
Je suis dans la classe B
```

Lorsqu'une méthode est redéfinie (s'est spécialisée), c'est la version la plus spécialisée qui est appelée. La recherche de la méthode se fait dans la classe réelle de l'objet. La recherche s'est fait lors de l'exécution et non lors de la compilation. Ce processus s'appelle la **liaison dynamique**.

Il s'appelle aussi : **liaison tardive, dynamic binding, late-binding** ou **run-time binding**.

Remarques :

- Dans les instructions précédentes, la dernière instruction « a.message(); » fait appel à la méthode « message() » de la classe **B**.
- Si on ajoute l'instruction :
  a.f() ;
  après l'instruction :
  a = new B();
  on aura une erreur de compilation, du fait que la méthode « f() » n'est pas implémentée dans la classe de déclaration de l'objet « a » même si la classe réelle (la classe B) possède « f() ».
- Pour éviter l'erreur précédente, il faut faire un cast :
  ((B) a).f() ;

Remarques :

- La visibilité d'une méthode spécialisée peut être augmentée (par exemple de protected vers public) mais elle ne peut pas être réduite (par exemple de public vers private)

Méthodes de classes

Il n'y a pas de polymorphisme avec les méthodes de classes !

Exemple

```
class Etudiant {
    public void message(){
        System.out.println("Je suis un etudiant");
    }
}

class EtudiantEtranger extends Etudiant {
    public void message(){
        System.out.println("Je suis un etudiant
            etranger");
    }
}
```

Exemple

```
public class MethodesInstances {
    public static void main(String[] args){
        Etudiant e = new Etudiant();
        e.message();

        e = new EtudiantEtranger();
        e.message();
    }
}
```

L'exécution du programme précédent donnera :

```
Je suis un etudiant
Je suis un etudiant etranger
```


Si on modifie la méthode « **message()** » de la classe **Etudiant**, en la rendant statique :

```
class Etudiant {
    public static void message(){
       System.out.println("Je suis un etudiant");
    }
}

class EtudiantEtranger extends Etudiant {
    public static void message(){
       System.out.println("Je suis un etudiant
          etranger");
    }
}
```

```
public class MethodesInstances {
    public static void main(String[] args){
        Etudiant e = new Etudiant();
        e.message();
        e = new EtudiantEtranger();
        e.message();
    }
}
```

alors l'exécution du programme précédent donnera :

```
Je suis un etudiant
Je suis un etudiant
```

C'est la classe du type qui est utilisée (ici Etudiant) et non du type réel de l'objet (ici EtudiantEtranger).

Abstraction

Exemples introductifs

Reprenons les classes Personne Etudiant et EtudiantEtranger et ajoutons aux classes Etudiant et EtudiantEtranger la méthode saluer() :


```
class Etudiant extends Personne {
    ...
    public void saluer() {
        System.out.println("Assalam alaikoum");
    }
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    ...
    public void saluer() {
        System.out.println("Bonjour");
    }
}
```

Puisque la méthode « saluer() » est définie dans les deux classes **Etudiant** et **EtudiantEtranger**, on souhaite la définir dans la classe **Personne**.

Solution 1

Une première solution consiste à la définir comme suit :

```
class Personne {
    ...
    public void saluer() {
    }
}
```

Cette solution est mauvaise du fait que toute classe qui héritera de **Personne** pourra appeler cette méthode (qui ne fait rien !).

Solution 2

Une deuxième solution consiste à rendre la méthode « saluer() » abstraite dans **Personne** et par conséquent, obliger chaque classe qui hérite de **Personne** à définir sa propre méthode.

Remarques :

- Une méthode abstraite :
  - ne doit contenir que l'entête et doit être implémenté dans les sous classes ;
  - doit être public (ne peut pas être privée) ;
  - est déclarée comme suit :
    public abstract typeRetour methAbstr(args);
- Une méthode statique ne peut pas être abstraite.
- Une classe qui contient une méthode abstraite doit être elle aussi abstraite et doit être déclarée comme suit :
  public abstract class nomClasse{ ... }

Remarques :

- Une classe abstraite ne peut pas être utilisée pour instancier des objets. Une instruction telle que :
  obj=new nomClasse();
  est incorrecte (avec nomClasse est une classe abstraite).
- Une sous-classe d'une classe abstraite doit implémenter toutes les méthodes abstraites sinon elle doit être déclarée abstraite.

Solution 2 :

La classe **Personne** devient comme suit :

```
abstract class Personne {
    private String nom, prenom;
    public abstract void saluer();
    //...
}
```


Exemple 2

Considrons les classes **Cercle** et **Rectangle** ui sont des sous classes de la classe **FigureGeometrique**. Les surfaces d'un cercle et d'un rectangle ne sont pas calcules de la mme faon.

→ ne solution consiste dnir dans la classe **FigureGeometrique** une mthode abstraite surface() et obliger les classes **Cercle** et **Rectangle** implmenter cette mthode.

Exemple 2

```
abstract class FigureGeometrique {
    public abstract double surface();
}

class Cercle extends FigureGeometrique {
    private double rayon;

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

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


Exemple 2

```
class Rectangle extends FigureGeometrique {
    public double largeur, longueur;

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

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

Constructeurs et abstraction

Il est possible d'appeler une méthode abstraite dans le corps d'un constructeur, ceci est cependant déconseillé !

Exemple :

```
abstract class classeA {
    public abstract void m();

    public classeA() {
        m();
    }
}
```


Constructeurs et abstraction

```
class classeB extends classeA {
    private int b;
    public classeB() {
        // classeA() est invoquee implicitement
        // super(); (automatique)
        b = 1;
    }
    // definition de m pour classeB
    public void m() {
        System.out.println("b vaut : " + b);
    }
}
```

Constructeurs et abstraction

```
public class ConstructAbstraction {
    public static void main(String[] args) {
        classeB b = new classeB();
    }
}
```

Le résultat de l'exécution du programme précédent est :

```
b vaut : 0
```


Remarque

La classe Math n'est pas une classe abstraite même si on ne peut pas créer une instance de cette classe. Pour définir une classe non instanciable , il suffit de lui ajouter un et un seul constructeur privé sans arguments.

Remarque

Extrait de la classe **Math** :

```
public final class Math {

    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}

    ...
}
```

Toute classe abstraite est non instanciable mais l'inverse n'est pas vrai.

Chapitre 7
Tableaux

Déclaration et initialisation

Comme pour les autres langages de programmation, java permet l'utilisation des tableaux.

La déclaration d'un tableau à une dimension se fait de deux façons équivalentes :

`type tab[];` ou `type[] tab;`

`tab` est une référence à un tableau.

Exemple :

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

Contrairement au langage C, la déclaration `int tab[10];` n'est pas permise. On ne peut pas fixer la taille lors de la déclaration.

Création

La création d'un tableau peut se faire soit, lors de la déclaration soit par utilisation de l'opérateur **new**.

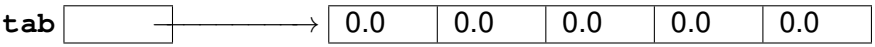
Exemples :

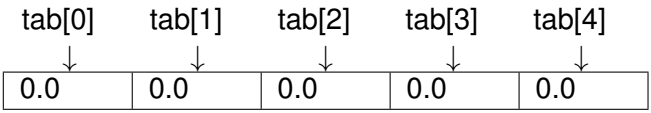
1. Création par initialisation au début :
   ```
   int [] tab={12,10,30*4};
   ```
2. Création par utilisation de **new** :
   ```
   int [] tab;
   tab = new int [5];
   ```
   ou bien :
   ```
   int [] tab = new int [5];
   ```


La déclaration

```
double[] tab = new double[5];
```

Crée un emplacement pour un tableau de 5 réels (double) et fait la référence à **`tab`** comme illustré par le schéma suivant :





Les valeurs du tableau sont initialisés aux valeurs par défaut (0 pour int, 0.0 pour double ...).

Déclaration mixte

On peut combiner la déclarations de tableaux avec d'autres variables de même type. La déclaration :

```
double scores[], moyenne;
```

crée :

- un tableau non initialisé de type **double** nommé **scores** ;
- une variable de type **double** nommée **moyenne**.

Par contre, la déclaration :

```
double[] scores, moyenne;
```

crée deux tableaux non initialisés.

Taille d'un tableau

La taille d'un tableau est accessible par le champ « public final » length.

Exemple :

```
double[] scores = new double[10];
System.out.println(scores.length); // Affiche 10
```

Remarque :

La taille d'un tableau ne peut pas être changée. Par contre, la référence du tableau peut changer. Elle peut référencée un tableau de taille différente.

Exemple :

```
double[] tab1 = new double[10];
double[] tab2 = new double[5];
tab1 = new double[7];//tab1 est maintenant un nouveau tableau de taille 7
tab2 = tab1;// tab1 et tab2 referencent le meme tableau
```

Parcours d'un tableau

Comme pour le langage C, on peut accéder aux éléments d'un tableau en utilisant l'indice entre crochets ([]).

Exemple

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

for(int i=0; i<tab.length;i++)
    System.out.printf("tab[%d] = %.2f\n",i,tab[i]);
```

Remarque :

La lecture d'un tableau peut se faire de la façon suivante :

```
for(double carre: tab)
    System.out.println(carre);
```

Cette méthode de parcours d'un tableau n'est valable que pour la lecture et ne peut pas être utilisée pour la modification. Elle a l'avantage d'éviter l'utilisation des indices.

Correspond à : « pour chaque élément ... » (for each ...)

Exemple

Dans le listing suivant, pour calculer la somme des éléments du tableau, on n'a pas besoin de connaître les indices.

```
double somme = 0.0;
for(double carre : tab)
    somme += carre;

System.out.println("Somme ="+somme);
```

Parcours des arguments de la ligne de commande

La méthode principale « main() » contient un tableau de « String ». Ce tableau peut être utilisé comme les autres tableaux. Pour afficher son contenu, on peut utiliser le code suivant :

```
System.out.println("taille de args "+args.length);
for(String arg : args)
    System.out.print(arg.toUpperCase()+" ");
```

Si le programme s'appel « Test » et on exécute la commande :

```
java Test bonjour tous le monde
```

alors le code précédent affichera :

```
taille de args : 4
BONJOUR TOUS LE MONDE
```


Copie de tableaux

Comme on l'a vu précédemment, pour copier le contenu d'un tableau « tab2 » dans autre tableau « tab1 », l'instruction :

tab1 = tab2;

ne copie que la référence et non le contenu. Pour copier le contenu, on peut soit, procéder de façon classique et copier élément par élément, soit utiliser des méthodes prédéfinies.

Utilisation de : System.arraycopy()

Sa syntaxe est :

System.arraycopy(src, srcPos, dest, destPos, nb);

Les arguments sont définis comme suit :

- **src** : tableau source ;
- **srcPos** : indice du premier élément copié à partir de src ;
- **dest** : tableau destination ;
- **destPos** : indice de destination où sera copié le premier élément ;
- **nb** : nombre d'éléments à copier.

Exemple

```
double [] carres = new double[10];
double [] carresBis = new double[10];
double [] carresTest = new double[6];

for(int i=0; i<carres.length;i++)
    carres[i]=i*i;

//Copie de carres dans carresBis
System.arraycopy(carres, 0, carresBis, 0, 10);

//Copie de 6 elements de carres a partir de l'indice
    4 dans carresTest
//(a partir du premier element)
System.arraycopy(carres, 4, carresTest, 0, 6);
```

Utilisation de : Arrays.copyOf() ou Arrays.copyOfRange()

La classe **java.util.Arrays** contient différentes méthodes pour la manipulation des tableaux (copie, trie, ...). Pour copier un tableau :

- en entier, on peut utiliser la méthode **Arrays.copyOf()** ;
- juste une partie, on peut utiliser la méthode **Arrays.copyOfRange()**.

Exemple

```
import java.util.Arrays;
...
double [] t1 = new double[10];

for(int i = 0; i < t1.length;i++)
    t[i] = i*i;

int[] t2 = Arrays.copyOf(t1, 10);
```

1. crée le tableau t2
2. affecte au tableau t2 les 10 premiers éléments du tableau tab.

Arrays.copyOfRange

```
int[] t3= Arrays.copyOfRange(t1, debut, fin);
```

1. crée le tableau t3
2. affecte à t3 les éléments de t1 situés entre les indices : debut et (fin-1) :

Comparer le contenu de deux tableaux de types primitifs

Comme pour le cas du copie, pour comparer le contenu de deux tableaux de types primitifs, on peut, soit :

- procéder de façon classique et comparer les éléments des deux tableaux un à un ;
- utiliser la méthode **equals()** de la classe **Arrays**.

Exemple :

Soient t1 et t2 deux tableaux de primitifs déjà initialisés :

```
import java.util.Arrays;
...
boolean b=Arrays.equals(tab1,tab2);
```

Utilisation de la classe Arrays

La classe **Arrays** contient différentes méthodes pour la manipulation des tableaux. Nous avons déjà utilisé les méthodes **Arrays.copyOf()**, **Arrays.copyOfRange()**, et **Arrays.equals()**. Dans ce qui suit, nous allons voir quelques méthodes pratiques :

Trie

La méthode **void sort(type[] tab)**, permet de trier le tableau **tab** par ordre croissant. Le résultat du trie est retourné dans **tab**.

La méthode **void sort(type[] tab, int indiceDeb, int indiceFin)** permet de trier le tableau **tab** par ordre croissant à partir de l'indice **indiceDeb** (inclue) jusqu'au **indiceFin** (exclue)

Exemple

```
double[] tab = new double[10];

for(int i=0; i<10;i++)
    tab[i] = Math.random()*10;
//Math.random() : genere un nombre aleatoire.

Arrays.sort(tab);

for(int i=0; i<10;i++)
    tab[i] = Math.random()*10;

Arrays.sort(tab, 2, 5);
//Trie les elements tab[2], tab[3] et tab[4] par
   ordre croissant
```


Recherche

La méthode **int binarySearch(type [] a, type val)**, permet de chercher **val** dans le tableau **tab**. Le tableau doit être trié par ordre croissant, sinon, le résultat de retour sera indéterminé.

Le résultat de retour est :

- l'indice du tableau qui contient **val** si le tableau contient **val** ;
- une valeur négative si le tableau ne contient pas **val**.

La méthode **int binarySearch(type [] a, int indiceDeb, int indiceFin, type val)**, permet de chercher **val** dans l'intervalle du tableau **tab** entre l'indice **indiceDeb** (inclue) et **indiceFin** (exclue).

Exemple

```
double[] tab = new double[10];
double val=5;

for(int i=0; i<10;i++)
    tab[i] = Math.random()*10;

tab[3] = val;

Arrays.sort(tab);

//Recherche dans le tableau
System.out.println(Arrays.binarySearch(tab, val));

//Recherche dans l'intervalle 2..7
System.out.println(Arrays.binarySearch(tab, 2, 7,
   val));
```


Remplissage

La méthode **void fill(type[] tab, type val)** affecte **val** à tous les éléments du tableau.

La méthode **void fill(type[] tab, int indiceDeb, int indiceFin, type val)** affecte **val** aux éléments du tableau compris entre **indiceDeb** et **indiceFin-1**.

```
double[] tab = new double[10];
double val=10;

Arrays.fill(tab, val);

val=7;
//tab[2]=tab[3]=tab[4]=7
Arrays.fill(tab,2,5,val);
```

Méthode toString()

La méthode **void toString(type[] tab)** permet de convertir le tableau en une chaîne de caractères. Elle met le tableau entre [ ] avec les valeurs séparés par « , » (virgule suivie par espace).

```
int[] tab = {1, 2, 3, 4};

System.out.println(Arrays.toString(tab));
//Affiche : [1, 2, 3, 4]
```


Passage d'un tableau dans une méthode

Les tableaux sont des objets et par conséquent lorsqu'on les passe comme arguments à une méthode, cette dernière obtient une copie de la référence du tableau et par suite elle a accès aux éléments du tableau. Donc, toute modification affectera le tableau.

Exemple

Le programme suivant :

```
import java.util.Arrays;

public class TabMethodesArg {
    public static void main(String[] args) {
        int[] tab = { 1, 2, 3, 4 };

        System.out.print("Debut de main: ");
        System.out.println(Arrays.toString(tab));

        // Appel de test
        test(tab);

        System.out.print("Fin de main: ");
        System.out.println(Arrays.toString(tab));
    }
```


Exemple (suite)

```
    public static void test(int[] x) {
        System.out.println("Debut :\t" + Arrays.
            toString(x));
        Arrays.fill(x, 10);
        System.out.println("fin :\t" + Arrays.
            toString(x));
    }
}
```

affichera :

```
Debut de main: [1, 2, 3, 4]
Debut :    [1, 2, 3, 4]
fin :    [10, 10, 10, 10]
Fin de main: [10, 10, 10, 10]
```

Retour d'un tableau dans une méthode

Une méthode peut retourner un tableau.

Exemple

La méthode :

```
public static int [] scoreInitial() {
    int score[] ={2, 3, 6, 7, 8};
    return score;
}
```

peut être utilisé comme suit :

```
public static void main(String[] args) {
    int[] tab = scoreInitial();

    System.out.println(Arrays.toString(tab));
}
```


Tableaux d'objets

L'utilisation des tableaux n'est pas limité aux types primitifs. On peut créer des tableaux d'objets. On a déjà utilisé les tableaux d'objets dans la méthode « main(String[] args) », puisque « String » est une classe.

Considérons la classe « Etudiant » vue dans les chapitres précédents et en TP :

```
class Etudiant {
    private String nom, prenom, cne;
    public Etudiant() {
    }
    ...
}
```

Tableaux d'objets

La déclaration :

```
Etudiant[] etudiants = new Etudiant[30];
```

Crée l'emplacement pour contenir 30 objets de type « Etudiant ». Elle ne crée que les références vers les objets. Pour créer les objets eux mêmes, il faut utiliser, par exemple, l'instruction suivante :

```
for(int i=0; i<etudiants.length; i++)
    etudiants[i]=new Etudiant();
```


Attention

Pour les tableaux d'objets, les méthodes de la classe « **Arrays** » opèrent sur les références et non sur les valeurs des objets.

Exemple

```
import java.util.Arrays;

public class TableauxObjets {
    public static void main(String[] args) {
        Etudiant [] etud1 = new Etudiant[2];
        Etudiant [] etud2 = new Etudiant[2];
        Etudiant [] etud3 = new Etudiant[2];
        boolean b;
        //Initialisation de etud1
        etud1[0]= new Etudiant("Oujdi","Ali");
        etud1[1]= new Etudiant("Berkani","Lina");

        //Initialisation de etud2
        etud2[0]= new Etudiant("Mohammed","Ali");
        etud2[1]= new Etudiant("Figuigui","Fatima");
```


Exemple (suite)

```
        b = Arrays.equals(etud1, etud2);
        System.out.println(b); //affiche false

        etud2[0] = etud1[0];
        etud2[1] = etud1[1];
        b = Arrays.equals(etud1, etud2);
        System.out.println(b); //affiche true

        etud3 = etud1;
        b = Arrays.equals(etud1, etud3);
        System.out.println(b); //affiche true
    }
}
```

Objets qui contiennent des tableaux

On peut avoir un objet qui contient des tableaux.

Considérons la classe « Etudiant » et ajoutons à cette classe le tableau **notes** comme suit :

```
class Etudiant {
    private String nom, prenom, cne;
    private double [] notes = new double[6];
    ...
}
```


Tableaux à plusieurs dimensions

On peut créer des tableaux à plusieurs dimensions par ajout de crochets (`[ ]`). Par exemple, l'instruction :

double [][] matrice;

déclare un tableau à 2 dimensions de type **double**.

Comme pour le tableau à une seule dimension, la création d'un tableau multi-dimensionnelle peut se faire par utilisation de l'opérateur **new**.

Exemple

L'instruction :

matrice = new double[4][3];

crée un tableau de 4 lignes et 3 colonnes.

On peut combiner les deux instructions précédentes :

double [][] matrice = new double[4][3];

Remarques

- En langage C, un tableau à plusieurs dimensions est en réalité un tableau à une dimension. Par exemple, la déclaration :
  double matrice [4][3];
  crée en mémoire un tableau (contiguë) de 12 double.
- En Java, un tableau à plusieurs dimensions n'est pas contiguë en mémoire. En Java, un tableau de plusieurs dimensions est un tableau de tableaux.
- On peut définir un tableau à 2 dimensions dont les colonnes n'ont pas la même dimension.

Exemple 1

```
double[][] tabMulti = new double[2][];

//tabMulti[0] est un tableau de 3 doubles
tabMulti[0] = new double[3];

//tabMulti[1] est un tableau de 4 doubles
tabMulti[1] = new double[4];

System.out.println(tabMulti.length);      //Affiche 2
System.out.println(tabMulti[0].length); //Affiche 3
System.out.println(tabMulti[1].length); //Affiche 4
```


Exemple 1 (suite)

```
for(int i=0;i<tabMulti.length;i++)
    for(int j=0; j<tabMulti[i].length;j++)
        tabMulti[i][j]=i+j;


System.out.println(Arrays.deepToString(tabMulti));
//Affiche [[0.0, 1.0, 2.0], [1.0, 2.0, 3.0, 4.0]]
```

Exemple 2

Dans l'exemple suivant, on va créer un tableau triangulaire qui sera initialisé comme suit :

```
0
0    0
0    0    0
0    0    0    0
```


Exemple 2

```
final int N = 4;
int [][] tabTriangulaire = new int[N][];
for(int n=0; n<N; n++)
    tabTriangulaire[n]= new int[n+1];

System.out.println(Arrays.deepToString(
   tabTriangulaire));
//Affiche [[0], [0, 0], [0, 0, 0], [0, 0, 0, 0]]

for(int i=0;i<tabTriangulaire.length;i++){
    for(int j=0; j<tabTriangulaire[i].length;j++)
        System.out.print(tabTriangulaire[i][j]+"\t")
            ;
    System.out.println();
}
```

Parcours d'un tableau multi-dimensionnel

Comme pour le cas à une seule dimension, on peut utiliser la boucle (pour chaque -for each) pour accéder au contenu d'un tableau multi-dimensionnel.

```
double [][] matrice = new double[4][3];

for(int i=0; i<4; i++)
    for(int j=0; j<3;j++)
            matrice[i][j]=i+j;

double somme=0;
for(double [] ligne : matrice)
    for(double val: ligne)
        somme += val;

System.out.println("somme = "+somme);
```


Exemple

```
double[][] tabMulti = {new double[4],new double[5]};

System.out.println(tabMulti.length); //Affiche 2
System.out.println(tabMulti[0].length); //Affiche 4
System.out.println(tabMulti[1].length); //Affiche 5


double[][] tabMulBis = {{1,2},{3,5},{3,7,8,9,10}};

System.out.println(tabMulBis.length); //Affiche 3
System.out.println(tabMulBis[0].length); //Affiche 2
System.out.println(tabMulBis[1].length); //Affiche 2
System.out.println(tabMulBis[2].length); //Affiche 5
```

Chapitre 8
Chaînes de caractères

Introduction

Considérons l'exemple suivant :

```
import java.util.Scanner;

public class TestComparaisonChaines {
  public static void main(String[] args) {
    String nom = "smi", str;
    Scanner input = new Scanner(System.in);
    System.out.print("Saisir le nom : ");
    str = input.nextLine();
    if (nom == str)
      System.out.println(nom + " = " + str);
    else
      System.out.println(nom + " different de " +
          str);
  }
}
```

Introduction

Dans cet exemple, on veut comparer les deux chaînes de caractères `nom` initialisée avec `"smi"` et `str` qui est saisie par l'utilisateur.

Lorsque l'utilisateur exécute le programme précédent, il aura le résultat suivant :

```
Saisir le nom de la filiere : smi
smi different de smi
```

Il semble que le programme précédent produit des résultats incorrects. Le problème provient du fait, que dans java, **String** est une classe et par conséquent chaque chaîne de caractères est un objet.

Remarque :

L'utilisation de l'opérateur == implique la comparaison entre les références et non du contenu.

En java, il existe des classes qui permettent la manipulation des caractères et des chaînes de caractères :

- **Character :** une classe qui permet la manipulation des caractères (un seul caractère).
- **String :** manipule les chaînes de caractères fixes.
- **StringBuilder** et **StringBuffer** : manipulent les chaînes de caractères modifiables.

Manipulation des caractères

Nous donnons quelques méthodes de manipulation des caractères.

L'argument passé pour les différentes méthodes peut être un caractère ou son code unicode.

Majuscule

isUpperCase() : test si le caractère est majuscule

toUpperCase() : si le caractère passé en argument est une lettre minuscule, elle retourne son équivalent en majuscule. Sinon, elle retourne le caractère sans changement.

Majuscule : Exemple

```
public class TestUpper {
    public static void main(String[] args) {
        char test='a';

        if (Character.isUpperCase(test))
            System.out.println(test + " est
                majuscule");
        else
            System.out.println(test + " n'est pas
                majuscule");

        test = Character.toUpperCase(test);
        System.out.println("Apres toUpperCase() : "
            + test);
    }
}
```

Minuscule

isLowerCase() : test si le caractère est minuscule

toLowerCase() : Si le caractère passé en argument est une lettre majuscule, elle retourne son équivalent en minuscule. Sinon, elle retourne le caractère sans changement.

isDigit() : Retourne `true` si l'argument est un nombre (0−9) et `false` sinon

isLetter() : Retourne `true` si l'argument est une lettre et `false` sinon

isLetterOrDigit() : Retourne `true` si l'argument est un nombre ou une lettre et `false` sinon

isWhitespace() : Retourne `true` si l'argument est un caractère d'espacement et `false` sinon. Ceci inclue l'espace, la tabulation et le retour à la ligne

Déclaration

Comme on l'a vu dans les chapitres précédents, la déclaration d'une chaîne de caractères se fait comme suit :

String nom;

L'initialisation se fait comme suit :

nom="Oujdi";

Les deux instructions peuvent être combinées :

String nom = "Oujdi";

L'opérateur **new** peut être utilisé :

String nom = new String("Oujdi");

Pour créer une chaîne vide : String nom = new String();

ou bien : String nom = "";

Remarques

Une chaîne de type "Oujdi" est considérée par java comme un objet.
Les déclarations suivantes :

```
String nom1 = "Oujdi";
String nom2 = "Oujdi";
```

déclarent deux variables qui référencent le même objet ("Oujdi").

Par contre, les déclarations suivantes :

```
String nom1 = new String("Oujdi");
String nom2 = new String("Oujdi"); //ou nom2 = new String(nom1)
```

déclarent deux variables qui référencent deux objets différents.

Méthodes de traitement des chaînes de caractères

La compilation du programme suivant génère l'erreur « `array required, but String found System.out.println("Oujdi"[i]);` »

```
public class ProblemeManipString {
    public static void main(String[] args) {
        for(int i=0; i<5; i++)
            System.out.println("Oujdi"[i]);
    }
}
```

Pour éviter les erreurs de ce type, la classe « String » contient des méthodes pour manipuler les chaînes de caractères. Dans ce qui suit, nous donnons quelques unes de ces méthodes.

Méthode **charAt()**

Retourne un caractère de la chaîne.

Une correction de l'exemple précédent est :

```
public class ProblemeManipStringCorrection {
    public static void main(String[] args) {
        for(int i=0; i<5; i++)
            System.out.println("Oujdi".charAt(i));
    }
}
```

Méthode **concat()**

Permet de concaténer une chaîne avec une autre.

```
nom = "Oujdi".concat(" Mohammed");
//nom<   "Oujdi Mohammed"
```


Méthode **trim()**

supprime les séparateurs de début et de fin (espace, tabulation, ...)

```
nom = "\n Oujdi"+" Mohammed \n\t";
nom = nom.trim(); //nom<--"Oujdi Mohammed"
```

Méthodes replace() et replaceAll()

- **replace()** : Remplace toutes les occurrences d'une chaîne de caractères avec une autre chaîne de caractères
- **replaceAll()** : Remplace toutes les occurrences d'une expression régulière par une chaîne

```
String str;
str = "Bonjour".replace( "jour", "soir" );
//str <--"Bonsoir"
str = "soir".replaceAll("[so]", "t");
//str <-- "ttir"
str = "def".replaceAll("[a-z]","A");
//str <-- "AAA"
```


Méthode compareTo()

Permet de comparer un chaîne avec une autre chaîne.

```
String abc = "abc";
String def = "def";
String num = "123";
if ( abc.compareTo( def ) < 0 )    // true
  if ( abc.compareTo( abc ) == 0 )  // true
     if ( abc.compareTo( num ) > 0 )    // true
         System.out.println(abc);
```

Méthode indexOf()

Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 's' );  // 18
int j = abcs.indexOf( "def" ); // 3
int k = abcs.indexOf( "smi" );  // -1
```


Méthode valueOf()

Retourne la chaîne qui est une représentation d'une valeur

```
double x=10.2;
str = String.valueOf(x); //str <-- "10.2"
int i = 20;
str = String.valueOf(i); //str <-- "20"
```

Méthode

Retourne une sous-chaîne de la chaîne :

- la sous-chaîne commence à partir du caractère qui se trouve à l'indice et se termine à la fin de la chaîne ;
- la sous-chaîne commence à partir du caractère qui se trouve à l'indice et se termine au caractère qui se trouve à l'indice .

```
str = "Bonjour".substring(3); //str <-- "jour"
str = "toujours".substring(3,7); //str <-- "jour"
```


Méthodes **startsWith()** et **endsWith()**

- **startsWith()** : Vérifie si une chaîne commence par un suffixe
- **endsWith()** : Vérifie si une chaîne se termine par un suffixe

```
String url = "http://www.ump.ma";
if ( url.startsWith("http") ) // true
  if ( url.endsWith("ma") ) // true
      System.out.println(str);
```

Méthodes equals() et equalsIgnoreCase()

La méthode :

- **equals()** compare une chaîne avec une autre chaîne en tenant compte de la casse (majuscule ou minuscule) ;
- **equalsIgnoreCase()** compare une chaîne avec une autre chaîne en ignorant la casse.

Exemple :

```
String str1="test", str2="Test";
if(str1.equals(str2))
   System.out.println("Les 2 chaines sont egaux");
else
   System.out.println("Les 2 chaines differents");

if(str1.equalsIgnoreCase(str2))
   System.out.println("Les 2 chaines sont egaux (
      sans tenir compte de la casse)");
```


Méthode getBytes()

Copie les caractères d'une chaîne dans un tableau de bytes.

Exemple :

```
String str1="abc_test_123";
byte[] b;
b = str1.getBytes();
for (int i = 0; i < b.length; i++)
  System.out.println(b[i]);
```

Méthode getChars()

Copie les caractères d'une chaîne dans un tableau de caractères.

Utilisation :
Soit **str** une chaîne de caractères. La méthode **getChars()** s'utilise comme suit :

str .getChars(srcDebut, srcFin, dst, dstDebut)

Paramètres :

- **srcDebut :** indice du premier caractère à copier ;
- **srcFin :** indice après le dernier caractère à copier ;
- **dst :** tableau de destination ;
- **dstDebut :** indice du premier élément du tableau de destination.

Méthode getChars() : **Exemple :**

```
str = "Ceci est un test";
char [] c = new char[str.length()];

str.getChars(0, str.length(), c, 0);
for (int i = 0; i < c.length; i++)
    System.out.println(c[i]);
```

Méthode toCharArray()

Met la chaîne dans un tableau de caractères.

Exemple :

```
str="Test";
char[] tabC=str.toCharArray();
for (int ind = 0; ind < tabC.length; ind++)
    System.out.println(tabC[ind]);
```


Méthode isEmpty()

Retourne **true** si la chaîne est de taille nulle.

Exemple :

```
if(str.isEmpty())
    System.out.println("Chaine vide");
else
    System.out.println("Chaine non vide");
```

Méthode indexOf()

Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne. Elle retourne l'indice du premier occurrence du caractère dans la chaîne ou **-1** si le caractère n'existe pas.

Exemple :

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 's' );   // 18
int j = abcs.indexOf( "def" ); // 3
int k = abcs.indexOf( "smi" );  // -1
```


Méthode lastIndexOf()

Cherche la dernière occurrence d'un caractère ou d'une sous-chaîne dans une chaîne. Elle retourne l'indice du dernier occurrence du caractère dans la chaîne ou **-1** si le caractère n'existe pas.

Exemple :

```
String abcs = "abcdefghijklmnopqrstuvwxyz+str";
int i = abcs.lastIndexOf( 's' );   // 27
int j = abcs.lastIndexOf( "def" ); // 3
int k = abcs.lastIndexOf( "smi" );  // -1
```

Méthode replaceFirst()

Remplace la première occurrence d'une expression régulière par une chaîne.

Exemple :

```
str = "Test1test2".replaceFirst("[0-9]", "_");
//Test_test2
```


MéthodestoLowerCase()et toUpperCase()

La méthode :

- **toLowerCase() convertie la chane en minuscule**
- **toUpperCase() convertie la chane en majuscule.**

Exemple :

```
String nom      ujdi
nom   nom.toUpperCase()   nom --    UJDI

str    TEst
str   str.toLowerCase()   str --   test
```

Méthode split()

split() sépare la chane en un tableau de chanes en utilisant une expression régulière comme délimiteur.

```
nom       ujdi Mohammed
String[] tabStr   nom.split( )
for (int ind   0  ind   tabStr.length  ind  )
  System.out.println(tabStr[ind])   ffichera :
    ujdi
  Mohammed
str  Un:deux trois quatre
tabStr   str.split([:  ])
for (int ind   0  ind   tabStr.length  ind  )
    System.out.println(tabStr[ind])    ffichera :
  Un
  deux
  trois
  quatre
```


Méthode hashCode()

Retourne le code de hachage (hashcode) d'une chane. Pour une chane s le code est calculé de la fa on suivante :

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-1]$$

avec : $s[i]$ est le code du caractère la position i (voir la méthode getBytes()).

Exemple :

```
str     12
System.out.println(code      str.hashCode())
  retourne : 156      *31 50
```

Méthode valueOf()

Retourne la chaîne qui est une représentation d'une valeur.

Exemple :

```
double x = 10.2;
str = String.valueOf(x); //str <-- "10.2"

int test = 20;
str = String.valueOf(test); //str <-- "20"
```


Conversion entre String et types primitifs

Comme on l'a vu précédemment, la méthode **`valueOf()`** de la classe **`String`** permet de récupérer la valeur d'un type primitif.

Les classes **Byte, Short, Integer, Long, Float,** et **Double**, disposent de la méthode statique **toString()** qui permet de convertir un type primitif vers un **String**. Elles disposent respectivement des méthodes statiques **parseByte(), parseShort(), parseInt(), parseLong(), parseFloat()** et **parseDouble()**, qui permettent de convertir une chaîne en un type primitif.

Exemple :

```
public class TestConversion{
  public static void main(String[] args){
    double x = 4.5;
    String str = Double.toString(x);//str<--"4.5"

    int test = 20;
    str = Integer.toHexString(test); //str <-- "20"

    str = "2.3";
    x = Double.parseDouble(str); //x <-- 2.3

    str = "5";
    test = Integer.parseInt(str); //test <-- 5
  }
}
```


Affichage des objets

Le code suivant :

```
Etudiant etud = new Etudiant("Oujdi","Ali","A20");
System.out.println(etud);
```

affichera quelque chose comme : `Etudiant@1b7c680`.

Cette valeur correspond à la référence de l'objet.

Affichage des objets

Si on souhaite afficher le contenu de l'objet en utilisant le même code, il faut utiliser la méthode **toString** prévu par Java.

Affichage des objets : Exemple

```
public class TestEtudiantToString {
  public static void main(String[] args) {
    Etudiant et = new Etudiant("Oujdi","Ali","A20");
    System.out.println(etud);
  }
}

class Etudiant {
  private String nom, prenom, cne;
  ...
  public String toString(){
    return "Nom : "+nom+"\nPrenom : "+prenom+"\nCNE
        : "+cne;
  }
  ...
}
```

Affichage des objets : Exemple

Affichera :

```
Nom : Oujdi
Prenom : Ali
CNE : A20
```


Comparaison des objets

Le code suivant :

```
Etudiant etud1 = new Etudiant("Oujdi","Ali","A20");
Etudiant etud2 = new Etudiant("Oujdi","Ali","A20");

if(etud1 == etud2)
    System.out.println("Identiques");
else
    System.out.println("Differents");
```

affichera toujours « `Differents` », du fait que la comparaison s'est faite entre les références des objets.

Comparaison des objets

Comme pour la méthode **toString**, Java prévoit l'utilisation de la méthode **equals** qui pourra être définie comme suit :

```
class Etudiant {
  private String nom, prenom, cne;
  ...
  public boolean equals(Etudiant bis){
    if (nom == bis.nom && prenom == bis.prenom  &&
       cne == bis.cne)
       return true;
    else
       return false;
  }
}
```


Comparaison des objets : utilisation de equals

```
public class TestEtudiantToString {
  public static void main(String[] args) {
  Etudiant et = new Etudiant("Oujdi","Ali","A20");
  Etudiant et1 = new Etudiant("Oujdi","Ali","A20");

  if(et.equals(et1))
    System.out.println("Identiques");
  else
    System.out.println("Differents");
  }
}
```

Les classes StringBuilder et StringBuffer

Lorsqu'on a dans un programme l'instruction

```
str = "Bonjour";
```

suivie de

```
str = "Bonsoir";
```

le système garde en mémoire la chaîne "Bonjour" et crée une nouvelle place mémoire pour la chaîne "Bonsoir". Si on veut modifier "Bonsoir" par "Bonsoir SMI5", alors l'espace et "SMI5" ne sont pas ajoutés à "Bonsoir" mais il y aura création d'une nouvelle chaîne. Si on fait plusieurs opérations sur la chaîne str, on finira par créer plusieurs objets dans le système, ce qui entraîne la consommation de la mémoire inutilement.

Les classes StringBuilder et StringBuffer

Pour remédier à ce problème, on peut utiliser les classes **StringBuilder** ou **StringBuffer**. Les deux classes sont identiques à l'exception de :

- **StringBuilder** : est plus efficace.
- **StringBuffer** : est meilleur lorsque le programme utilise les threads.

Puisque tous les programmes qu'on va voir n'utilisent pas les threads, le reste de la section sera consacré à la classe **StringBuilder**.

Déclaration et création

Pour créer une chaîne qui contient "Bonjour", on utilisera l'instruction :

```
StringBuilder  message = new StringBuilder("Bonjour");
```

Pour créer une chaîne vide, on utilisera l'instruction :

```
StringBuilder  message = new StringBuilder();
```


Remarque :

L'instruction :

```
StringBuilder  message = "Bonjour";
```

est incorrecte. Idem, si « message » est un StringBuilder, alors l'instruction :

```
message = "Bonjour";
```

est elle aussi incorrecte.

Méthodes de StringBuilder

length()	Retourne la taille de la chaîne
charAt()	Retourne un caractère de la chaîne
substring()	Retourne une sous-chaîne de la chaîne
setCharAt(i,c)	permet de remplacer le caractère de rang i par le caractère c.
insert(i,ch)	permet d'insérer la chaîne de caractères ch à partir du rang i
append(ch)	permet de rajouter la chaine de caractères ch à la fin
deleteCharAt(i)	efface le caractère de rang i.
toString()	Convertie la valeur de l'objet en une chaîne (conversion de StringBuilder vers String)
concat()	Concaténer une chaîne avec une autre
contains()	Vérifie si une chaîne contient une autre chaîne

endsWith()	Vérifie si une chaîne se termine par un suffixe
equals()	Compare une chaîne avec une autre chaîne
getBytes()	Copie les caractères d'une chaîne dans un tableau de bytes
getChars()	Copie les caractères d'une chaîne dans un tableau de caractères
hashCode()	Retourne le code de hachage (hashcode) d'une chaîne
indexOf()	Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne
lastIndexOf()	Cherche la dernière occurrence d'un caractère ou d'une sous-chaîne dans une chaîne
replace()	Remplace toutes les occurrences d'un caractère avec un autre caractère

Remarque :

On ne peut pas faire la concaténation avec l'opérateur + entre des StringBuilder. Par contre :

StringBuilder + String

produit une nouvelle chaîne de type String.

Exemple

```
public class TestStringBuilder {
  public static void main(String args[]) {
  StringBuilder strBuilder = new StringBuilder("
     Bonjour SMI5");
  int n = strBuilder.length(); // n <-- 12

  char c = strBuilder.charAt(2); // c <-- 'n'

  strBuilder.setCharAt(10, 'A');
  // remplace dans strBuilder le caractere 'I' par '
     A'.
  // strBuilder <-- "Bonjour SMA5"
```

Exemple (suite)

```
  strBuilder.insert(10, " semestre ");
  // insere dans strBuilder la chaine " semestre "
     a
  // partir du rang 10.
  // strBuilder <   "Bonjour SMA semestre 5"


  strBuilder.append(" (promo 14 15)");
  // strBuilder <   "Bonjour SM semestre A5 (promo
     14 15)"

  strBuilder = new StringBuilder("Boonjour");
  strBuilder.deleteCharAt(2);
  // supprime de la chaine strBuilder le caractere
     de rang 2.
  // strBuilder <   "Bonjour"
```


Exemple (suite)

```
  String str = strBuilder.toString();
  // str <-- "Bonjour"

  str = strBuilder.substring(1, 4);
  // str <-- "onj"

  str = strBuilder + " tous le monde";
  // str <-- "Bonjour tous le monde"
  }
}
```

Exercices

Exercice

Écrivez une application qui compte le nombre d'espaces contenus dans une chaîne de caractères saisie par l'utilisateur. Sauvegardez le fichier sous le nom **NombreEspaces.java**.

Solution

```
import java.util.Scanner;

public class NombreEspaces {
  public static void main(String[] args) {
    int nombreEspaces = 0;
    String str;

    Scanner clavier = new Scanner(System.in);
    System.out.println("Saisir une chaine ");
    str = clavier.nextLine();

    for (int i = 0; i < str.length(); i++)
      if (str.charAt(i) == ' ')
        nombreEspaces++;
    System.out.println("Nombre d'espaces : " +
       nombreEspaces);
```


Nombre de caractères d'espacement :

```
    nombreEspaces = 0;
    for (int i = 0; i < str.length(); i++)
      if (Character.isWhitespace(str.charAt(i)))
        nombreEspaces++;

    System.out.println("Nombre de caracteres d'
       espacement : " + nombreEspaces);
    clavier.close();
  }
}
```

Exercice

L'utilisation de trois lettres dans les acronymes est courante. Par exemple :

- JDK : Java Development Kit ;
- JVM :Java Virtual Machine ;
- RAM : Random Access Memory ;
- ...

Écrivez un programme qui demande à l'utilisateur de saisir trois mots et affiche à à l'écran l'acronyme correspondant composé des trois premières lettres en majuscule. Si l'utilisateur saisisse plus de trois mots, le reste sera ignoré.

Solution

```
import java.util.Scanner;

public class Acronymes {
   public static void main(String[] args) {
      String str, acronyme = "";
      String[] tabStr;

      Scanner clavier = new Scanner(System.in);
      do {// pour forcer l'utilisateur a saisir plus
          de 3 mots
         System.out.println("Saisir une chaine
            composee de plus de trois");
         str = clavier.nextLine();
         tabStr = str.split(" ");
      } while (tabStr.length < 3);
```

```
        for (int i = 0; i < 3; i++)
            acronyme += tabStr[i].charAt(0);

        System.out.println("Acronyme : " + acronyme.
            toUpperCase());
        clavier.close(); //fermer le clavier
    }
}
```

Chapitre 9
Interfaces et packages

Introduction

Le langage c++ permet l'héritage multiple, par contre Java ne permet pas l'héritage multiple. Pour remédier à ceci, Java utilise une alternative qui est la notion **d'interfaces**.

Définition

une **interface** est un ensemble de méthodes abstraites.

Déclaration

La déclaration d'une interface se fait comme celle d'une classe sauf qu'il faut remplacer le mot clé class par interface.

Exemple

```
public interface Forme {
    public abstract void dessiner ( ) ;
    public abstract void deplacer (int x , int y) ;
}
```

Propriétés

Les interfaces ont les propriétés suivantes :

- une interface est implicitement abstraite. On n'a pas besoin d'utiliser le mot clé abstract dans sa déclaration ;
- chaque méthode définie dans une interface est abstraite et public, par conséquent, les mots clés public et abstract peuvent être omis.

Propriétés

L'exemple précédent devient :

```
interface Forme {
    void dessiner ( ) ;
    void deplacer (int x , int y) ;
}
```

Règles

Une interface est similaire à une classe dans les points suivants :

- une interface peut contenir plusieurs méthodes ;
- une interface peut se trouver dans un fichier séparé (.java) ;
- peut se trouver dans un paquetage (voir section packages).

Restrictions

Il y a des restrictions concernant les interfaces. Une interface :

- ne peut pas instancier un objet et par conséquent ne peut pas contenir de constructeurs ;
- ne peut pas contenir d'attributs d'instances. Tous les attributs doivent être static et final ;
- peut hériter de plusieurs interfaces (héritage multiple autorisé pour les interfaces).

Remarque :

Dans Java 8, une interface peut contenir des méthodes statiques.

Implémenter une interface

Une classe peut implémenter une interface en utilisant le mot clé implements. On parle d'implémentation et non d'héritage.

Exemple

```
class Rectangle implements Forme {
    private double largeur, longueur;
    ...

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

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


Implémentation partielle

Une classe doit implémenter toutes les méthodes de l'interface, sinon elle doit être abstraite.

Exemple

```
abstract class Rectangle implements Forme
    {
    private double largeur, longueur;

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

Implémentation multiple

Une classe peut implémenter plusieurs interfaces.

Exemple

Considérons les 2 interfaces **I1** et **I2** :

```
interface I1 {
    final static int MAX = 20;
    void meth1();
}
```

```
interface I2 {
    void meth2();
    void meth3();
}
```

Exemple

La classe **A** peut implémenter les 2 interfaces **I1** et **I2** :

```
class A implements I1 , I2 {
    // attributs
    public void meth1 () {
        int i=10;
        //on peut utiliser la constante MAX
        if (i < MAX) i++;
        //implementation de la methode
    }
    public void meth2 () {
        // ...
    }
    public void meth3 () {
        // ...
    }
}
```


Implémentation et héritage

Une classe **B** peut hériter de la classe **A** et implémenter les 2 interfaces **I1** et **I2**, comme suit :

```
class B extends A implements I1,I2
```

Polymorphisme et interfaces

Déclaration :

On peut déclarer des variables de type interface :
Forme forme;

Pour instancier la variable « forme », il faut utiliser une classe qui implémente l'interface « Forme ». Par exemple :

```
forme = new Rectangle(2.5, 4.6);
```


Tableaux

Ceci peut être étendu pour les tableaux. Considérons la classe **Cercle** qui implémente elle aussi l'interface **Forme** et la classe **Carre** qui hérite de **Rectangle**.

Tableaux

Classe Cercle :

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

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

    public double perimetre() {
        return 2 * Math.PI * rayon;
    }

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


Tableaux

Classe Carre :

```
class Carre extends Rectangle{
    private double largeur;

    public Carre(double largeur){
        super(largeur,largeur);
    }
}
```

Tableaux

On pourra écrire :

```
public static void main(String[] args) {
    Forme [] tabForme = new Forme[3];
    tabForme[0]=new Rectangle(10, 20);
    tabForme[1]=new Cercle(3);
    tabForme[2]=new Carre(10);

    for(int i=0; i<3; i++)
        System.out.printf("%.2f\t%.2f\n",
            tabForme[i].surface(),
            tabForme[i].perimetre());
}
```


Casting

Ajoutons à la classe **Cercle** la méthode « diametre() » :

```
class Cercle implements Forme {
    private double rayon;
    ...
    public double diametre() {
        return 2 * rayon;
    }
}
```

Casting

Considérons l'instruction :

Forme forme = new Cercle(5);

L'instruction :

double d = forme.diametre();

génère une erreur de compilation. Pour éviter cette erreur, il faut faire un cast comme suit :

double d = ((Cercle) forme).diametre();

Héritage

Une interface peut hériter d'une ou plusieurs interfaces.

Exemple :

Considérons les deux interfaces **I1** et **I2**

```
interface I1 {
    final static int MAX = 20;
    void meth1();
}
```

```
interface I2 {
    void meth2();
    void meth3();
}
```

Héritage

Une interface **I3** pourra hériter des deux interfaces **I1** et **I2**

```
interface I3 extends I1, I2 {
    void meth4();
}
```

On fait l'instruction interface I3 extends I1, I2, est équivalente à :

```
interface I3  {
    final static int MAX = 20;
    void meth1();
    void meth2();
    void meth3();
    void meth4();
}
```


Héritage

Important

L'héritage entre interfaces est différents de celui des classes.
L'héritage multiple est autorisé pour les interfaces et n'est pas autorisés pour les classes

Exercices

Exercice

Corrigez le programme suivant et justifiez chaque correction :

```
interface  Test {
    double m1();
    int m2(){}
}

public class ClasseA {
    public static void main(String[] args
        ) {
         Test i1 = new Test();
         Test i2;
    }
}
```

Solution

```
interface  Test {
    double m1();
    //methode abstraite, ne doit pas contenir un
       corps
    int m2();

}


public class ClasseA {
    public static void main(String[] args) {
        //On ne peut pas instancier une interface
        Test i1 ;
        Test i2;
    }
}
```


Exercice

Créez une interface nommée « **Tourner** » qui contient une seule méthode nommée « **tourner** ».

Créez une classe nommée :

- « **Page** » qui implémente **tourner** qui affiche « Tourner la page » ;
- « **Disque** » qui implémente **tourner** qui affiche « Faire une rotation ».
- « **Voiture** » qui implémente **tourner** qui affiche « Faire un tour ».

Utilisez la classe « **TestTourner** » pour tester les trois classes

Solution

L'interface **Tourner :**

```
public interface Tourner {
    void tourner();
}
```

Classe **Page :**

```
public class Page implements Tourner {
    public void tourner() {
        System.out.println("Tourner la page");
    }
}
```


Classe **Disque :**

```
public class Disque implements Tourner {
    public void tourner() {
      System.out.println("Faire une rotation");
   }
}
```

Classe **Voiture :**

```
public class Voiture implements Tourner {
   public void tourner() {
      System.out.println("Faire un tour");
   }
}
```

Classe **TestTourner :**

```
public class TestTourner {
    public static void main(String[] args) {
        Page page = new Page();
        Disque disque = new Disque();

        page.tourner();
        disque.tourner();

        Tourner[] tab = new Tourner[3];
        tab[0] = new Page();
        tab[1] = new Disque();
        tab[2] = new Voiture();

        for (Tourner tr : tab)
            tr.tourner();
    }
}
```


Packages

Introduction

Un package est un ensembles de classes. Il sert à mieux organiser les programmes. Si on n'utilise pas de packages dans nos programmes, alors on travail automatiquement dans le package par défaut (default package).

Pour la saisie à partir du clavier, nous nous avons utilisé la **Scanner**, pour cela nous avons importé le package **java.util**.

Création d'un package

Pour créer un package, on utilise le mot clé package. Il faut ajouter l'instruction

package nomPackage;

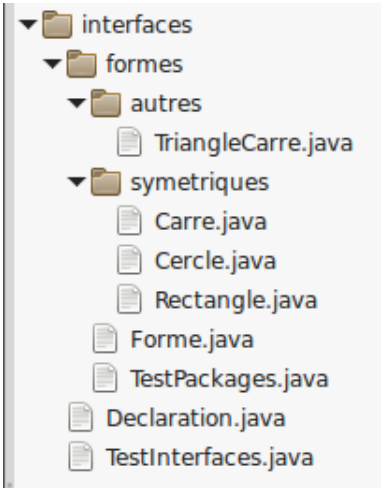
au début de chaque fichier.

Pour qu'elle soit utilisable dans un package, une classe publique doit être déclarée dans un fichier séparé.

On accède à une classe publique en utilisant le nom du package.

Exemple

Considérons la hiérarchie suivante :



Exemple

Nous avons la structure suivante :

- le répertoire **interfaces** contient le répertoire **formes** et les fichiers **Declaration.java** et **TestInterfaces.java** ;
- le répertoire **formes** contient les répertoires **symetriques** et **autres**, et les fichiers **Forme.java** et **TestPackages.java** ;
- le répertoire **autres** contient le fichier **TriangleCarre.java** ;
- le répertoire **symetriques** contient les fichiers **Carre.java**, **Cercle.java** et **Rectangle.java**.

Exemple

Nous avons la classe **TriangleCarre** se trouve dans le répertoire **autres**, donc on doit inclure, au début l'instruction :

package interfaces.formes.autres;

Nous avons aussi la classe **TriangleCarre** hérite de la classe **Rectangle**, donc, on doit inclure l'instruction :

import interfaces.formes.symetriques.Rectangle;

La classe **TriangleCarre** aura la structure suivante :

```
package interfaces.formes.autres;

import interfaces.formes.symetriques.Rectangle;

public class TriangleCarre extends Rectangle {
    private double cote;

    public TriangleCarre(double largeur, double
       longueur, double cote) {
        super(largeur, longueur);
        this.cote = cote;
    }

    public double surface() {
        return super.surface() / 2;
    }
}
```

La classe **TestPackages** sert pour tester les différentes classes, donc on doit inclure, au début les instructions :

```
package interfaces.formes.autres;

import interfaces.formes.autres.TriangleCarre;
import interfaces.formes.symetriques.Carre;
import interfaces.formes.symetriques.Cercle;
import interfaces.formes.symetriques.Rectangle;
```


Pour simplifier, on peut remplacer les trois dernières instructions par :

```
import interfaces.formes.symetriques.*;
```

La structure de la classe **TestPackages** est comme suit :

```
package interfaces.formes;
import interfaces.formes.autres.TriangleCarre;
import interfaces.formes.symetriques.*;

public class TestPackages {
    public static void main(String[] args) {
        Forme[] tabForme = new Forme[4];
        tabForme[0] = new Rectangle(10, 20);
        tabForme[1] = new Cercle(3);
        tabForme[2] = new Carre(10);
        tabForme[3] = new TriangleCarre(3,4,5);

        for (int i = 0; i < 3; i++)
            System.out.printf("%.2f\t%.2f\n",
                tabForme[i].surface(),
                    tabForme[i].perimetre());
    }
}
```


Classes du même package

Pour les classes qui sont dans le même package, on n'a pas besoin de faire des importations et on n'a pas besoin de mettre une classe par fichier si on veut que cette classe ne soit pas visible pour les autres packages.

Classes du même package

Soit le fichier **Carre.java** qui contient les 2 classes **Carre** et **Cube** :

```
package interfaces.formes.symetriques;

public class Carre extends Rectangle {
    private double largeur;

    public Carre(double largeur) {
        super(largeur, largeur);
    }
    ...
}
class Cube extends Carre {
    ...
    public double volume() {
        return surface() * super.getLargeur();
    }
```


Remarques

1. Les classes **Rectangle** et **Carre** sont dans le même package, donc on n'a pas besoin de faire importations.

2. La classe **Cube** est invisible dans les autres packages. Dans la classe **TestPackages**, une instruction comme :
   Cube cube =new Cube();
   génère une erreur de compilation, du fait que la classe **TestPackages** se trouve dans le package **interfaces.formes**

Fichiers jar

Un fichier **jar** (**J**ava **AR**chive) est un fichier qui contient les différentes classes (compilées) sous format compressé.

Pour générer le fichier **jar** d'un projet sous eclipse, il faut

- cliquer avec la souris sur le bouton droit sur le nom du projet puis cliquer sur **export**
- choisir après **JAR** dans la section **java**,
- cliquer sur **next**, et choisir un nom pour le projet dans la partie **JAR file** (par exemple **test.jar**)
- puis cliquer 2 fois sur **next**
- enfin, choisir la classe principale (qui contient **main**) et cliquer sur **Finish** pou terminer l'exportation.

Exécution

En se positionnant dans le répertoire qui contient le fichier **test.jar**, ce dernier peut être exécuté en utilisant la commande :

```
java -jar test.jar
```

Utilisation dans un autre projet

Vous pouvez utiliser le fichier jar dans un autre projet, en procédant, sous eclipse, comme suit :

- cliquer avec la souris sur le bouton droit sur le nom du projet puis cliquer sur **properties**
- choisir après **Java Build Path**
- cliquer ensuite sur l'onglet **Librairies**
- puis cliquer sur **Add External JARs**
- enfin, choisir le fichier **JAR** et valider par **ok**.

Vous pouvez ensuite importé les packages et utilisé les classes qui se trouvent dans le fichier **jar**.

Chapitre 10
Gestion des exceptions

Introduction

Définition

Une **exception** est une erreur qui se produit durant l'exécution d'un programme.

Les programmes peuvent générer plusieurs types d'exceptions :

- division par zéro ;
- une mauvaise valeur entré par l'utilisateur (une chaîne de caractères au lieu d'un entier) ;
- dépassement des capacités d'un tableau ;
- lecture ou écriture dans un fichier qui n'existe pas, ou pour lequel le programme n'a pas les droits d'accès ;
- ...

Ces erreurs sont appelés **exceptions** du fait qu'elles sont exceptionnelles.

Gestion des erreurs

Supposons qu'on veut écrire une fonction qui calcule la fonction $f(x) = \sqrt{x-1}$.

Une première version pourrait s'écrire comme suit :

```
public static double f(double x) {
        return Math.sqrt(x-1);
    }
```


Gestion des erreurs

Dans une méthode main(), l'instruction :

System.out.println("f(" + x + ") = " + f(x));

avec x = 0, on aura l'affichage :

f(0.0) = NaN

Gestion des erreurs

On pourra modifier le code de la fonction de la façon suivante :

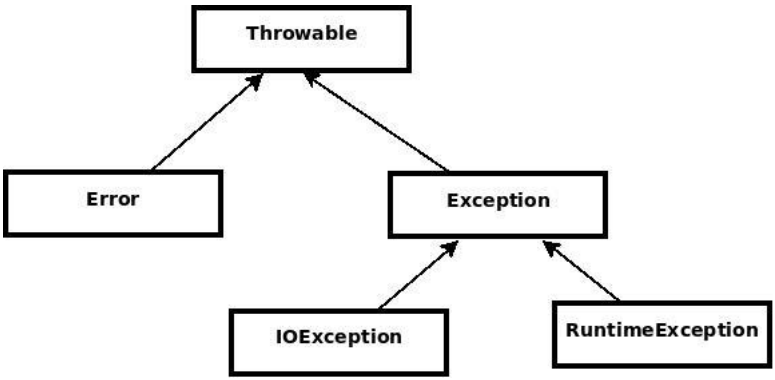
```
public static double f(double x) {
    if (x >= 1)
        return Math.sqrt(x - 1);
    else {
        System.out.println("Erreur");
        return -1;
    }
}
```

Le problème avec ce code c'est qu'il fait un affichage, qui n'est pas prévu par la fonction, de plus, il retourne une valeur qui n'est pas vraie qui pourrait aboutir à de mauvaises conséquences pour le reste du programme.

Le code précédent peut être généralisé en utilisant les exceptions.

En java, il y a deux classes pour gérer les erreurs : **Error** et **Exception**.

Les deux classes sont des sous-classe de la classe **Throwable** comme le montre la figure suivante :



Classe Error

La classe **Error** représente les erreurs graves qu'on ne peut pas gérer. Par exemple, il n'y a pas assez de mémoire pour exécuter un programme.

Le programme suivant :

```
package chap10exceptions;

//permet de gerer des tableaux dynamiques(vecteurs)
import java.util.Vector;

public class Erreurs {
    public static void main(String[] args) {
        Vector<Double> liste = new Vector<Double>();
        boolean b = true;
        while (b) {
            liste.add(2.0);
        }}}
```


Classe Error

génère l'erreur suivante :

```
Exception in thread "main" java.lang.OutOfMemoryError:
Java heap space
at java.util.Arrays.copyOf(Arrays.java:3210)
at java.util.Arrays.copyOf(Arrays.java:3181)
at java.util.Vector.grow(Vector.java:266)
at java.util.Vector.ensureCapacityHelper(Vector.java:246)
at java.util.Vector.add(Vector.java:782)
at chap10.ClasseError.main(ClasseError.java:10)
```

Classe Exception

La classe **Exception** représente les erreurs les moins graves qu'on peut gérer dans les programmes.

Le programme suivant :

```
import java.util.Scanner;
public class SaisieEntier {
    public static void main(String[] args) {
        int n;
        Scanner clavier = new Scanner(System.in);
        System.out.print("Saisir un entier : ");
        n=clavier.nextInt();
        System.out.println("1/" + n + " = " + 1/n);
        clavier.close();
    }
}
```


Classe Exception

génère l'erreur suivante, si n=0 :

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at chap10.SaisieEntier.main(SaisieEntier.java:11)
```

Si on saisi un caractère au lieu d'un entier, le programme précédent génère l'erreur suivante :

```
Saisir un entier : a
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:864)
at java.util.Scanner.next(Scanner.java:1485)
at java.util.Scanner.nextInt(Scanner.java:2117)
at java.util.Scanner.nextInt(Scanner.java:2076)
at chap10.SaisieEntier.main(SaisieEntier.java:10)
```

Types d'exceptions

Dans ce qui suit, nous donnons quelques types d'exceptions :

ArithmeticException

Erreur arithmétique, comme une division par zéro.

ArrayIndexOutOfBoundsException

Indice d'un tableau qui dépasse les limites du tableau. Par exemple :
double [] tab = new double[10];
tab[10]=1.0; ou bien tab[−1]=1.0;

NegativeArraySizeException

Tableau créé avec une taille négative. Par exemple :
double [] tab = new double[−12];

Types d'exceptions

ClassCastException : cast invalide. Par exemple :

```
Object [] T = new Object[2];

T[0] = 1.0;
T[1] = 2;

//Trie d'un tableau qui contient des entiers et des
   reelles
Arrays.sort(T);
```

NumberFormatException

Mauvaise conversion d'une chaîne de caractères vers un type numérique. Par exemple :
String s = "tt";
x = Double.parseDouble(s);

Types d'exceptions

StringIndexOutOfBoundsException

Indice qui dépasse les limites d'une chaîne de caractères. Par exemple :

```
String s = "tt";
char c = s.charAt(2);
```

ou bien

```
char c = s.charAt(-1);
```

NullPointerException

Mauvaise utilisation d'une référence. Par exemple utilisation d'un tableau d'objets créé mais non initialisé :

```
ClasseA [] a = new ClasseA[2]; // ClasseA une classe
a[0].x = 2;   // l'attribut x est public
```


Méthodes des exceptions

Dans ce qui suit, une liste de méthodes disponible dans la classe **Throwable** :

Méthode	**Description**
public String getMessage()	Retourne un message concernant l'exception produite.
public String toString()	Retourne le nom de la classe concaténé avec le résultat de « getMessage() »
public void printStackTrace()	Affiche le résultat de « toString() » avec la trace de l'erreur .

Capture des exceptions

Pour les différents tests, le programme s'est arrêté de façon brutale. Il est possible de capturer (to catch) ces exceptions et continuer l'exécution du programme en utilisant les 5 mots clés **try**, **catch**, **throw**, **throws** et **finally**.

Forme générale d'un bloc **try**

```
try {
    //Code susceptible de generer une erreur
} catch (TypeException1 excepObj) {
    //traitement en cas d'exception de type TypeException1
} catch (TypeException2 excepObj) {
    //traitement en cas d'exception de type TypeException2
//  ...
finally{
//code a executer avant la fin du bloc try
}
//code après try
```

TypeException est le type d'exception généré. **excepObj** est un objet.

Remarque :

TypeException peut-être une classe prédéfinie de Java ou une classe d'exception créée par l'utilisateur.

Un seul catch

Lors de l'exécution du programme :

```
public class Divise ero {
    public static void main(String[] args) {
        int n = 0;
        System.out.println("1/" + n + " = " + 1/n);
    }
}
```

on a obtenu l'exception **ArithmeticException**.

Pour capturer cette exception, on pourra modifier le programme en utilisant le bloc **try** de la façon suivante :

```
public class DiviseZero {
  public static void main(String[] args) {
     int n = 0;
     try {
         System.out.println("1/" +n+ " = " + 1/n);
         System.out.println("Ne sera pas affiche");
     } catch (ArithmeticException ex) {
         System.out.println("Division par zero");
     }
     System.out.println("Reste du programme");
  }
}
```


plusieurs catch

Reprenons le programme :

```
import java.util.Scanner;
public class Exceptions {
    public static void main(String[] args) {
        int n;
        Scanner clavier = new Scanner(System.in);
        System.out.print("Saisir un entier : ");
        n=clavier.nextInt();
        System.out.println("1/" + n + " = " + 1/n);
        System.out.println("Fin du programme");
        clavier.close();
    }
}
```

Traitons les différentes exceptions générées par le programme et améliorons ce dernier pour forcer l'utilisateur à saisir un entier :

```
//Pour pouvoir utiliser "InputMismatchException"
import java.util.InputMismatchException;
import java.util.Scanner;

public class Exceptions {
public static void main(String[] args) {
  int n;
  Scanner clavier = new Scanner(System.in);
  boolean b = true;
  while (b) {
    try {
      System.out.print("Saisir un entier : ");
      n = clavier.nextInt();
      System.out.println("1/" + n + " = " + 1 / n);
      break;
    }
```


Suite du programme

```
 catch (ArithmeticException e) {
   System.out.println("Division impossible par 0");
   System.out.println(e.getMessage());
 } catch (InputMismatchException e) {
   System.out.println("Vous n'avez pas saisi un
       entier");
   System.out.println(e);
   //equivalent a System.out.println(e.toString())
   //pour recuperer le retour a la ligne
   clavier.nextLine();
 }
}//Fin de while
 System.out.println("Fin du programme");
 clavier.close();
}
```

Remarque :

Puisque les classe **ArithmeticException** et **InputMismatchException** sont des sous classes de la classe **Exception**, on peut les combiner dans bloc **catch** générique qui utilise la classe **Exception**


```
public class ExceptionsGeneriques {
  public static void main(String[] args) {
     int n;
     Scanner clavier = new Scanner(System.in);
     try {
        System.out.print("Saisir un entier : ");
        n = clavier.nextInt();
        System.out.println("1/" + n + " = " + 1/n);
     } catch (Exception e) {
        System.out.println("Erreur\n"+e.toString());
     }
     System.out.println("Fin du programme");
     clavier.close();
  }
}
```

Bloc finally

Le bloc **finally** est toujours exécuté, même si aucune exception ne s'est produite.

Exemple :

```
public class BlocFinally {
   public static void main(String[] args) {
     int tab[] = new int[2];
     try {
        tab[2] = 1;
     } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception : ");
        e.printStackTrace();
        //ou bien e.printStackTrace(System.out);
     } finally {
       tab[0] = 6;
       System.out.println("tab[0] = " + tab[0]);
       System.out.println("Le bloc finally est
          execute");
    }
   }
}
```

Blocs try imbriqués

On peut mettre un bloc **try** dans un autre bloc **try**.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

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



```
try {
    //Permet d'ouvrir un fichier en ecriture
    BufferedWriter out = new BufferedWriter(
        new FileWriter("test.txt"));
    out.append("Simple test\n");
    for (int i = 0; i < 5; i++)
        out.append("i = " + i + "\n");
    out.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

```
try {
    //Permet d'ouvrir un fichier en lecture
    BufferedReader fichier = new BufferedReader(
        new FileReader("test.txt"));
    String ligne;

    try {
        //lecture du fichier ligne par ligne
        while((ligne = fichier.readLine()) != null){
            System.out.println(ligne);
        }
    } catch (IOException e) {
            e.printStackTrace();
    }

} catch (FileNotFoundException e1) {
        e1.printStackTrace();
}
```


Exceptions personnalisées

Soit la classe **Personne** définie comme suit :

```
class Personne {
   private String nom, prenom;
   private int age;
   public Personne(String nom, String prenom, int
      age){
     this.nom = nom.toUpperCase();
     this.prenom =
         prenom.substring(0,1).toUpperCase()
         + prenom.substring(1).toLowerCase();
     this.age = age;
   }
   public String toString() {
     return nom + " " + prenom + " " + age + " ans";
   }
}
```

Gestion d'une seule exception

Supposons qu'on veut générer une exception si l'utilisateur veut instancier un objet **Personne** avec une valeur négative pour l'age. Par exemple :

Personne p = new Personne("Oujdi","Ali",−9)

Pour cette raison, nous allons définir une nouvelle classe, appelée **AgeException** qui hérite de la classe **Exception** :

```
class AgeException extends Exception {
    public AgeException() {
        System.out.println("L'age ne doit pat etre
            negatif");
    }
}
```

La classe **Personne** va être modifiée de la façon suivante :

```
class Personne {
    private String nom, prenom;
    private int age;
    public Personne(String nom, String prenom, int
       age)
     throws AgeException {
        if (age < 0)
            throw new AgeException();
        else {
            this.nom = nom;
            this.prenom = prenom;
            this.age = age;
        }
    }
    ...
}
```


Remarque :

Dans une méthode, une instruction de type :

Personne p = new Personne("Oujdi","Ali",18);

génère une erreur de compilation du fait que le constructeur génère des exceptions, il faut gérer les exceptions en utilisant le bloc **try**.

Exemple :

```
public static void main(String[] args) {
  try {
     Personne p = new Personne("Oujdi", "Ali", 18);
     System.out.println(p);
  } catch (AgeException e) {
     System.out.println(e);
     System.exit(-1);
  }
}
```


Gestion de plusieurs exceptions

Supposons maintenant qu'on veut, en plus de l'exception pour l'age, on veut générer une exception si l'utilisateur veut instancier un objet **Personne** avec un **nom** et/ou un **prénom** qui contiennent un chiffre (par exemple : Personne p = new Personne("Oujdi12","Ali",19)).

Nous allons définir une nouvelle classe nommé **PersonneException** comme suit :

```
class PersonneException extends Exception {
   public PersonneException() {
       System.out.println("Un nom ne doit pas
          contenir de chiffres!");
   }
}
```


La classe **Personne** va être modifiée de la façon suivante :

```
class Personne {
  private String nom, prenom;
  private int age;

  public Personne(String nom, String prenom,
    int age)throws AgeException, PersonneException {
    boolean b = false;

    for (int i = 0; i <= 9; i++)
      if (nom.contains(String.valueOf(i))
          || prenom.contains(String.valueOf(i))) {
        b = true;
        break;
      }
```

suite de la classe Personne

```
    if (age < 0)
        throw new AgeException();
    else if (b)
        throw new PersonneException();
    else {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }
 }
    ...
}
```


Puisque le constructeur doit gérer deux exceptions, on doit les tester tous les deux lors de l'instanciation d'un nouveau objet :

Exemple :

```
public static void main(String[] args) {
    try {
        Personne p = new Personne("OUjdi", "a1Li",
            12);
        System.out.println(p);
    } catch (AgeException e) {
        System.out.println(e);
        System.exit(-1);
    } catch (PersonneException e) {
        System.out.println(e);
        System.exit(-1);
    }
}
```

Spécifier l'exception qu'une méthode peut générer

Une méthode qui peut générer une exception mais qui ne peut être capturée mais qui peut être capturée par une autre méthode doit spécifier l'exception en utilisant le mot clé **throws**.

Exemple :

Soit la classe **Etudiant** définie comme suit :

```
class Etudiant {
    private String nom, prenom, cne;
    private double[] note = new double[6];

    public Etudiant(String nom, String prenom,
       String cne) {
        ...
    }

    public void setNote(int i, double x) {
        note[i] = x;
    }
    ...
}
```

Supposons qu'on veut générer une exception si l'utilisateur veut mettre une note différente de -1 ou non comprise entre 0 et 20 (par exemple : setNote(3,29) ou setNote(3,−10)).

Pour cette raison, nous allons définir une nouvelle classe, appelée **NoteException** qui hérite de la classe **Exception** :

```
class NoteException extends Exception {
    public NoteException() {
        System.out.println("Une note doit etre egale
            a -1 ou comprise entre 0 et 20");
    }
}
```


La classe **Etudiant** va être modifiée de la façon suivante :

```
class Etudiant {
    ...
    public void setNote(int i, double x) throws
       NoteException {
        if ((x != -1) && (x < 0 || x > 20))
            throw new NoteException();
        else
            note[i] = x;
    }
    ...
}
```

Pour une utilisation dans une méthode « main() », on appelle la méthode « setNote() » comme suit :

```
try {
    et1.setNote(3, 23);
    System.out.println(et1.getNote(3));
} catch (NoteException e) {
    System.out.println(e);
}
```