République Tunisienne
Ministère de l'enseignement supérieur et de la recherche scientifique
Direction générale des études technologiques

Programmation Orientée Objet

Support de cours et séries de travaux dirigés

Par

Manel ZOGHLAMI
Assistante technologue à ISET Jendouba

Année universitaire 2011/2012

Programmation Orientée Objet

Support de cours et séries de travaux dirigés

Fiche matière

- **Domaine de formation** : Sciences et technologies
- **Mention** : Technologie de l'informatique (TI)
- **Parcours** : Tronc Commun
- **Semestre** : S3
- **Unité d'enseignement** : Programmation orientée objet
- **Volume horaire** : 45 heures (22,5 cours – 22,5 TD)
- **Coefficient** : 2

Objectifs du cours

- Comprendre les concepts de programmation orientée objet: comprendre une conception orientée objet en vue de la traduire en programme orienté objet.
- Implémenter des classes d'objets : implémenter une classe en précisant ses attributs et ses opérations en précisant leurs visibilités.
- Utiliser les tableaux d'objets et écrire des programmes avec des objets de différentes classes.
- Créer des classes avec le principe d'héritage : à partir d'une classe déjà définie, créer une autre classe qui hérite la première.
- Comprendre l'utilité des classes abstraites et des interfaces et les utiliser dans des programmes Java.
- Gérer les exceptions dans un programme java.

Pré-requis

Programmation, algorithmique et structures de données

Moyens et Outils Pédagogiques

- Condensé du cours
- Travaux dirigés.
- Tableau
- Explication orale

Table des matières

Chapitre 1. Introduction au langage Java

Objectifs

Ce chapitre s'intéresse à faire une présentation générale du langage Java (caractéristiques et principe d'exécution). Il passe ensuite à présenter la syntaxe du langage, particulièrement l'utilisation des types et des variables, les opérateurs et les structures conditionnelles et itératives.

Pré-requis

- Algorithmique et structures de données 1

Eléments de contenu

1. Présentation du langage Java
2. Exécution d'un programme en Java
    - 2.1. La compilation d'un code source
    - 2.2. L'exécution d'un programme
    - 2.3. Les Environnements de Développement Intégré
3. Les règles générales de la syntaxe
4. Les commentaires
5. Les variables
    - 5.1. Les types primitifs
    - 5.2. Déclaration et affectation
    - 5.3. La conversion des types
6. Les opérateurs
    - 6.1. Les opérateurs arithmétiques
    - 6.2. Les opérateurs de comparaison
    - 6.3. Opérateurs logiques
    - 6.4. Opérateurs d'affectation
    - 6.5. L'incrémentation et la décrémentation
7. Les structures de contrôle
    - 7.1. Les structures conditionnelles
    - 7.2. Les structures itératives

1. Présentation du langage Java

Java est un langage de programmation orienté objet. Il a été développé par la société Sun Microsystems (rachetée dernièrement par Oracle Corporation). Les principales caractéristiques de Java sont les suivantes :

- ***Java est orienté objet***. Donc il tire profit des avantages de la programmation orientée objet. Cette dernière sera abordée dans le chapitre 3.
- ***Java est compilé-interprété*** : Java n'est pas un langage totalement compilé ou totalement interprété. L'exécution se fait sur deux étapes: le code source est compilé en pseudo code puis ce dernier est exécuté par un interpréteur Java appelé Java Virtual Machine (JVM). Ce concept est à la base du slogan de Sun pour Java : WORA (Write Once, Run Anywhere : écrire une fois, exécuter partout).
- ***Java est indépendant de toute plate−forme*** : Le pseudo code généré ne contient pas de code spécifique à une plate−forme particulière. Il peut être exécuté sur toutes les machines disposant d'une JVM et obtenir quasiment les mêmes résultats. On dit que Java est un langage *portable*.
- ***Java*** possède ***une API riche***. Le terme API (pour Application Programming Interfaces) regroupe l'ensemble des librairies, classes, interfaces,etc prédéfinies en Java. Java possède une API très riche qui facilite la tâche des développeurs et permet un gain de temps.
- ***Java assure la gestion de la mémoire*** : l'allocation de la mémoire est automatique pour les objets créés. Java récupère automatiquement la mémoire inutilisée grâce à un système appelé ramasse miette (connu par garbage collector en anglais) qui libère les espaces de mémoire libres.
- ***Simplicité*** (relative): Le développement en Java ne contient pas la manipulation des éléments généralement jugés compliqués tels que la notion de pointeurs (pour éviter les incidents en manipulant directement la mémoire) et l'héritage multiple. En plus, la syntaxe de Java proche de celle du C/C++ contribue à sa simplicité.
- ***Java est fortement typé*** : Toutes les variables doivent avoir un type et il n'existe pas de conversion automatique qui risquerait une perte de données. Ceci est une source de la robustesse du langage Java.
- ***Java est multitâche*** : il permet l'utilisation de threads (processus légers) qui sont des unités d'exécution isolées.

Java a actuellement trois éditions principales:

- J2SE : Java 2 Standard Edition. Cette édition est destinée aux applications pour poste de travail.

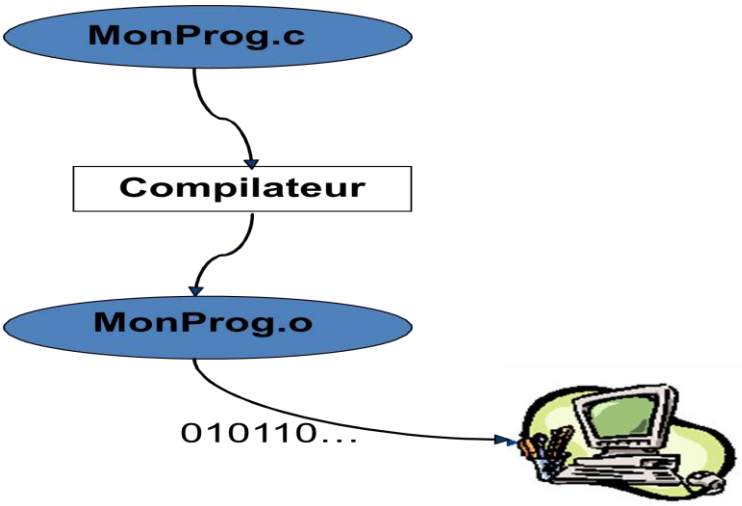
- J2EE : Java 2 Entreprise Edition. Cette édition est spécialisée dans les applications serveurs. Elle inclut le développement des applications Web.
- J2ME : Java 2 Micro Edition. Cette édition est spécialisée dans les applications mobiles.

Remarque: ***JDK et JRE et Java***

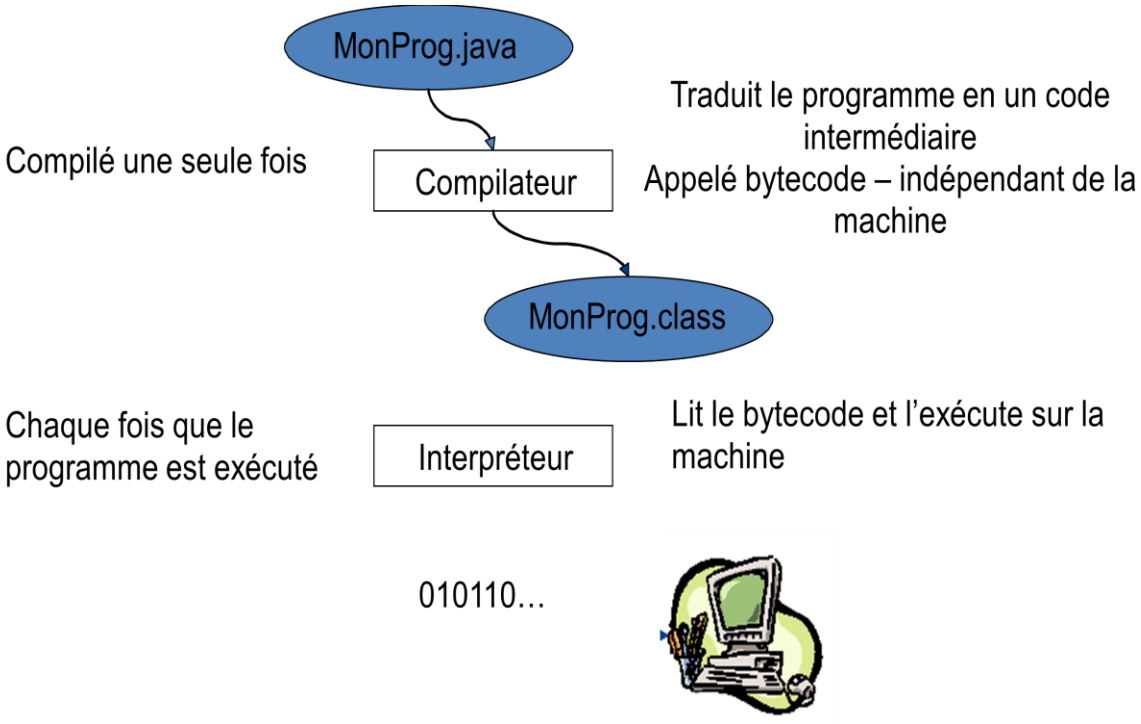
Un ensemble d'outils et de programmes permettant le développement de programmes avec Java se trouve dans le Java développement Kit connu par JDK. La JRE (pour Java Runtime Environment) est une partie du JDK qui contient uniquement l'environnement d'exécution de programmes Java. Le JRE seul doit être installé sur les machines où des applications Java doivent être exécutées.

2. Exécution d'un programme en Java

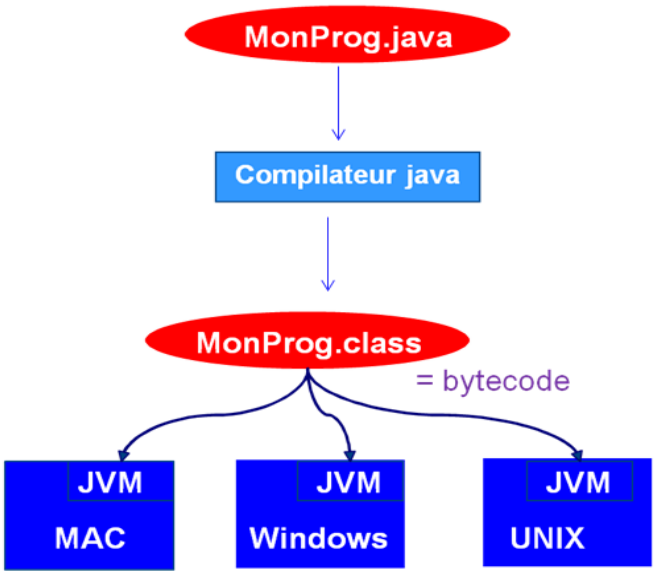
Pour exécuter un programme en C, on le compile. On génère alors un code natif spécifique à l'environnement d'exécution de la machine. Ce code est directement exécutable.



Dans le cas de Java, c'est différent. Il est nécessaire de compiler la source pour la transformer en pseudo-code Java (code intermédiaire ou byte code) indépendant de la plateforme d'exécution. Ce pseudo-code sera ensuite exécuté par la machine virtuelle (JVM) installée sur la machine. On rappelle ici la devise de java : écrire une fois , exécuter partout.



La figure suivante re-illustre ce principe: le compilateur génère un fichier d'extension .class indépendant de la plateforme d'exécution qui sera ensuite exécuté dépendamment de la plateforme. Ceci fait la portabilité de Java.



2.1. La compilation d'un code source

N'importe quel éditeur de texte peut être utilisé pour éditer un fichier source Java. Pour compiler ce fichier, il suffit d'invoquer la commande javac avec le nom du fichier source avec son extension .java

javac NomFichier.java

Suite à la compilation, le pseudo code Java est enregistré sous le nom NomFichier.class

2.2. L'exécution d'un programme

Un programme ne peut être exécutée que si il contient une méthode main() correctement définie. Pour exécuter un fichier contenant du byte-code il suffit d'invoquer la commande java avec le nom du fichier source sans son extension .class

java NomFichier

2.3. Les Environnements de Développement intégrés

En pratique, les développeurs Java utilisent des Environnements de Développement Intégrés (EDI ou IDE en anglais pour integrated development environment) pour créer des application Java. Un EDI est un programme regroupant un ensemble d'outils pour le développement de logiciels. En règle générale, un EDI regroupe un éditeur de texte, un compilateur, des outils automatiques de fabrication, et souvent un débogueur. Parmi les EDI les plus connus pour le langage Java, on trouve eclipse, Netbeans et JCreator.

3. Les règles générales de la syntaxe

Java est sensible à la casse: il fait la différence entre majuscule et minuscule.

Exemple: La variable nommée id et celle nommée ID représentent deux variables différentes.

Chaque instruction Java finit pas par un ;

Une instruction peut tenir sur plusieurs lignes

Exemple:

int age = 20; est équivalent à

```
int age
=
20;
```

L'indentation est ignorée de la part du compilateur mais elle permet une meilleure compréhension du code.

Chaque variable ou sous-programme est associé à un nom : l'identificateur. Il peut se composer de tous les caractères alphanumériques et des caractères _ et $. Le premier caractère doit être une lettre, le caractère de soulignement ou le signe dollar. Un identificateur ne peut pas appartenir à la liste des mots réservé du langage Java

4. Les commentaires

Les commentaires ne sont pas pris en compte par le compilateur. Un commentaire sur une seule ligne commence par '//', celui portant sur une ou plusieurs lignes est écrit

entre '/*' et '*/'. Les commentaires sont généralement utilisés par les développeurs pour expliquer le code.

Exemples :

```
int age=20; // âge de l'étudiant
int nom; /* c'est le  nom
de l'étudiant */
```

Il existe un type particulier de commentaires qui servent à enrichir le code écrit: les commentaires de documentation automatique.

Exemple:

```
/**
* commentaire de la méthode
* @param … (les paramètres)
* @return …. (type de retour )
*/
```

5. Les variables

5.1. Les types primitifs

Le langage Java utilise plusieurs types élémentaires, appelés aussi les types **primitifs**. Ils commencent tous par une minuscule.

Type	**Désignation**	**Longueur**	**Valeurs (à titre indicatif)**
Byte	octet signé	1 octet	−128 à 127
Short	entier court signé	2 octets	−32768 à 32767
Int	entier signé	4 octets	−2147483648 à 2147483647
Long	entier long	8 octets	−9223372036854775808 à 9223372036854775807
float	virgule flottante simple précision (IEEE754)	4 octets	1.401e−045 à 3.40282e+038
double	virgule flottante double précision (IEEE754)	8 octets	2.22507e−308 à 1.79769e+308
Char	caractère Unicode	2 octets	\u0000 à \uFFFF
boolean	valeur logique : true ou false	1 bit	true ou false

- **Catégorie logique : Type boolean** En java, un booléen n'est pas considéré comme une valeur numérique, les seules valeurs possibles sont true ou false.
  exemple: boolean trouve=true;

- **Catégorie caractère : Type char** Le type 'char' permet de représenter les caractères (caractère Unicode représenté sur 16 bits). Il est délimité par deux apostrophes. Exemples:

  char x='a'; char omega=(char) 969;
  char z='\u1200'

- **Catégorie entier : types byte, short, int et long** Il existe 4 types d'entiers en java. Chaque type est déclaré avec l'un des mots clés byte, short, int et long.

Exemple: int x =1;

- **Catégorie flottant : type float et double** Une constante numérique est flottante si elle contient un point décimal, une partie exponentielle (lettre E) ou si elle est suivie par la lettre F ou D.

  Exemples:
  2.3F
  2.3
  2.3E4
  4F

Remarque 1: Les types par références A part les types primitifs qui possèdent des tailles standards et fixes, il existe des types qui ne peuvent pas avoir une taille standard. Il s'agit des types références (tableaux, classes) qui seront abordés dans les chapitres suivants.

Remarque 2: Les chaînes de caractères Pour manipuler les chaînes de caractères Java offre le type *String*. Une constante chaîne de caractères est placée entre doubles cotes.

5.2. Déclaration et affectation

La déclaration se fait avec la syntaxe suivante

Type nomVariable ;

Par convention, le nom de la variable en Java commence par une minuscule. Il est possible de définir plusieurs variables de même type dans une seule ligne en séparant chacune d'elles par une virgule. On peut aussi initialiser une variable lors de sa déclaration. Le signe = est l'opérateur d'affectation.

Exemples:

```
int a,b;
int c = 2;
```

5.3. La conversion des types

Si dans une expression les opérandes sont de différents types le résultat est alors converti vers le type le plus grand.

Exemple :

```
int x=2; long y=4, z;
z=x*y; //le résultat est alors convertis en long
```

Pour l'affectation, la donnée à droite est convertie dans le type de celle de gauche. Si le type de la destination est plus faible l'instruction est considérée erronée.

Exemple :

```
int i=2; long x=5;
x=i; //instruction légale
i=x; // instruction erronée
```

On peut convertir explicitement une valeur en **forçant la transformation**. → On parle de "**cast**" ou "forçage de type". Le type ciblé est placé entre ( ) et est utilisé comme préfixe de l'expression dont on veut modifier le type. La syntaxe est:

<nom_variable> = (<type>) <expression>

Dans l'exemple précédent on écrit:

```
int i=2; long x=5;
i=(int)x; // instruction non erronée
```

Autre exemple:

```
float z; z=(float) 14.33;
int x;   x=(int) z; //x contient la valeur 14
```

➔ Le cast peut donc engendrer une perte d'information. (14.33 → 14)

6. Les opérateurs

6.1. Les opérateurs arithmétiques

Java reconnaît les opérateurs arithmétiques usuels suivants :

+ (addition), **−** (soustraction), ***** (multiplication), **/** (division) et **%** (reste de la division). Ils peuvent se combiner à l'opérateur d'affectation comme en C.

Exemple: nombre += 10;

Il existe diverses fonctions mathématiques prédéfinies. Voici quelques-unes:

Math .sqrt (x)	racine carree
Math .pow(*x, y*)	x à la puissance y
Math .abs(x)	valeur absolue
Math .cos(x)	Cosinus
Math .sin(x)	Sinus

Exemple:

```
public static void main(String[] args) {
    int a = -3;
    int b= Math.abs(a); // valeur absolue
    double c = Math.pow(a, 2); // a au carré (retourne un double)
    double d = Math.sqrt(c);// racine carrée (retourne un double)
```

```
        // affichage
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
    }
```

Exécution:

```
-3
3
9.0
3.0
```

6.2. Les opérateurs de comparaison

Le résultat d'une expression de comparaison est un booléen. Comme en C, Les opérateurs sont les suivants:

Opérateur	Exemple	Signification
>	a > 10	strictement supérieur
<	a < 10	strictement inférieur
>=	a >= 10	supérieur ou égal
<=	a <= 10	inférieur ou égal
= = (double = , à ne pas confondre avec le signe d'affectation)	a = = 10	Egalité
!=	a != 10	diffèrent de

6.3. Opérateurs logiques

Les opérations logiques sont les suivantes:

Opérateur	**Exemple**	**Signification**	**Exemple**
&&	a && b	ET logique	Si ((condition1) && (condition2)) alors …
||	a || b	OU logique	Si ((condition1) || (condition2)) alors …
!	!a	Non logique	Si (!condition) alors …

6.4. Opérateurs d'affectation

Il est possible d'écrire a+=b qui signifie a=a+b.

Exemple:

```
int a = 3;
 int b   = 2;
a+=b;
// affichage
```

```
System.out.println(a);

On affiche 5
```

De même on a:

a- =b → a=a-b
a*=b → a=a*b
a/=b → a=a/b

6.5. L'incrémentation et la décrémentation

Comme en C, les opérateurs ++ et -- permet d'incrémenter / décrémenter la valeur dune variable.

Exemple:

```
int a =4, b=4;
a++; // a est égal à 5
b- -; // b est égal à 3
```

Si l'opérateur est placé avant la variable (il est alors dit préfixé), la modification de la valeur est immédiate, sinon la modification n'a lieu qu'à l'issu de l'exécution de la ligne d'instruction ((il est alors dit opérateur postfixé).

Exemple:

```
int x = 2;
System.out.println(x++);    → on affiche 2 puis on incrémente
Ceci est équivalent à    System.out.println(x); x = x + 1;

int x = 2;
System.out.println(++x); → on incrémente puis affiche 3
x = x + 1; System.out.println(x);
```

Remarque: Il existe aussi les opérateurs de bits, qui ne sont pas couramment utilisés:

7. Les structures de contrôle

7.1. Les structures conditionnelles

Le programmeur est très souvent amené à tester des valeurs et à orienter le programme selon ces valeurs.

a) Structure conditionnelle si.. alors .. sinon

La syntaxe en Java est :

```
if (<expression booléenne>)  {
    //instruction ou bloc d'instruction à exécuter si condition= vrai.
```

```
}else{
      //instruction ou bloc d'instruction à exécuter si condition= faux.
         }
```

Pour cette syntaxe, la clause else est optionnelle. On peut imbriquer des structures conditionnelles.

Exemple

```
float moyenne = 16;
   if (moyenne>=10 ){
   System.out.println("Réussite!");}
   else {System.out.println("Echec!");
   }
```

Exécution: Réussite!

b) L'opérateur ternaire

Il s'agit d'une forme simplifiée de la structure conditionnelle précédente. Il s'agit d'évaluer une expression et, selon la valeur logique prise par cette expression, on affecte une valeur ou une autre à une variable. La syntaxe est :

```
variable= ( condition ) ? valeur−vrai : valeur−faux;
```

Exercice

Donner une version utilisant l'opérateur ternaire équivalente à l'exemple précédent utilisant la structure si/alors.

Solution

```
  float moyenne = 16;
  String resultat=( moyenne>=10 ) ? "Réussite!" : "Echec!";
  System.out.println(resultat); // afficher le résultat
```

Exécution: Réussite!

c) La structure conditionnelle selon.. faire

Dans la structure conditionnelle à *alternatives,* une *expression* peut être testée par rapport à plusieurs valeurs possibles. La syntaxe est la suivante:

```
switch (<expression >)
      {
```

```
        case <valeur 1> :    <bloc 1>    break;
        case <valeur 2> :    <bloc 2>    break;
        … …
        case <valeur n> :    <bloc n>    break;
        default : <bloc n+1>
    }
```

- La spécification du **break est nécessaire** pour gérer les ruptures de séquences.
- On ne peut utiliser switch qu'avec des types primitifs suivants: byte, short, int, char.

Exercice

Donner le résultat d'exécution des deux codes suivants:

Code 1	Code2

Int k=0, i=2;
switch(i)
{
case 1: k+=20; break;
case 2: k+=2; break;
case 3: k+=10; break;
}
System.out.println("k= "+k);
	int k=0, i=2;
	switch(i)
	{
	case 1: k+=20;
	case 2: k+=2;
	case 3: k+=10;
	}
	System.out.println("k= "+k);

Solution

Le résultat du code 1 est "k=2". Le deuxième code ne contient pas l'instruction *break*, on va donc exécuter les deux dernières clauses *case*. Le résultat est donc "k=12".

7.2. Structures de contrôle itératives

Une boucle sert à exécuter un ensemble d'instructions répétées plusieurs fois.

a) La boucle pour

La boucle pour est utilisée si le nombre de répétitions est connu à l'avance. La syntaxe est la suivante:

```
for (<initialisation> ; <expression_booléenne> ; <incrémentation> )
{
        <instruction ou bloc d'instruction>
}
```

Exemple:

```
for (int i=1 ; i<3 ; i++)      // Attention aux ";" entre les 3 parties
        { System.out.println(i) ;
}
```

Exécution:

```
1
2
```

Il est à noter que:

- for (; ;) est une boucle infinie.
- Dans l'initialisation, on peut déclarer une variable qui sert d'index (la variable i dans l'exemple) →Elle est dans ce cas **locale à la boucle.**
- Il est possible d'inclure plusieurs traitements dans l'initialisation et la modification de la boucle: chacun des traitements doit être séparé par une **virgule**.

Exemple:

```
int i,j;
for (i = 0 , j = 0 ;    i + j < 10;     i++ , j+= 3) {
….   }
```

Exercice:

Donner un code Java permettant de calculer puis afficher le factoriel d'un entier n.

Solution:

```
int n =3;
int fact = 1;
for (int i=1; i<=n; i++)
   fact = fact * i;
System.out.println(i+ "! = " + fact);
```

Exécution

3!=6

b) Les boucles tant que.. faire et répéter .. jusqu'à

Ces deux boucles sont généralement utilisées quand le nombre d'itérations est inconnu à l'avance. La syntaxe est la suivante :

while (<expression_booléenne>)
{
  <instruction ou bloc>
}
	Do
	{
	  <instruction ou bloc>
	}
	while (<expression_booléenne>) **;**

▪ On boucle tant que la condition est vérifiée. ▪ On teste la condition au début de la boucle, donc il y a une possibilité de ne pas entrer dans la boucle :	▪ On sort de la boucle quand la condition est vérifiée. ▪ On teste la condition à la fin: instructions exécutées au moins une fois.

instructions exécutées 0 ou plusieurs fois.	

Exercice:

Donner la somme 1+2+3+4 en utilisant les deux boucles.

Solution:

int s = 0, i = 1;
while (i <= 4) {
s += i;
i++;
}
System.out.println(s);
 	
Exécution dans les 2 cas:
10
	int s = 0, i = 1;
	do {
	s += i;
	i++;
	} while (i <= 4) **;**
	System.out.println(s);

Remarques:

break : permet de quitter immédiatement une boucle ou un branchement.

continue : est utilisé dans une boucle pour passer directement à l'itération suivante

Exemple:

int n = 5;
for (int i = 1; i <= n; i++) {
System.out.println(i);
}
 	
Exécution:
1
2
3
4
5
	int n = 5;
	for (int i = 1; i <= n; i++) {
	if (i = = 3) {
	continue ; }
	System.out.println(i);
	}
	Exécution:
	1
	2
	4
	5
		int n = 5;
		for (int i = 1; i <= n; i++) {
		if (i = = 3) {
		break ; }
		System.out.println(i);
		}
		
		Exécution:
		1
		2

8. Série d'exercices

Exercice 1

Ecrire un programme qui initialise deux entiers puis affiche leur somme et leur produit.

Solution

```
public static void main(String[] args) {
   int a = 4;
   int b = 3;
    int somme= a+b;
    int produit= a*b;
    System.out.println("La somme est "+ somme+ " - Le produit est : " + produit);
}
```

Exercice 2

Écrire un programme qui affiche successivement les factorielles des N premiers entiers. La constante N sera déclarée et initialisée dans main comme suit :

final int N = 4;

Solution

```
public static void main(String[] args) {
   final int N = 4;
   int n;
   int fact;

   n = 1;
   fact = 1;
   while ( n <= N ) {
     fact = fact * n;
     System.out.println(n+"! = "+fact);
     n = n + 1;
   }
 } // fin de main
```

Exécution

```
1! = 1
2! = 2
3! = 6
4! = 24
```

Exercice 3

a) Ecrire un programme qui calcule la nième valeur de la suite de Fibonacci qui définie par

U0 = 1
U1 = 1
Un = Un-1 + Un-2, pour n>2

b) Vérifier le résultat pour n =4.

Solution

```
public static void main(String[] args) {
    int n = 4;
    for (int i = 1; i <= n; i++) {
        System.out.println("U"+i + "= " + fib(i));      }
}
public static long fib(int n) {
    if (n <= 1) {
       return n;
    } else {
       return fib(n - 1) + fib(n - 2);
    }
}
```

Exécution :

U1= 1
U2= 1
U3= 2
U4= 3

Exercice 4

Ecrire un programme qui permet de calculer le PGCD de deux nombres donnés. On utilisera l'algorithme d'Euclide: PGCD(a,b) = PGCD(a-b,b) (pour a > b) de manière itérative.

Exemple:

PGCD(6,9) = PGCD(6,3) = PGCD(3,3) = 3

Solution

```
public static void main(String[] args) {
   int a = 6;
   int b = 9;

   if ( a > 0 && b > 0 ) {
     System.out.print("PGCD("+a+","+b+") = ");
     while ( a != b ) {
         if ( a < b )
           b = b - a;
```

```
            else
              a = a - b;
          System.out.print("PGCD("+a+","+b+") = ");
        }
        System.out.println(a);
      }
    }
```

Exécution
PGCD(6,9) = PGCD(6,3) = PGCD(3,3) = 3

Chapitre 2. Les bases de la Programmation Objet : Classe - Objet

Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Comprendre les notions de classe et d'objet
- Etre capable de développer une classe en précisant ses attributs, ses constructeurs et ses opérations, puis utiliser cette classe dans un programme Java.
- Comprendre le principe d'encapsulation et la surcharge des méthodes.

Pré-requis

- Algorithmique et structures de données
- Syntaxe de Java

Eléments de contenu

1. Les classes

1.1. Notion de classe

Les langages de programmations que vous avez vu jusqu'à maintenant (Pascal et C) ne sont pas orientés objet. Vous avez vu la **programmation procédurale**: un ensemble de procédures et fonctions qui appellent les unes les autres. La programmation orientée objet (POO) est basé sur la notion d'objet. C'est une notion très proche de celle de la vie courante qu'on va découvrir dans ce chapitre.

La POO est basée sur l'utilisation d'un ensemble d'unités appelées classes. Ces dernières offrent généralement une description des objets concrets ou abstraits du monde réel.

Exemple: Pour développer un programme de gestion de *votre ISET*, on vous propose de créer les classes suivantes: classe Etudiant, classe Enseignant, classe matière, classe module, classe laboratoire….

Une classe offre le modèle (ou description abstraite) de ses objets, in s'agit d'un ensemble de données et d'opérations qui permettent de manipuler ces dernières regroupées dans une même entité. Les données sont généralement appelées attributs et les opérations sont appelées méthodes. Java est un langage orienté objet : généralement tout appartient à une classe, sauf les variables de type primitives.

Exemple : la classe personne

Personne	
Prénom: chaine de caractères Nom: chaine de caractères Age : entier …	→Des attributs
afficherNom() incrementerAge() …	→Des méthodes

1.2. Déclaration des classes

Une classe se compose en deux parties : l'en-tête et le corps. Le corps peut être divisé en 2 sections : la déclaration des attributs (des variables contenant les données) et la définition des méthodes (elles implémentent les traitements).

```
[Modificateurs] class nomDeClasse [extends classe_mere] [implements interface] {

        // Déclaration des attributs
[<modificateur >] <type> <nom>[=<expression>];
```

```
        //Déclaration les méthodes
[<modificateur >] <type_retour> <nom>([<arguments>])
{       // les instructions}

} // fin classe
```

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans la méthode A.

Il est à noter que les modificateurs de classe sont des mots réservés tels que "public", "private"... Ils vont être abordés plus tard. Pour le moment, on ne va pas les utiliser. De même, les modificateurs des méthodes et des attributs ainsi que les mots clé implements et extends seront traités plus tard. On va donc se contenter de la forme la plus réduite.

Exemple: Une classe Personne qui modélise une personne en Java.

```
class Personne {
   //_____les attributs
    String prenom;
   String nom;
    int age;

   //_____les méthodes

//une méthode qui affiche le prénom de la personne
   void afficherPrenom() {
      System.out.println("le prénom de la personne est:" + prenom);
   }

//une méthode qui incrémente l'age de la personne
   void incrementerAge() {
      age++;
   }
}
```

Remarque : **Conventions de nommage en Java**

Ce sont des conventions: on peut ne pas suivre ces règles et avoir un programme Java exécutable sans erreurs. Cependant, il est recommandé de les suivre pour augmenter la qualité du code.

1) Les noms de classes (et des interfaces à voir dans le chapitre 4) commencent par des majuscules.

Exemple: class **E**tudiant – class **L**ivre

2) Les noms des attributs, des variables et des méthodes commencent par des minuscules.

Exemple: int **c**ompteur; - void **a**fficher() {…}

3) Si le nom est une concaténation de plusieurs mots alors les premières lettres de tous les mots, sauf le premier, seront écrites en majuscule. Ceci permet de délimiter les mots visuellement.

Exemple:
class **E**tudiant**E**n**I**nformatique
int **c**ompteur**L**ivre;
void **a**fficher**E**tudiant() {…}

4) Le nom d'une constante est entièrement en majuscule.
Exemple: MAXIMUM

Exercice

1) Donner la classe **ISET** permettant de représenter une ISET par un identifiant (entier), une adresse, nombre d'étudiants. Doter cette classe par une méthode d'affichage et une méthode qui ajoute augmente le nombre des étudiants avec un entier passé comme paramètre.

2) Donner la classe **Ordinateur** qui permet de représenter un ordinateur par un identifiant, une marque et la taille de la RAM. Doter cette classe par une méthode d'affichage.

3) Classe **matière** qui représente une matière par un identifiant, un nom, une note oral, une note de DS et une note d'exam. Elle a une méthode calculerMoyenne qui calcule la moyenne dans une matière avec la façon suivante : Moyenne= 02*oral+0.4*DS + 0.4*Examen

Solution

```
class Iset {
   // attributs
   int identifiant;
   String adresse;
   int nombreEtudiants;

   // méthodes
   void afficherIset() {
      System.out.println("l'Iset n°" + identifiant + " située à " + adresse + "contient " +
```

```
nombreEtudiants + " étudiants. ");
   }

void ajouterEtudiants(int nbrAjoutes) {
      nombreEtudiants = nombreEtudiants + nbrAjoutes;
   }
}
```

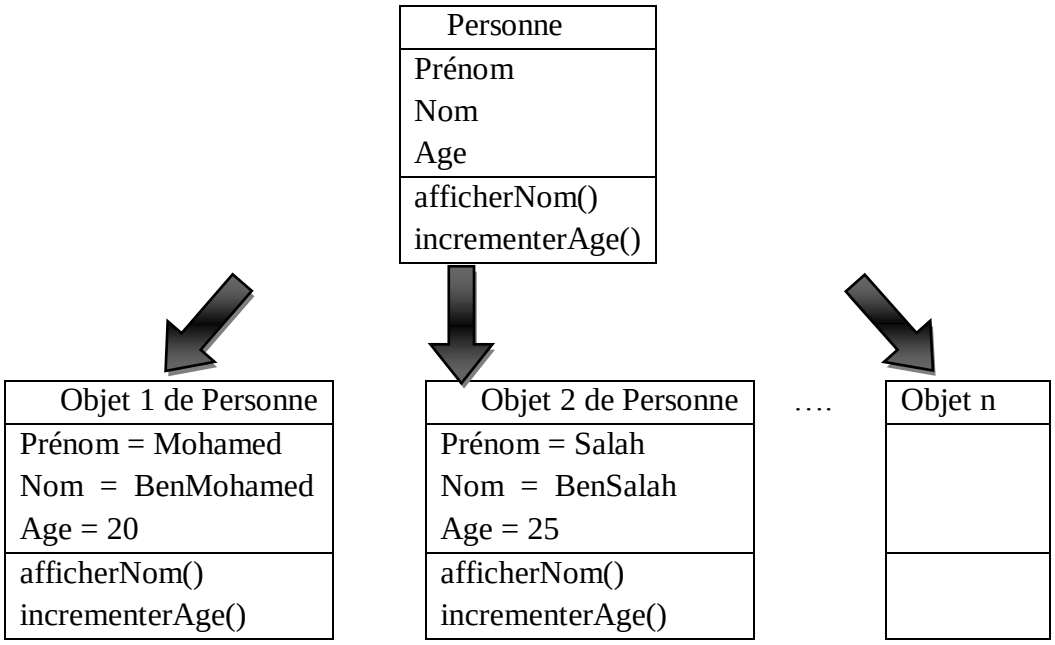
```
class Ordinateur {
   int id;
   String marque ;
   double tailleRam;

 void afficherOrdinateur (){
            System.out.println("****Ordinateur****\n id: "+ id + "Marque: "+marque);
   }
}
```

```
class Matiere {
// les attributs
   int id;
   String nom;
   double noteOral, noteDS, noteExam;
// le méthodes
   double calculerMoyenne() {
      double res;
      res = 0.2 * noteOral + 0.4 * noteDS + 0.4 * noteExam;
      return res;
   }
}
```

2. Les Objets

2.1. Notion d'objet

Un objet est l'unité de base de la conception OO. On a dit qu'une classe est la description d'un objet. Un objet est une instance d'une classe: instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable. Pour chaque instance d'une classe, le code est le même, seules les données sont différentes à chaque objet.



2.2. La création d'un objet

Elle consiste à l'instanciation de la classe relative à cet objet. Il faut différencier entre la création d'un objet et sa déclaration.

a) **Déclaration d'un objet:** Elle est similaire à la déclaration d'une variable de type primitif: Déclarer une variable ayant comme type la classe que l'on désire instancier. Cette variable n'est pas l'objet lui-même mais une référence à cet objet. La déclaration est de la forme:

```
nomClasse  nomObjet;
```

Exemple :

```
Personne p1; // déclarer un objet p1 de la classe Personne
Personne p2; // déclarer un autre objet p2 de la classe Personne
```

b) Création de l'instance d'un objet: L'opérateur **new** se charge de créer une instance de la classe et de l'associer à la variable, il s'agit ici d'allouer l'espace mémoire nécessaire à l'objet et renvoyer l'adresse de l'objet à la variable référence.

Exemple : p1 = new Personne **( )**;

Il est possible de tout réunir en une seule ligne comme suit:

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

L'opérateur new est suivi du constructeur.

2.3. Les constructeurs

Un constructeur est une méthode particulière appelée au moment de la création d'un objet. Deux règles importantes pour le constructeur:

- ✓ Le nom du constructeur doit être le même que le nom de la classe
- ✓ Il n'a pas de type de retour.

Pour une même classe, on peut associer plusieurs constructeurs. Ils doivent nécessairement avoir des paramètres différents (nombre, type et ordre des paramètres.) On dit alors qu'on a **surchargé** le constructeur. On utilise principalement ces trois manières de définir un constructeur :

1) **le constructeur vide sans paramètres**, c'est le plus simple. Si on ne définit aucun constructeur pour la classe, un constructeur pareil est définit par défaut.

Exemple: public Personne () {}

2) le constructeur avec initialisation fixe.

Exemple :

```
public Personne() {
age = 5;
}
```

3) Le constructeur avec initialisation des attributs à partir des paramètres.

Exemple:

```
public Personne(String prenom1, String nom1, int age1) {
      prenom = prenom1;
      nom = nom1;
       age = age1;
   }
```

Si on veut lui ajouter des constructeurs, la classe "Personne" devient comme suit:

```
class Personne {
// les attributs
    String prenom;
   String nom;
    int age;
    //les constructeurs
    public Personne() {
    }
```

```
    public Personne(String prenom1, String nom1, int age1) {
      prenom = prenom1;
      nom = nom1;
       age = age1;
    }
    // les méthodes
    void afficherPrenom() {
       System.out.println("le prénom de la personne est: " + prenom);
    }
    void incrementerAge() {
       age++;
    }
}// fin personne
```

Remarque: Le constructeur par défaut

La définition d'un constructeur n'est pas obligatoire. En fait, si le développeur ne spécifie aucun constructeur pour la classe, Java appelle un constructeur par défaut créé automatiquement qui correspond à un constructeur vide sans paramètres. Par exemple pour la classe Personne, le constructeur par défaut est de la forme:

public Personne (){ }

Cependant, dès que le développeur défini explicitement un constructeur, Java ne va plus utiliser le constructeur par défaut (Java considère que le développeur prend en charge la création des constructeurs). Si on en a besoin, on doit définir explicitement le constructeur sans paramètres.

Exercice

Ajouter un constructeur qui initialise les attributs pour les trois classes Iset, Ordinateur et Matière de l'application précédente.

Solution

```
    public Iset(int id1, String adr1, int nbrE1) {
       identifiant = id1;
       adresse = adr1;
       nombreEtudiants = nbrE1;

}
```

```
    public Ordinateur(int id1, String marque1, double tailleRam1) {
        id = id1;
        marque = marque1;
        tailleRam = tailleRam1;
}
```

```
public Matiere(int id1, String nom1, double noteOral1, double noteDS1, double noteExam1) {
        id = id1;
        nom = nom1;
        noteOral = noteOral1;
        noteDS = noteDS1;
        noteExam = noteExam1;
  }
```

2.4. Les destructeurs

Un destructeur permet d'exécuter du code lors de la libération, par le garbage collector, de l'espace mémoire occupé par l'objet. En Java, les destructeurs appelés finaliseurs (finalizers), sont automatiquement appelés par le garbage collector. Pour créer un finaliseur, il faut redéfinir la méthode finalize() héritée de la classe Object.

2.5. Utiliser un objet

L'accès à un attribut d'un objet donné se fait toujours à l'intérieur d'une méthode. On crée un objet de la classe puis on applique cette syntaxe: **nomObjet.nomAttribut**

Exemple:

```
Personne p1 = new Personne("Ahmed","Mohamed",12);
System.out.println ("L'âge de la personne est :" +p1.age);
```

De même, pour invoquer une méthode d'un objet donné, on crée l'objet puis on applique cette syntaxe: **nomObjet.nomMéthode (arguments);**

Exemple:

```
p1.incrementerAge();
```

Pour tester une classe, on crée souvent une classe de test permettant de créer un objet puis invoquer les méthodes de la classe.

Exemple

```
public class TestPersonne {
 public static void main(String[] args) {
Personne pers = new Personne("Mohamed", "benMohamed", 20) ;
pers.afficherPrenom();
pers.incrementerAge();
System.out.println("l'âge devient: "+ pers.age);
 }
}
```

L'exécution donne:

le prénom de la personne est: Mohamed
l'âge devient: 21

Exercice

On veut tester les trois classes de l'exercice précédent.

1) Créer la classe TestIset possédant une méthode main permettant de tester la classe Iset. Dans la méthode main on doit:
    a) Créer un objet et initialiser ses attributs
    b) Appeler les méthodes de chaque objet.
    c) Donner le résultat de l'exécution attendu.
2) Créer la classe TestOrdinateur permettant de tester la classe Ordinateur avec la même démarche décrite dans 1.
3) Créer la classe TestMatière permettant de tester la classe Matière avec la même démarche décrite dans 1.

Solution:

```
class TestIset {
    public static void main(String[] args) {
   Iset iset = new Iset(123, "Jendouba", 1200);
   iset.afficherIset();
   iset.ajouterEtudiants(5);
   iset.afficherIset();
    }
}
```

Exécution:

```
l'Iset n°123 située à Jendouba contient 1200 étudiants.
l'Iset n°123 située à Jendouba contient 1205 étudiants.
```

```
class TestOrdinateur {
   public static void main(String[] args) {
      Ordinateur ord = new Ordinateur(12, "Dell", 1024);
      ord.afficherOrdinateur();
   }
}
```

Exécution

```
****Ordinateur****
 id: 12Marque: Dell
```

```
public class TestMatiere {
 public static void main(String[] args) {
 Matiere mat = new Matiere(1, "POO", 11, 12, 11.5);
 double moy=mat.calculerMoyenne();
 System.out.println(moy);
 }
}
```

Exécution
11.600000000000001

2.6. Les références et la comparaison des Objets

Les références sont les variables qui permettent de désigner et manipuler les objets: ces variables ne contiennent pas un objet mais une référence vers cet objet. Lorsqu'on affecte une référence dans une autre (c.à.d on écrit p1 = p2 avec sont des objets), on copie la référence de l'objet p2 dans p1.

→ p1 et p2 réfèrent au même objet (ils pointent sur le même objet).

Généralement, 3 cas de comparaisons se présentent:

1) Pour comparer deux références, on utilise les opérateurs logiques suivants:
   - = =: teste si 2 références désignent le même objet.
   - != : teste si 2 référence ne désignent pas le même objet

Exemple:

```
Personne p1 = new Personne("Mohamed", "BenMohamed",20);
Personne p2 = new Personne ("Salah", "BenSalah",25);
if (p1 = = p1) { ... } // vrai
if (p1 = = p2) { ... } // faux
```

2) Pour comparer l'égalité de deux objets selon les valeurs de leurs attributs (et non pas les références), on doit doter la classe d'une méthode qui permet de le faire (ne pas utiliser l'operateur = =).
3) Pour s'assurer que deux objets sont de la même classe, il faut utiliser la méthode getClass().

Exemple :

```
If (obj1.getClass().equals(obj2.getClass()) {….}
```

Le 2ème cas est le cas le plus utilisé.

Exercice

Ajouter à la classe Personne la méthode egaleA qui permet de comparer deux personnes. On supposera que les deux personnes doivent avoir le même nom et prénom pour que la méthode retourne True.

Indication: Pour comparer deux chaines de caractères, on utilise la méthode equals avec la syntaxe suivante: chaine.equals(chaine2) . Cette méthode retourne true si les deux chaines sont identiques et false sinon.

Solution:

```
boolean egaleA (Personne p2){
if (this.nom.equals(p2.nom) && this.prenom.equals(p2.prenom))
   return true;
return false;
}
```

2.7. La référence this

Le mot résérvé *this* représente une référence sur l'objet courant d'utilisation (celui qui est entrain d'exécuter la méthode contenant le this). Cette référence est habituellement implicite :

```
void incrementerAge() { age++;  } est equivalent à :
void incrementerAge() { this.age++;   }
```

Le mot clé this peut être utilisé aussi dans ces deux cas :

a) Lorsqu'une variable locale ou paramètre cache, en portant le même nom, un attribut de la classe.

Exemple:

```
class Personne{
int age;
public Personne (int age) {
age = age; // attribut de classe = variable en paramètre du constructeur
} }
 →Dans ce cas on distingue les deux variables "age" en écrivant:
this.age = age;
```

b) Pour déclencher un constructeur depuis un autre constructeur.

Exemple

```
public class Personne {
   //_____les attributs
   String nom, prenom;
   int age;

   //Premier constructeur
```

```
        public Personne(String nom ) {
        this.nom=nom;
        }
    // Deuxième constructeur
        public Personne(String nom, String prenom, int age) {
         this (nom); // l'appel à this(..) doit être la première ligne dans ce constructeur
         this.age = age;
         this.prenom = prenom;
        }
```

2.8. Le mot clé null

Le mot clé null permet de représenter la référence qui ne représente rien. On peut assigner cette valeur à n'importe quelle variable contenant une référence. C'est aussi la valeur par défaut d'initialisation des attributs représentant des références.

```
Class TestPersonne{
        Personne p1; // objet initialisé à  null par défault.
        void methode()        {
                if (p1= =null)
                    p1=new Personne();
        …
        }
}
```

2.9. L'operateur instanceOf

Il permet de déterminer la classe de l'objet qui lui est passé en paramètre: il retourne true si l'objet à gauche est une instance de la classe placé à sa droite ou si la comparaison est faite entre un objet d'une classe implémentant une interface. Sinon il retourne false. La syntaxe est la suivante:

objet instanceof nomClasse

Exemple:

```
        class Personne { .....}

        class Test
        {
         public static void main (String[] args){
        Peronne p=new Personne();
        if (pers instanceof Personne)
           System.out.println("c'est une personne ");
```

```
if (pers instanceof Test)
   System.out.println("c'est une instance de la classe Test ");

}}
```

3. Le principe de l'encapsulation

En orienté objet, le regroupement d'attribut et de méthodes dans une "boite" appelée classe ainsi que le marquage des données est appelé: encapsulation. C'est principe important dans la philosophie de l'orienté objet.

3.1. Les membres publiques et les membres privés

L'encapsulation permet d'offrir à l'utilisateur de la classe (celui qui veut appeler une classe) une liste de services exploitables appelés interface. Cette dernière comporte les attributs et méthodes dits **publiques**. On peut donc accéder à ces membres à partir de l'extérieur de la classe. On utilise le mot clé public lors de leurs déclarations.

Exemple:

```
public int monAttribut;
public void maMéthode ( ){…}
```

On peut avoir une liste d'attributs et de méthodes réservés à l'utilisation interne d'un objet. Ces membres sont dits **privés.** Les attributs privés sont donc utilisables seulement dans les méthodes du même objet. On utilise le mot réservé *private* pour la déclaration

Exemple:

```
private int monAttribut;
private void maMéthode (){….}
```

3.2. Les accesseurs

Pour respecter le principe de l'encapsulation, les attributs sont généralement déclarés privés (sauf en cas nécessité). Ils ne peuvent être vus et modifiés que par des méthodes définies dans la même classe. Si une autre classe veut accéder à ces attributs, elle doit utiliser une méthode prévue pour cet effet : il s'agit de l'accesseur qui est une méthode publique qui donne l'accès à un attribut d'instance. Il s'agit de deux types:

- un accesseur en lecture qui commence par convention par **get** (d'où le nom *getter*). Il retourne la valeur de l'attribut en question.
- un accesseur en écriture qui commence par convention par **set**(d'où le nom *setter*). Il change la valeur de l'attribut.

Notre classe Personne devient donc:

```
public class Personne {
   //_____les attributs privés
   private String nom;
   private String prenom;
   private int age;

  //_____ les accesseurs publiques
      public int getAge() {
      return age;    }

   public void setAge(int age) {
      this.age = age;    }

   public String getNom() {
      return nom;    }

   public void setNom(String nom) {
      this.nom = nom;    }

   public String getPrenom() {
      return prenom;    }

   public void setPrenom(String prenom) {
      this.prenom = prenom;    }
}
```

4. Attribut d'instance et attribut de classe

Les attributs peuvent être des variables d'instances (ceux déjà vus jusqu'à maintenant), des variables de classes ou des constantes. Un attribut marqué par le mot clé ***static*** est un attribut de classe (appelé aussi attribut statique), sinon il s'agit d'un attribut d'instance. Les variables d'instance sont des variables propres à un objet. Un attribut statique est commun à tous les objets de la classe concernée

➔ Si on modifie cet attribut pour un objet donné, il sera modifié pour tous les objets de la classe. Une variable de classe permet de stocker une constante ou une valeur modifiée par plusieurs instances d'une classe.

Exercice:

1) Déterminer le résultat de l'exécution de ce programme.
2) Pourquoi l'attribut compteur n'est pas déclaré privé?

```
class Personne {
   //_____les attributs
  private   int id;
  static int compteur=0;

   //_____les constructeurs
   public Personne() {
     compteur++;
     id= compteur;    } // → ceci permet d'avoir des id successifs
}
```

```
class TestPersonne {
 public static void main(String[] args) {
Personne pers1 = new Personne() ; // id=1
Personne pers2 = new Personne() ;// id=2

// Le compteur  est commun entre les deux personnes:
pers1.compteur=44;
 System.out.println(pers2.compteur); }}
```

Solution:

1) Le programme affiche 44. Bien que modifié par l'instance pers1, et appelé par l'objet pers2, compteur contient 44.
2) L'attribut compteur n'est pas déclaré privé pour qu'on puisse l'appeler dans la classe TestPersonne.

On accède à un attribut statique selon deux manières:

a) Via une instance quelconque de la classe

Exemple: pers1.compteur

b) Via le nom de la classe

Exemple: Personne.compteur

Le mot clé final est utilisé pour déclarer une variable dont la valeur, une fois initialisée, ne peut plus être modifiée. On peut associer static à final pour avoir une constante commune à tous les objets de la classe.

Exemple: **final static** double PI=3.14;

5. Les méthodes

On a vu que les méthodes permettent d'implémenter les traitements de la classe. Elles sont définies selon la façon suivante:

```
[modificateurs]  typeDeRetour nomMéthode ( [parametres] ) {...
// définition des variables locales + les instructions
}
```

Le type de retour peut être un type élémentaire ou le nom d'une classe. Si la méthode ne retourne rien, alors on utilise void.

5.1. Passage des paramètres

Si on passe en paramètre d'une méthode une variable de type simple (int, float, boolean,…), le passage se fait par **valeur.** → Les modifications de la valeur d'un argument de type simple ne sont pas transmissibles à l'extérieur de la méthode, car ces dernières travaillent sur une copie de l'argument.
Pour transmettre la valeur modifiée d'un argument de type simple à l'extérieur d'une méthode, on peut:

- Soit retourner la valeur de la variable par la méthode elle-même
- Soit la définir comme attribut dans l'objet (Les opérations opèrent directement sur les attributs sans avoir besoin de les passer en paramètres)

Si on passe en paramètre d'une méthode **un objet**, le passage se fait par **référence:** Le contenu de l'objet peut etre modifié dans la méthode appelée, par contre la référence elle-même ne va pas être changée.

Exercice:
Déterminer le résultat de l'exécution de ce programme.

```
class Test {
private  int val = 11;

public  void modifEntier(int n) {
      n = 22;
   }
public  void modifObjet(Test obj) {
      obj.val = 22;
   }
   public static void main(String a[]) {
      Test t = new Test();
      int n = 0;
```

```
        // modifier n
        t.modifEntier(n);
        System.out.println("Valeur de n: "+n);

        // modifier la valeur  de l'attribut de l'objet
        t.modifObjet(t);
        System.out.println("Valeur de l'attribut val: "+t.val);
    }
}
```

Solution:

Le résultat de l'exécution est:
Valeur de n: 0
Valeur de l'attribut val: 22

5.2. La surcharge des méthodes

Java permet de réutiliser un même nom de méthode pour plusieurs fonctionnalités: on parle alors de la surcharge des méthodes. La surcharge permet de définir des méthodes portant **le même nom** mais acceptant **des paramètres différents** en types et/ou en nombre.

Exemple:

```
class MiniCalculatrice {
//méthode1
public int additionner (int a, int b){
 System.out.println("J'additionne deux entiers");return a+b;}

//méthode2
public double additionner (double a, double b){
System.out.println("J'additionne deux doubles");return a+b;}

//méthode3
public  int additionner (int a, int b, int c ){
System.out.println("J'additionne trois entiers");return a+b+c;}    }
```

Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des paramètres.

Exercice

Déterminer les méthodes de la classe MiniCalculatrice appelées successivement dans les lignes 5, 6 et 8.

```
1. class TestMiniCalculatrice {
2. public static void main(String[] args) {
3. MiniCalculatrice calc = new MiniCalculatrice();
4. int x = 2, y = 3;
5. System.out.println("resultat: " + calc.additionner(x, y));
   System.out.println("resultat: " + calc.additionner(x, 3, y));
6. double i = 2.2, j = 3.2;
7. System.out.println("resultat: " + calc.additionner(i, j));
8. }
9. }
```

Solution

L'ordre d'appel des méthodes est:

1) Méthode 1
2) Méthode 3
3) Méthode 2

Il est à noter que le type de retour des méthodes surchargées peut être différent mais cette différence à elle seule n'est pas suffisante, c'est-à-dire qu'il n'est pas possible d'avoir deux méthodes avec le même nom et les mêmes paramètres et dont seul le type retourné diffère.

Exemple:

```
int    additionner (int a, int b){ return a+b;}
double additionner (int a, int b){ return a+a+b;}
// → erreur, pour corriger on change le nom de la méthode, ou les paramètres
```

5.3. Méthodes de classe

Tout comme les attributs, on peut trouver des méthodes de classes. Une méthode de classe:

- ✓ est précédée par le mot clé ***static***
- ✓ ses actions concernent la classe entière
- ✓ Ne peut accéder qu'aux attributs statiques. Cependant, une méthode non statiques peut accéder aux attributs statiques.
- ✓ On n'y utilise pas la référence this (car elle peut être appelée sans l'intermédiaire d'un objet).

Comme dans le cas d'un attribut statique, on peut invoquer une méthode statique selon deux manières :

a) Via une instance quelconque de la classe

b) Via le nom de la classe (sans instancier un objet)

Exercice:

1) Créer la classe MathTool contenant un attribut statique *PI*, une méthode statique *foiPI* qui retourne la multiplication d'un nombre par *PI* et une méthode *carr*ée qui calcule le carré d'un nombre donné en paramètre.
2) Tester la méthode *carrée* de deux manières.

Solution

```
class MathTool {
   // Pi est une constante
private    final static double PI = 3.14;

public   static double foixPI(double x) {
       return x * PI;  // accés à une constante
   }
 public  static double carré(double x) {
       return (x * x);
   }
}

class Test {
   public static void main(String[] args) {
      // Première méthode de test: appel à partir de la classe
      double x = 6;
      double result1 = MathTool.foixPI(x);
      double result2 = MathTool.carré(x);
      System.out.println("x*PI= " + result1 + "  x*x= " + result2);

      // Deuxième méthode de test: appel à partir d'un objet
      MathTool tool = new MathTool();
      result1 = tool.foixPI(x);
      result2 = tool.carré(x);
      System.out.println("x*PI= " + result1 + "  x*x= " + result2);
   }
}
```

Exécution

```
x*PI= 18.84  x*x= 36.0
x*PI= 18.84  x*x= 36.0
```

La méthode *main* a la forme suivante:

*public **static** void main(String[] args) {….}*

C'est donc une méthode de classe, elle est la première à être appelée quand on exécute sa classe. Cette déclaration de la méthode main() est imposée par la machine virtuelle pour reconnaitre le point d'entrée d'une application.

6. Série d'exercices

Remarque: Dans vos réponses, respectez le principe d'encapsulation. Le code java doit être commenté.

Exercice 1

1) Définir une classe Date permettant de représenter le jour, le moi et l'année d'une date. Doter la d'un constructeur. La classe contient aussi les méthodes suivantes:
   - nbredeJours qui retourne le nombre de jours du moi de la date.
   - dateValide qui vérifie si une date est valide
   - lendemain qui retourne la date du jour d'après.
   - afficherDate qui affiche une date sous la forme jour/moi/année.
2) Créer la classe TesterDate contenat une méthode main qui
   - Crée une date et l'affiche.
   - Si la date est invalide, on affiche un message d'erreur, sinon on affiche le nombre de jours du moi puis la date du lendemain.

Solution

```
1)
class Date {
   // attributs
   private int jour,  moi,  annee;

   // Constructeur
   public Date(int jour, int moi, int annee) {
      this.jour = jour;
      this.moi = moi;
      this.annee = annee;
   }

   // méthode qui retourne le nombre de jours du moi
   public int nbreJours() {
      int nb;
      switch (moi) {
         case 1:
         case 3:
         case 5:
         case 7:
```

```
        case 8:
        case 10:
        case 12:
            nb = 31;
            break;

        case 4:
        case 6:
        case 9:
        case 11:
            nb = 30;
            break;
        case 2:
            if (annee % 4 == 0) // si année bissextile
            {  nb = 29;
            } else {
                nb = 28;
            }
        default:
            nb = 0;
            break;
    }
    return nb;
}

//méthode qui vérifie si la date est valide
public boolean dateValide() {
    if (jour > 0 && jour <= nbreJours() && moi > 0 && moi <= 12 && annee > 0) {
        return true;
    }
    return false;
}

// méthode qui retourne la date du lendemain
public Date lendemain() {
    Date d = new Date(jour, moi, annee);
    if (d.jour < nbreJours()) {
        d.jour++; // si le moi en cours n'est pas fini, incrémenter le jour
    } else {
        d.jour = 1; // sinon , un nouveau moi commence
```

```
            if (d.moi > 12) {
                d.moi++;
            } else {
                d.moi = 1;
                d.annee++;
            }
        }
        return d;
    }

    public void afficherDate() {
        System.out.println("Date: " + "/" + jour + "/" + moi + "/" + annee);

    }
} // Fin TestDate


2)
public class TestDate {

    public static void main(String[] args) {
        Date d = new Date(9, 8, 1985);
        d.afficherDate();
        if (!d.dateValide()) {
            System.out.println("Date invalide!");
        } else {

            System.out.println("Nombre de jours du moi: " + d.nbreJours());
            Date lendemain = d.lendemain();
            lendemain.afficherDate();
        }

    }
}
```

Exercice 2

On désire représenter les nombres complexes sous la forme a+ib où a est la partie réelle, b la partie imaginaire (a et b étant deux entiers positifs) et i le nombre dont le carré vaut -1. On voudra pouvoir additionner et de multiplier des complexes, calculer le conjugué et le module ... On se propose de créer alors la classe *Complexe*.

1) Déterminer les deux attributs de la classe Complexe.

2) Donner les constructeurs suivants:
   - un constructeur par défaut
   - un constructeur qui initialise les deux attributs par les valeurs passées en paramètre
   - un constructeur permettant de construire un complexe ne possédant pas de partie imaginaire.

   Qu'appelle-t-on l'opération de programmer ces trois constructeurs?
3) Donner la méthode permettant de tester l'égalité de deux complexes. (on va tester l'égalité des deux parties réelles et des deux parties imaginaires)
4) Donner la méthode *"addition"* permettant d'additionner deux complexes.
5) Donner la méthode *"addition"* permettant d'additionner un entier au complexe en cours.

   Peut-on appeler cette méthode *"addition"* ? De quoi sont caractérisées les deux méthodes des questions 4 et 5?
6) Donner la méthode qui retourne le conjugué.
7) Donner la méthode qui calcule le module du complexe.
8) Donner la méthode qui multiplie deux complexes.
9) Créer la classe TestComplexe qui teste la méthode de calcul du module.

Solution

```
public class Complexe {
   // Les attributs privés
  private  int a; //Partie reelle
  private  int b; //Partie imaginaire

   // Les constructeurs
   public Complexe() {
   }

   public Complexe(int r, int i) {
      a = r;
      b = i;
   }

   public Complexe(int r) {
      a = r;
      b = 0;
   }

   // Les méthodes
```

```
    // tester l'égalité
    boolean testEgaliter(Complexe c2) {
        if ((this.a == c2.a) && (this.b == c2.b)) {
            return true;
        }
        return false;
    }

    /*additionner deux complexes: cette méthode retourne un nouveau complexe
representant l'addition du complexe courant
             * au complexe passe en parametre. */
    Complexe addition(Complexe c2) {
        Complexe res = new Complexe();
        res.a = a + c2.a;
        res.b = b + c2.b;
        return res;

    }

    /** Ajoute au complexe courant un entier. Cette methode ne retourne
     * pas de resultat car elle modifie le complexe courant */
    void additionEntier(int r) {
        a = a + r;
    }

//Calcul du conjugué du complexe
    Complexe calculerConjugé() {
        return new Complexe(a, -b);
    }

//     Calcul le module du complexe
    double calculerModule() {
        double module = Math.sqrt(a * a - b * b);
        return module;
    }

// multiplier deux complexes
    public Complexe multiplication(Complexe z) {
        return new Complexe(a * z.a - b * z.b, a * z.b + z.a * b);
    }
```

```
}

class TestComplexe {

   public static void main(String[] args) {
      Complexe c = new Complexe(3,2);
      double module = c.calculerModule();
      System.out.println("Module: "+ module);

   }
}
```

Chapitre 3. Tableaux et chaînes de caractères

Objectifs

Ce chapitre s'intéresse à la manipulation des tableaux et des chaines de caractères.

Pré-requis

- Notion de classe et d'objet.
- Algorithmique et structures de données

Eléments de contenu

1. Les tableaux
    - 1.1. Déclaration
    - 1.2. Création
    - 1.3. Initialisation
    - 1.4. Parcours d'un tableau
    - 1.5. Tableaux à deux dimensions
2. Les chaînes de caractères

1. Les tableaux

En java le type tableau est assimilable à une classe, un tableau est un objet référencé.

- Un objet tableau ne référence pas l'objet lui-même mais uniquement l'adresse mémoire a qui il fait référence. Déclarer un tableau en java revient à réserver de la place mémoire pour l'adresse de ce tableau.
- Pour créer un tableau il faut lui allouer l'espace mémoire nécessaire après l'avoir déclaré.

Si les éléments d'un tableau sont de type simple, Java stocke directement les valeurs dans les éléments du tableau. S'il s'agit d'un tableau d'objets, les éléments du tableau contiennent toujours des références (pointeurs) aux objets et non pas les objets eux mêmes.

1.1. Déclaration

La déclaration d'un tableau à une dimension suit cette syntaxe:

<type> <nomTableau>**[ ]**;

ou bien

<type> **[ ]**<nomTableau>;

➔ On peut placer les crochets après ou avant le nom du tableau dans la déclaration.

Exemples:

```
int ages[]; // tableau entiers
char[] lettres; //tableau de caractères
float tab[100]; //instruction illégale car il n'est pas possible de définir un
                //tableau de taille fixe à la déclaration.
```

1.2. Création

La déclaration ne fait que réserver l'espace mémoire allant contenir l'adresse du tableau. Cette place mémoire est identifiée par le nom du tableau. L'allocation de la place mémoire se fait par le mot clé new :

<nomTableau> = new <Type>[<dimension>];

Exemples:

```
int ages[] = new int[8];
char[] lettres = new char[28];
```

1.3. Initialisation

Lorsqu'un tableau est créé, chacun de ses éléments est initialisé par défaut. Cette initialisation se fait :

- à 0 si les éléments sont des entiers ou des réels
- à false s'ils sont des booléens
- à null sinon.

On peut aussi initialiser le tableau lors de sa création par des valeurs particulières.

Exemple:

```
int tab[ ]={2, 4, 6};
```

est équivalent à faire

```
int tab[ ] =new int[3]; tab[0]=2; tab[1]=4; tab[2]=6;
```

1.4. Parcours d'un tableau

La longueur d'un tableau peut être obtenue à partir de la variable length (Elle retourne le nombre d'éléments du tableau). La syntaxe est la suivante:

<nomTableau>.length

Le premier élément d'un tableau est d'indice 0, le dernier élément est d'indice n-1 avec n le nombre d'éléments du tableau(tab.lenght -1).

L'accès à un éléments du tableau se fait en mettant l'indice de cet élément entre crochets. Si on essaie d'accédé à un élément qui ce trouve en dehors des bornes du tableau une erreur (de type *java.lang.arrayIndexOutOfBoundsException*) est générée.

Exercice:

1) Créez et initialisez un tableau contenant 3 notes d'un étudiant.
   - Afficher la somme des notes en utilisant la boucle for.
   - Afficher la moyenne de ces 3 notes.
2) Affichez la somme des notes paires.

Solution:

1)
```
float tab[] = {12, 10, 13};
float s = 0;
for (int i = 0; i < tab.length; i++) {
   s += tab[i];
}
System.out.println("La somme est: " + s);
System.out.println("La moyenne est: " + s/tab.length);
```

2)
```
float tab[] = {12, 10, 13};
float sPaires = 0;
for (int i = 0; i < tab.length; i++) {
   if (tab[i]%2 ==0)
   sPaires += tab[i];
```

```
        }
        System.out.println("La somme est: " + sPaires);
```

Exécution:
1)La somme est: 35.0
La moyenne est: 11.666667

2) La somme est: 22.0

1.5. Tableaux à deux dimensions

En java, pour déclarer un tableau à plusieurs dimensions, on déclare un tableau de tableaux. Un tableau à 2 dimensions est donc un tableau dont chaque composante est formée par un tableau.

Il est à noter que la taille des tableaux de la seconde dimension peut ne pas être identique pour chaque occurrence.

Exemple:
int[][] notes=new int[3][]; // 3 sous tableaux

notes[0]=new int[4]; notes [0][0]=10; notes [0][1]=11; notes [0][2]=12; notes [0][3]=13;	10	11	12	13
notes[1]=new int[3]; notes [1][0]=5; notes [1][1]=6 ; notes [1][2]=7;	5	6	7	
notes[2]=new int[1]; notes [2][0]=20	20			

Si le tableau à deux dimensions est rectangulaire, on peut le créer comme suit :

<type>[ ][ ]<nomtableau>=new<type>[N1][N2];

- N1: nombre des sous-tableaux.
- N2: nombre des éléments dans chaque sous-tableau.

Exemple:

float tab [ ] [] = new float[3][3];

Crée le tableau ayant la forme suivante:

x	x	x
x	x	x
x	x	x

- Le nombre des sous-tableaux peut être obtenu en utilisant la variable length.

Exemple:

```
int[][] tab={{1,2,9},{12,8},{11,15,9,10}};
tab.length➔3
tab[1].length➔2
```

Exercice:

Créer un tableau qui a la forme ci-dessous puis afficher ses valeurs:

Sous tableau1:	10	20
Sous tableau2:	30	

Solution:

```
        // Déclaration et création
        int[][] tab = new int[2][];
        tab[0] = new int[2];
        tab[1] = new int[1];
    // Remplir par des valeurs
      tab[0][0] = 10;
        tab[0][1] = 20;
        tab[1][0] = 30;
    // Afficher les valeurs
        for (int i = 0; i < tab.length; i++) {
          System.out.println("Sous tabelau n° "+ i);
            for (int j = 0; j < tab[i].length; j++) {
            System.out.println( tab[i][j]);
          }

  }
```

L'exécution donne:

```
Sous tableau n° 0
10
20
Sous tableau n° 1
30
```

2. Les chaînes de caractères

Le type String sert à représenter des chaines de caractères en Java. Il ne s'agit pas d'un tableau de caractères comme en C. String est une classe qui est fourni avec l'API Java, elle décrit des objets qui contiennent une chaîne de caractère **constante.** Voici un exemple de déclaration est initialisation d'une chaine:

String s=‘‘Bonjour ISET";

Les opérateurs + et += servent pour la concaténation des chaînes de caractères. Java permet un traitement très riche des chaînes de caractères grâce à un ensemble méthodes prédéfinies. Les plus utilisés sont décrits dans le tableau qui suit.

méthode de la classe String	**Utilité**	**Exemple pour String s=‘‘Bonjour RSI";**
int length()	retourne le nombre de caractères de la chaîne.	System.out.println(s.length()); ➔ 11
boolean isEmpty()	true si la chanine est vide, false sinon	System.out.println(s.isEmpty()); ➔ false
char charAt(int i)	retourne la caractère qui se trouve à la position i.	System.out.println(s.charAt(2)); ➔ n
boolean equals(String s)	effectue la comparaison de 2 chaînes (elles sont égales si elles contiennent la même suite de caractères)	if(s.**equals**("Bonjour RSI")) {}→ true if (s.equals("BONJOuR RSI")) {} → false
boolean equalsIgnoreCase(String s)	Comme la méthode "equals" mais elle ignore la case. (majuscule ou minuscule)	if (s.equals("BONJOuR RSI")) {} → devient true
String toLowerCase ()	Retourne une chaine avec les caractères en minuscule.	System.out.println(s.toLowerCase()); ➔ bonjour rsi
String toUpperCase ()	Retourne une chaine avec les caractères en majuscule.	System.out.println(s.toUpperCase()); ➔ BONJOUR RSI
int indexOf(String ch, int i)	retourne la position de la chaine ch à partir de la position i.	System.out.println(s.indexOf("jour", 2)); → 3
String replace(char oldChar, char newChar)	Remplace un caractère par un autre.	System.out.println(s.replace('o', 'i')); →Binjiur RSI
String substring(int i, int j)	retourne une chaîne extraite de la chaîne sur laquelle est appliquée la méthode en partant de la position i à la position j.	System.out.println(s.substring(1, 4)); →onj

boolean endsWith (String suffix)	Teste si la chaine fini par la chaine suffixe. (il existe aussi la méthode startsWith)	System.out.println(s.endsWith("RSI")); ➔ true

Ces méthodes facilitent les opérations traditionnelles que le développeur Java pourrait avoir besoin, ce qui permet un gain de temps et d'effort. Ceci met l'accent sur un des avantages qui ont fait le succès du langage Java: une API riche.

Remarque: La classe StringBuffer permet de représenter une chaine de taille variable.

3. Série d'exercices

Exercice 1

Soit la classe suivante:

```
class Personne {
   //les attributs
   private String nom;
   private String prenom;
   private int age;

   // constructeur
   public Personne(String nom, String prenom, int age) {
     this nom=nom;
     this.age = age;
     this.prenom = prenom;      }

   //Méthode qui retourne l'age
   public int getAge() {       return age;    }
} // Fin classe
```

Donner la méthode main permettant de:

- créer puis remplir un tableau de 3 personnes
- Le remplir
- afficher la somme de leurs âges.

Solution

```
public static void main(String[] args) {
      Personne tab[] = new Personne[3];
      tab[0] = new Personne("Mohamed", "Amine", 20);
      tab[1] = new Personne("Manel", "Mohamed", 15);
      tab[2] = new Personne("Ali", "Saleh", 36);
```

```
        int somme = 0;
        for (int i = 0; i < tab.length; i++) {
            somme = somme + tab[i].getAge();
        }
        System.out.println("La somme des ages est : " + somme);
    }
```

Exercice 2

1. Créer la classe *ChainePlindrome* contenant un attribut str de type String.
2. Doter la classe d'un constructeur permettant d'initialiser la chaine par la chaine passée en paramètre
3. Donner la méthode *inverser* qui retourne l'inverse de str
4. Donner la méthode *estPalindrome* qui teste si une chaine est palindrome (se lit de la même manière de droite à gauche et de gauche à droite)
5. Donner la classe *TestChainePalindrome* permettant de tester la classe précédente.

Solution

```
public class ChainePlindrome {
    // attributs privés
    private String str;

    // constructeur
    public ChainePlindrome(String str) {
        this.str = str;
    }
// La méthode inverser qui inverse la chaine
    public String inverser (){
        String res="";
        for (int i  =str.length()-1;i>=0; i--) {
            res+=str.charAt(i);
        }
        return res;
    }

// la méthode estPalindrome qui teste si une chaine est palindrome
    public boolean  estPalindrome () {
    if (str.equals(this.inverser()))
        return true;
    return false;
        }
}
```

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

   ChainePlindrome ch= new ChainePlindrome("aziza");
   if (ch.estPalindrome())
      System.out.println("La chaine est palindrome");
   else
       System.out.println("La chaine n'est pas palindrome");
 }

}
```

Chapitre 4. Association et agrégation entre les classes

Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

· Etre capable de comprendre les relations d'association ou d'agrégation entre les classes.

- Créer des classes java avec des relations d'association ou agrégation entre les elles.

Pré-requis

- Classes et objets en Java
- Manipulation des tableaux.

Eléments de contenu

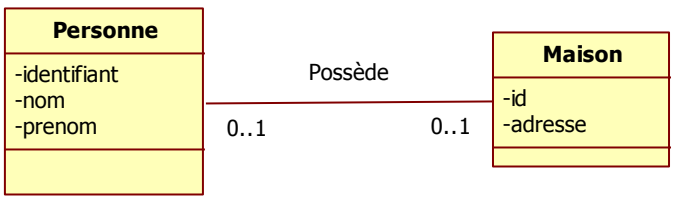
1. Association entre classes

Une application réalisée selon l'approche objet est constituée d'objets qui collaborent entre eux pour exécuter des traitements. Ces objets ne sont donc pas indépendants, ils ont des relations entre eux. Comme les objets ne sont que des instances de classes, il en résulte que les classes elles-mêmes sont reliées.

Une association est une relation simple entre deux classes. Elle peut avoir différents types de cardinalité et on peut avoir plusieurs types de relations entre des objets. En pratique, les relations sont sous la forme de **références vers un ou plusieurs autres objets** dans une variable d'instance.

1.1. Relation un à un

Dans l'exemple suivant, il s'agit d'une relation directe entre deux classes: La classe personne et la classe Maison. Ci-dessous une présentation en diagramme de classes d'UML (Unified Modeling Language) de cet exemple.



Il est à noter qu'on n'as pas nécessairement besoin d'une classe pour construire l'autre: on peut créer une personne sans maison et une maison sans propriétaire. Voici la traduction en Java :

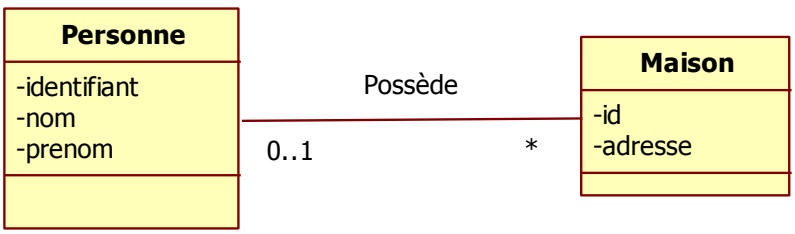
```
class Maison{
private int id;
private String adresse;
private Personne proprietaire; // on fait une référence à la personne
}

class Personne {
private int identifiant;
private  String nom, Prénom;
private Maison maison;  // on fait une référence à la maison
}
```

Remarque: La navigabilité restreinte

Il arrive qu'une classe A ait comme attribut une ou plusieurs références à une autre classe B, mais que la classe B ne possède aucune référence à la classe A. On dit dans ce cas que la navigabilité est restreinte de A vers B.

1.2. Relation un à plusieurs

Une personne peut posséder plusieurs maisons. Dans ce cas, on aura besoin d'une référence à un tableau d'objets.



Exercice

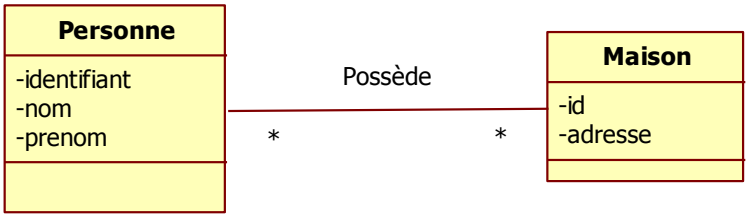
Donner la traduction en Java de l'exemple.

Solution

```
class Personne {
private int identifiant;
private String nom, Prénom;
private Maison[ ] maisons; // référence à un tableau d'objets

}
class Maison{
private int id;
private String adresse;
private Personne proprietaire;
}
```

1.3. Relation plusieurs à plusieurs

Une personne possède plusieurs maisons, une maison est possédée par plusieurs personnes: on doit utiliser deux tableaux d'objets.



Exercice

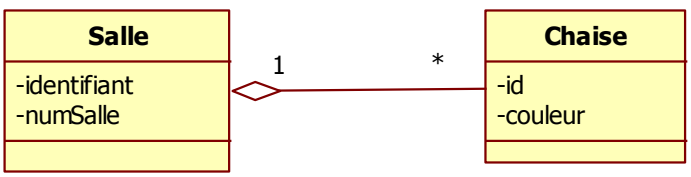
Donner la traduction en Java de l'exemple.

Solution

```
class Personne {
private int identifiant;
private String nom, Prénom;
private Maison[ ] maison;

}
class Maison{
private int id;
private String adresse;
private Personne [ ] proprietaires;
}
```

2. Agrégation entre classes

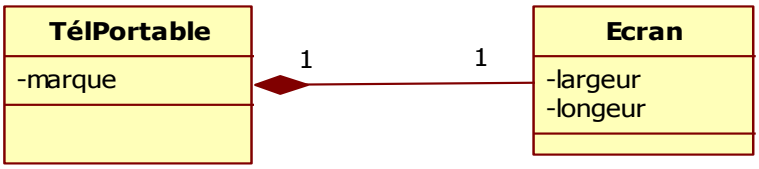
L'agrégation est une association particulière où une classe est une partie d'une autre classe. C'est est un lien entre un agrégat est une partie. Par exemple, une salle de réunion contient plusieurs chaises. Une chaise n'a pas besoin des salles de réunion pour exister. Un objet chaise a donc un sens en lui-même. Ci-dessous une représentation UML de l'exemple.



Une composition est un cas particulier de l'agrégation qui relie entre un composé et un composant, c'est une agrégation **forte**. L'objet du composant n'a pas un sens en lui-même, sa durée de vie dépend de celle du composé. On le contrôle à partir du composé, ainsi

l'instanciation se fait dans la classe du composé. C'est donc dans le constructeur de la classe composée que sont instanciés le ou les objets composants. Alors que dans le cadre d'une association simple, les objets en relations sont instanciés indépendamment.

Exemple: Un Téléphone portable est composé d'un écran. (La composition est modélisée par un losange noir en UML)



Traduction en Java:

```
class TélPortable {
  private  String marque;
  private     Ecran ecran;

       public TélPortable(String marque) {
          this.marque = marque;
          ecran = new Ecran ();
       }
   }

   class Ecran {
    private   float longueur, largeur;
   }
```

3. Problème

On s'intéresse dans ce problème à représenter une étagère de livres.

1. Créez une classe Livre ne possédant que des attributs privés sachant qu'un Livre est caractérisé par son titre, son auteur, le nombre de page qui le constitue et le prix.

- ***a.*** Doter cette classe d'un constructeur qui initialise les attributs.
- ***b.*** Doter cette classe des méthodes pour récupérer ou accéder aux différents attributs.
- ***c.*** Donner la méthode toString afin d'avoir une représentation d'un objet de la classe Livre de cette façon : « livre de titre **Introduction à Java** écrit par **Mohamed Amin** ayant **300** pages et de prix **100.5**Dinars ».

2. Créez une classe Etagere pour représenter une étagère qui peut contenir un certain nombre de livres (fixe pour chaque étagère). Vous utiliserez pour cela un tableau.

- ***a.*** Créez un constructeur de la classe Etagère.

Ajoutez des méthodes permettant de:

b. Donner le nombre de livres que peut contenir l'étagère, et le nombre de livres qu'elle contient.

c. Ajouter des livres ("boolean ajouter(Livre)"). Vous ajouterez les livres "à la fin" de l'étagère. Il devra être impossible d'ajouter des livres dans une étagère pleine.

d. Récupérer un livre dont on donne la position sur l'étagère (le livre reste sur l'étagère, on récupère simplement une référence sur le livre). La position du premier livre d'une étagère devra être 1 (et pas 0, bien que le livre soit rangé dans la première position du tableau, qui est d'indice 0).

e. Chercher sur l'étagère un livre repéré par son titre et son auteur. La méthode *"chercher"* renverra la position du livre dans l'étagère (ou 0 si le livre n'y est pas). S'il y a plusieurs livres avec le même titre et le même auteur, la méthode renvoie celui qui a le plus petit indice.

f. Chercher sur l'étagère un livre comme dans la question précedente mais la méthode renvoie un tableau de toutes les positions du livre. On aimerait appeler cette méthode "*chercher*" ; est-ce possible ?

g. chercher tous les livres d'un auteur. Cette fois-ci, la méthode renvoie un tableau de livres.

h. Dans la méthode main(), vous créerez des livres, 2 étagères et ajouterez les livres dans les étagères. Vous chercherez un des livres dans une étagère ; s'il y est, vous l'affichez.

Solution

```
1)
public class Livre {
// attributs privés
  private String titre, auteur;
  private int nbPages;
  private double prix;
// constructeur
    public Livre(String titre, String auteur, int nbPages, double prix) {
       this.titre = titre;
       this.auteur = auteur;
       this.nbPages = nbPages;
       this.prix = prix;
    }
// les accesseurs
  public String getAuteur() {
    return auteur;  }
  public String getTitre() {
    return titre;  }
  public int getNbPages() {
    return nbPages;  }
```

```
        public double getPrix() {
          return prix;  }

        public void setAuteur(String unAuteur) {
          auteur = unAuteur;  }
        public void setTitre(String unTitre) {
          titre = unTitre;  }
        public void setNbPages(int n) {
          if (n > 0) {
            nbPages = n;    }              // le nombre de pages doit être positif
          else {
            System.err.println("Erreur : nombre de pages négatif !");    }
        }
        public void setPrix(double unPrix) {
          if (unPrix >= 0)                 // le prix doit être positif
            prix = unPrix;
          }
          else {
            System.err.println("Erreur : prix négatif !");
          }
        }
      // méthode  pour l'affichage
        public String toString() {
          return "livre de titre"+titre+" écrit par"+ auteur+" ayant"+nbPages+" pages et de
prix"+prix+" Dinars ";
  }
```

2).

```
public class Etagere {
   // Les attributs privés
   private Livre[] livres;
   private int nbLivres = 0; // nbre de livres qu'il y a dans l'étagère

   // Les constructeurs
   public Etagere(int nb) {
      livres = new Livre[nb];
   }
// retourner le nombre de livres
   public int getNbLivres() {
      return nbLivres;
```

```
    }

// retourner le nombre de livres maximum
    public int getNbLivresMax() {
        // Taille maximale du tabelaux
        return livres.length;
    }

    // ajouter un livre à la fin de l'étagère
    public void ajouter(Livre l) {
        if (nbLivres < livres.length) {
            livres[nbLivres++] = l;
        } else {
            System.err.println("Etagere pleine");
        }
    }

    // méthode qui renvoie le livre placé sur l'étagère dans la position indiquée.
    public Livre getLivre(int position) {
        if (position > 0 && position <= nbLivres) {
            return livres[position - 1];
        } else {
            return null;
        }
    }

    //Méthode qui cherche la position du premier livre de l'étagère avec un auteur et un
//titre donné.
    public int chercher(String titre, String auteur) {
        for (int i = 0; i < nbLivres; i++) {
            if (livres[i].getTitre().equals(titre) && livres[i].getAuteur().equals(auteur)) {
                return i + 1;
            }
        }
        return 0;
    }

    // Méthode qui cherche les positions de tous les livres de l'étagère avec un auteur et
//un titre donné.
    public int[] chercherLivres(String titre, String auteur) {
        int[] res = new int[nbLivres];
```

```
        int nbLivresTrouves = 0;
        for (int i = 0; i < nbLivres; i++) {
            if (livres[i].getTitre().equals(titre) && livres[i].getAuteur().equals(auteur)) {
                res[nbLivresTrouves++] = i + 1;
            }
        }

        return res;
    }

    //Méthode qui cherche tous les livres d'un auteur sur l'étagère.
    public Livre[] chercherAuteur(String auteur) {
        Livre[] res = new Livre[nbLivres];
        int nbLivresTrouves = 0;
        for (int i = 0; i < nbLivres; i++) {
            if (livres[i].getAuteur().equals(auteur)) {
                res[nbLivresTrouves++] = livres[i];
            }
        }

        return res;
    }
// method main pour le test
    public static void main(String[] args) {
        Livre l1 = new Livre("a1", "t1", 200, 200);
        Livre l2 = new Livre("a2", "t2", 100, 100);
        Livre l3 = new Livre("a3", "t3", 399, 160.8);
        Livre l4 = new Livre("a1", "t4", 800, 900);
        Livre l5 = new Livre("a1", "t1", 66, 88);

        Etagere etagere1 = new Etagere(4);
        Etagere etagere2 = new Etagere(3);

        etagere1.ajouter(l1);
        etagere1.ajouter(l2);
        etagere1.ajouter(l3);

        etagere2.ajouter(l1);
        etagere2.ajouter(l4);
        etagere2.ajouter(l5);
```

```
        // Recherche par auteur et titre dans la première étagère
        String auteur = "a1", titre = "t1";
        int position = etagere1.chercher(titre, auteur);
        if (position != 0) {
            System.out.println(etagere1.getLivre(position));
        } else {
            System.out.println("Livre d'auteur " + auteur + " et de titre " + titre + " pas trouvé
dans étagère 1");
        }

    }
}
```

Chapitre 5. L'héritage et encapsulation

Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Comprendre le principe d'héritage entre les classes et son l'utilité.
- Savoir écrire un programme en utilisant: la redéfinition des méthodes, et l'utilisation des membres de la classe de base dans la classe dérivée.
- Comprendre les classes et les méthodes finales.
- Savoir utiliser les paquetages en java et comprendre leurs utilités.
- Comprendre le principe d'encapsulation et les différents nivaux de visibilité.

Pré-requis

- Classes et objets en Java

Eléments de contenu

1. Utilité et mise en œuvre de l'héritage
2. Constructeur de la sous classe
3. La redéfinition des champs
4. La redéfinition des méthodes
5. Interdire l'héritage
6. La classe Object
7. Les paquetages
8. L'encapsulation et la visibilité des membres

1. Utilité et mise en œuvre

Soit la classe suivante:

```
class Personne
{ String nom;
  int age;

 Personne(String n, int a)
 {nom  = n; age=a;}

 public void afficherNom()
 { System.out.println("le nom de la personne est: " +nom);}
}
```

Problématique: Supposons qu'on veut définir une deuxième classe Etudiant qui présente un étudiant par son nom, son âge et son numéro de carte étudiant. On a besoin de deux méthodes, la première affiche le nom de l'étudiant et la deuxième affiche son numéro de carte étudiant.

On a deux solutions:

1) La première solution est de réécrire entièrement cette classe.

```
class Etudiant
{ String nom;
 int age;
   int numCarte;

public void afficherNom()
 { System.out.println("Le nom de la personne est: " +nom);}

public void afficherNumCarte()
 { System.out.println("Le numéro de carte étudiant est :"  + numCarte);}
}
```

Cette solution s'oppose à un avantage important de l'O.O qui est **la réutilisation du code** (réutiliser au maximum le code existant pour gagner en temps et en effort).

2) Deuxième solution: Puisqu'on a déjà la classe Personne, on a un moyen plus simple de définir la classe Etudiant qui est l'héritage**.** On dit que la classe Etudiant **hérite** les attributs et les méthodes de la classe de base Personne. On utilise pour ceci le mot clé extends.

```
class Etudiant extends Personne{
   int numCarte;

  public void afficherNumCarte()
  { System.out.println("Le numéro de carte étudiant est :"  + numCarte);}
}
```

Dans la classe Etudiant, on écrit donc seulement que la partie qui diffère de la classe Personne, **la partie commune sera héritée automatiquement**. L'héritage est donc un mécanisme qui facilite la réutilisation du code (donc gain de temps et d'effort). Il définit une relation entre deux classes :

- une classe mère, super classe ou classe de base.
- et une classe fille, sous-classe ou classe dérivée qui hérite de sa classe mère.

Il est à noter que tout objet est vu comme instance de sa classe et aussi comme instance de toutes les classes dérivées.

Exemple:

```
public static void main(String[] args) {
  Personne p = new Personne( …);
  Etudiant e = new Etudiant(…);
  p=e; // on peut affecter un étudiant à une personne
  e=p; // faux, on ne peut pas affecter une personne à un étudiant
  }
```

2. Constructeur de la sous classe

En Java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

Pour invoquer le constructeur de la classe de base, on fera appel à l'instruction **super(...).** Un constructeur d'une classe dérivée se compose généralement deux parties :

- ✓ celle concernant les champs de la classe de base
- ✓ et celle concernant les champs propres de la classe dérivée.

```
class Etudiant extends Personne
{  private int numCarte;

// le constructeur
  Etudiant (String a, int b,  int c){
super(a,b);    //appel au constructeur de la classe de Base.
 numCarteEtudiant =c; //initialisation des champ de la classe dérivée
  }
```

```
    …
}
```

L'invocation de super(...) doit être la première instruction du constructeur de la classe dérivée.

Remarque: Cas du constructeur par défaut

Si le constructeur de la classe dérivée n'invoque pas le constructeur de la classe de base explicitement avec l'instruction super(...), Java fait quand même appel au constructeur, sans argument, de la classe de base : super().

- **Enchainement des constructeurs**

Pour tout objet crée, le constructeur de la classe de base est invoqué qui lui a son tour invoque le constructeur de sa classe de base et ainsi de suite. Il existe donc un enchaînement d'invocation de constructeurs. Cette cascade d'appels aux constructeurs s'arrête dès que l'on atteint le constructeur de la classe Object.

La classe Object est la mère de toutes les classes: toute classe est dérivée directement ou indirectement de la classe Object. Ainsi, lors de la création d'un objet, le premier constructeur invoqué est celui de la classe Object suivi des autres constructeurs dans l'ordre de la hiérarchie de dérivation des classes.

3. La redéfinition des champs

Les champs déclarés dans la classe dérivée sont toujours des champs supplémentaires. Si l'on définit un champ dans la sous classe ayant le même nom qu'un champ de la classe de base, il existera deux champs de même noms. Le nom de champ désignera toujours le champ déclaré dans la classe dérivée. Pour avoir accès au champ de la classe de base, il faudra changer le type de la référence pointant sur l'objet ou en utilisant le mot clé super.

Exemple

```
class A
{
public int i ;
...
}
```

```
class B extends A {
public int i ;
...
public void uneMethode() {
        i = 0 ; // i est le champ défini dans la classe B
        this.i = 0 ; // i est le champ défini dans la classe B
        super.i = 1 ; // i est le champ défini dans la classe A
        ( (A) this ).i = 1 // i est le champ défini dans la classe A
… } }
```


La technique de redéfinition des champs peut s'appliquer en cascade de la manière suivante:

```
class C extends B {
   public int i ;
   ...
   public void uneMethode()
   {
        i = 0 ; this.i = 0 ; // i est le champ défini dans la classe C
                super.i = 1 ; ( (B) this ).i = 1; // i est le champ défini dans la classe B
        ( (A) this ).i = 1 // i est le champ défini dans la classe A
   } }
```

Par contre, l'instruction suivante est incorrecte :

super.super.i = 1 ; // Incorrect syntaxiquement !

Tout comme l'utilisation du mot clé this, le mot clé super ne peut être utilisé dans les méthodes static.

4. La redéfinition des méthodes:

Il est possible de dériver une classe pour uniquement modifier les méthodes de la classe de base.

Exemple: On veut afficher le nom d'un étudiant en spécifiant qu'il s'agit d'un étudiant pas d'une personne en général. On va donc redéfinir la méthode afficherNom de la classe personne: La redéfinition d'une méthode consiste à fournir une implantation différente de la méthode de même signature fournie par la classe de base.

```
class Etudiant extends Personne
{  private int numCarte;
….
// la méthode redéfinie
  public void afficherNom()
  { System.out.println("le nom de l'étudiant est: " +nom);}

}
```

On dit que la méthode afficherNom est **polymorphe**: elle a différentes formes puisqu'elle est déclarée dans la classe de base puis redéfinie avec une autre forme dans la classe dérivée.

Dans la classe fille, pour avoir accès à une méthode dans sa version avant redéfinition, on utilise le mot clé *super* comme pour le cas des attributs redéfinis .

Exercice:

Redéfinir la méthode AfficherNom de la classe personne pour qu'elle affiche:

Le nom de la personne est : xxxx

C'est un étudiant.

Solution:

La première partie de l'affichage est celle de la classe mère. → On peut donc appeler la méthode afficherNom de la classe mère dans la méthode redéfinie.

```
class Etudiant extends Personne
{  private int numCarte;
….
// la méthode redéfinie
  public void afficherNom()
  { super. afficherNom();
System.out.println("C'est un 'étudiant");}
  }
```

5. Interdire l'héritage

Pour interdire qu'une classe soit héritée, on utilise le mot réservé *final*: aucune classe ne peut donc hériter de cette classe.

Exemple

```
final class Personne
{…}

class Etudiant extends Personne
{…}  → C'est interdit car Personne est une classe finale.
```

Pour interdire qu'une méthode soit redéfinit dans une sous classe, on utilise aussi le mot *final*.

Exemple

```
class Personne{
…
  public final void afficherNom(){}
}

class Etudiant  extends Personne{
```

```
  public void afficherNom(){..}// → Interdit car  afficherNom() est final dans Personne.

}
```

6. La classe Object

C'est la classe de base de toutes les classes en Java. Les méthodes définit dans cette classe peuvent alors être utilisées ou redéfinies. Deux méthodes sont généralement redéfinies:

- **public String toString ()** : utilisée pour donner une représentation textuelle d'un objet (si elle n'est pas redéfinie, elle retourne entre autres le nom de la classe). Ainsi on pourra utiliser la méthode System.out.print en lui donnant comme paramètre le nom de l'objet.

- **public boolean equals (Object obj)** utilisée pour tester l'égalité sémantique entre deux objets (si elle n'est pas redéfinie, equals compare la référence des deux objets: retourne true si les deux références désignent le même objet).

Application

Redéfinir la méthode toString dans la classe Personne .

Solution

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

public String toString() {
      return "La personne s'appelle " + prenom+ " "+ nom + "il a "+ age+ " ans.";
   }
```

7. Les paquetages

7.1. Utilité des paquetages

Un paquetage (connu aussi par son nom anglais ***package***) est un ensemble de classes, d'interfaces et d'autres packages regroupés sous un nom. Ils correspondent, en quelque sorte, au concept de librairies adapté au langage Java. Au début de chaque fichier source destiné à faire partie d'un package nommé monpackage, on doit préciser la directive suivante (elle doit être la première instruction):

package nompaquet ;

Cette déclaration précise que toutes les classes et interfaces définies dans ce fichier font partie du package monpackage. Elle doit apparaître avant toute déclaration de classes et interfaces.

Un package est souvent utilisé pour regrouper des classes voisines ou qui couvrent un même domaine.

Exemple:

package matriel.info		package personnes	
Fichier source 1	Fichier source 2	Fichier source 3	Fichier source 4

package matrielInfo;
class Clavier{
…….
}
	package matrielInfo;
	class Routeur{
	…….
	}
		package personnes;
		class Etudiant {
		…….
		}
			package personnes;
			class Enseignant {
			…….
			}

Plusieurs IDE permettent d'afficher les fichiers sources sous forme d'arborescence comme décrit dans la figure suivante.

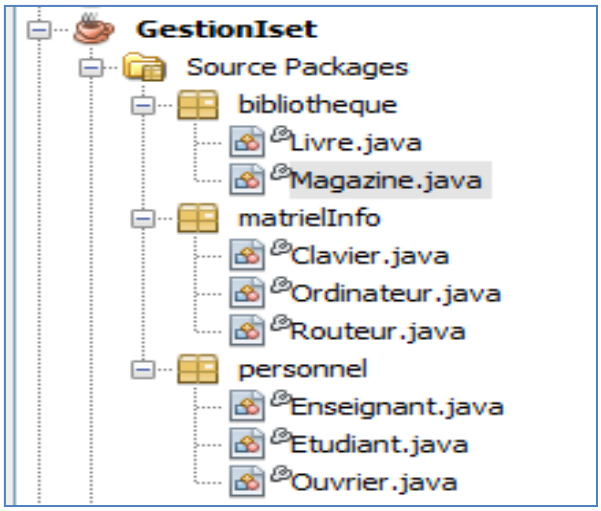


Figure 1: Exemple d'une arborescence de classes représentée par l'IDE NetBeans

Si

Les packages servent a

- Eliminer la confusion entre des classes de même nom (on peut définir deux classes de même nom à condition qu'elles appartiennent à des paquetages différents.)
- Structurent l'ensemble des classes selon une arborescence
- Faciliter la recherche de l'emplacement des classes.
- Permettent de gérer le nivaux de visibilité selon le fait que les classes appartiennent au même package ou non.

7.2. Importer des paquetages

Lorsqu'on veut utiliser une classe d'un package, le moyen le plus direct est de nommer cette classes par son nom absolu (fully qualified name) :

matrielInfo.Clavier c;

Cette manière de faire est fastidieuse pour un développeur. Pour éviter ce problème, Java propose l'utilisation de la directive ***import*** surtout quand on veut utiliser une classe dans une autre classe de package différent. Il s'agit de prévenir le compilateur qu'on risque d'utiliser des noms simplifiés pour nos classes, il doit donc préfixer tout seul les noms de classes quand c'est nécessaire.

Exemple: Utilisation la classe Clavier dans la classe Etudiant

```
package personnel;
import matrielInfo.Clavier;
// ou bien import matrielInfo.* ; pour importer toutes les classes de ce package,
// y compris la classe Clavier

public class Etudiant {
   void uneMethode(){
   Clavier c = new Clavier();
   //équivalent à matrielInfo.Clavier c= new matrielInfo.Clavier();
Enseignant e = new Enseignant (); // pas besoin d'import car Etudiant
                                  //et Enseignant sont dans le même package
 …..  }}
```

Remarques:

- Java importe automatiquement le package *java.lang* qui permet d'utiliser les classes comme System.
- Si aucun paquetage n'est déclaré, la classe appartient à un paquetage appelé: paquetage par défaut.

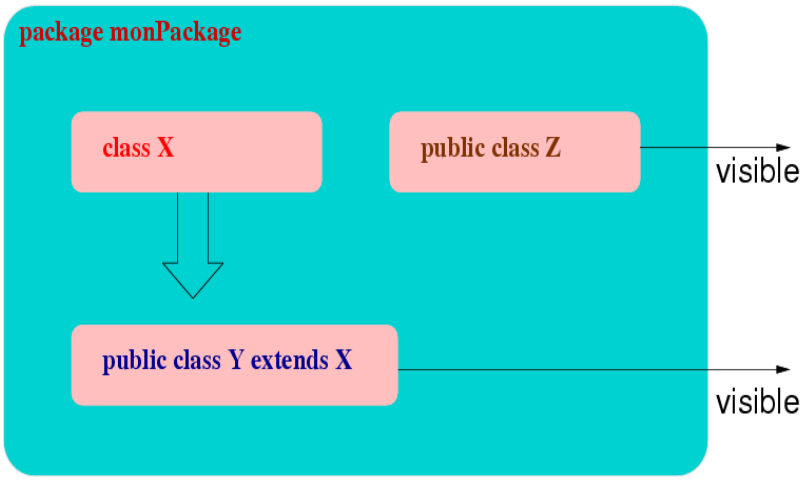
8. L'encapsulation et la visibilité des membres

Dans le chapitre précédent nous avons abordé le principe d'encapsulation et les deux nivaux de visibilité public et private. En fait, Il existe 3 modificateurs qui peuvent être utilisés pour définir les attributs de visibilité des entités. Leur utilisation permet de définir des niveaux de protection différents (présentés dans un ordre croissant de niveau de protection offert) :

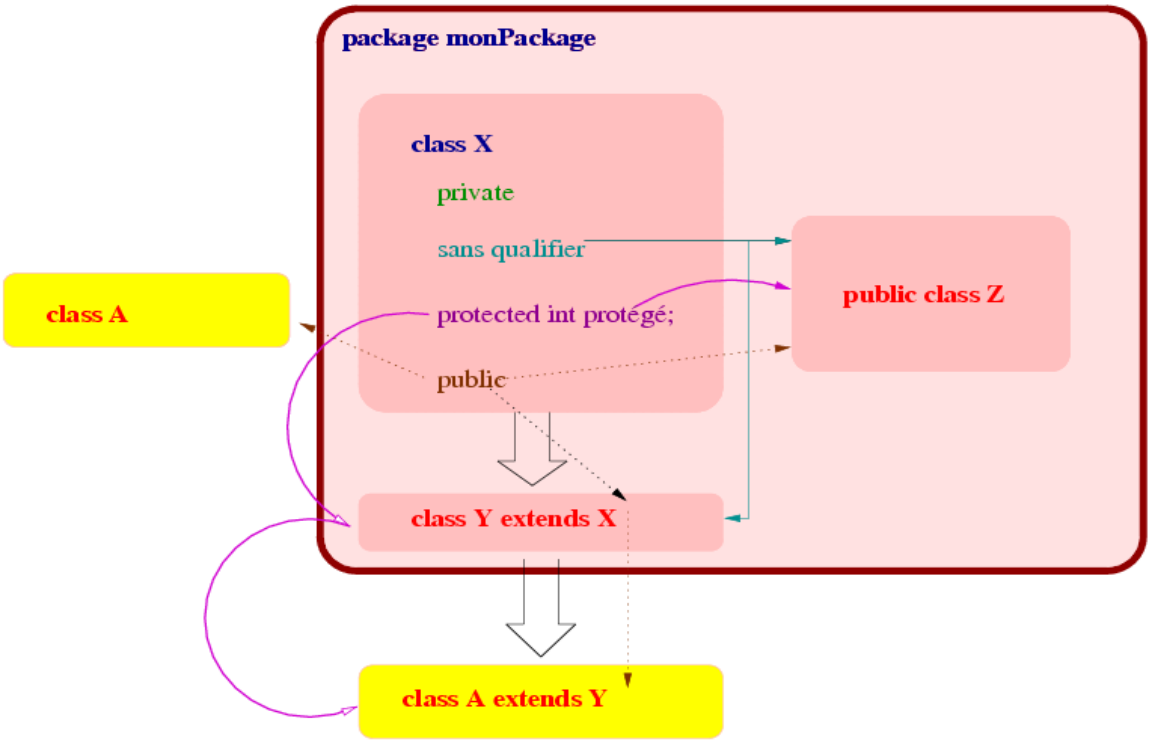
Modificateur	Rôle
public	-Un attribut, méthode ou classe déclarée public est visible par tous les autres objets.

	- Dans un fichier Java, il ne peut exister qu'une seule classe publique, portant le même nom du fichier.
par défaut : package friendly	-Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut (lorsqu'aucun modificateur n'est précisé). -Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package.
protected	- Si une classe, une méthode ou une variable est déclarée protected , seules les méthodes présentes dans le même package que cette classe ou ses sous classes pourront y accéder. - On ne peut pas qualifier une classe avec protected.
private	-C'est le niveau de protection le plus fort. -Les membres ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe prévues à cet effet.

Ces modificateurs d'accès sont mutuellement exclusifs. La figure suivante illustre les nivaux de visibilité des classes.



La figure suivante illustre les nivaux de visibilité des membres des sclasses.



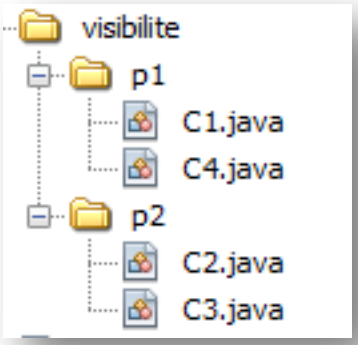
Remarque : Une classe dans un fichier indépendant ne peut avoir que la visibilité par défaut (package friendly) ou publique.

9. Série d'exercices

Exercice 1

Soient deux classes C1 et C4 dans le même package, C2 et C3 dans un autre package. Ci-dessous le contenu de chaque classe.

Déterminer les instructions invalides dans ces classes. Expliquer.




```
package visibilite.p1;
public class C1 {
    // les attributs

    private int xPrive = 10;
    int xDefaut = 20;
    protected int xProtege = 30;
```

```
package visibilite.p2;
import visibilite.p1.C1;
class C2 extends C1 {
    void methode() {
        1. System.out.println(xPrive);
        2. System.out.println(xDefaut);
        3. System.out.println(xProtege);
```

public int xPublique = 40; }	4. System.out.println(xProtege); } }

package visibilite.p1;
public class C4 {
void methode() {
C1 obj = new C1();
1. System.out.println(obj.xPrive);
2. System.out.println(obj.xDefaut);
3. System.out.println(obj.xProtege);
4. System.out.println(obj.xPublique);
}
}
	package visibilite.p2;
	import visibilite.p1.C1;
	class C3 {
	void methode() {
	C1 obj = new C1();
	1. System.out.println(obj.xPrive);
	2. System.out.println(obj.xDefaut);
	3. System.out.println(obj.xProtege);
	4. System.out.println(obj.xPublique);
	}
	}

Solution

Pour la classe C2, la seule instruction invalide est la première: seul l'attribut privé ne peut être hérité.

Pour la classe C3, les instructions invalides sont:

- Instruction 1: incorrect, l'attribut privé n'est pas visible à partir d'une autre classe que la sienne.
- Instruction 2: incorrect, l'attribut ayant la visibilité par défaut n'est visible que dans les classes du même package que C1. Ce n'est pas le cas pour C2.
- Instruction 3: incorrect, C2 n'hérite pas de C1 et elles ne sont pas dans le même package, donc C2 n'a pas accès à son attribut protégé.
- Instruction 4: correct, l'attribut publique est visible par toutes les classes.

Pour la classe C4, la seule instruction invalide est la première.

- Instruction 1: incorrect, l'attribut privé n'est visible que dans sa classe
- Instruction 2: Correct, car C4 est de même paquetage que C1, donc peut accéder à ses attributs ayants la visibilité par défaut.
- Instruction 3: Correct, car C4 est de même paquetage que C1, donc peut accéder à ses attributs protégés.
- Instruction 4: correct, L'attribut publique est visible par toutes les classes.

Exercice 2

1) Créez une classe *MaterielInformatique* qui décrit un matériel informatique par un code, un prix et une marque.
   a) Doter cette classe par un constructeur
   b) Doter la par des accesseurs.

c) Donner la méthode *afficher* qui affiche la chaine de caractère: "Je suis un matériel informatique"

2) Créer la classe *Ordinateur* qui hérite de *MaterielInformatique*. Un ordinateur est définit par un code, un prix et une marque, la taille de RAM et la taille du disque dur.
   a) Doter cette classe par un constructeur
   b) Doter la par des accesseurs.
3) Créer la classe TestHeritage qui crée un objet ordinateur, puis appelle la méthode *afficher* à partir de cet objet. Quel est le résultat d'exécution?
4) Redéfinir la méthode *afficher* dans la classe *Ordinateur* pour que le résultat du test devient: *Je suis un matériel informatique*
   Précisément, je suis un ordinateur

Solution

```
1)
class MaterielInformatique {
   // attributs
  protected int code;
 protected float prix;
 protected String marque;

// constructeur
     public MaterielInformatique(int code, float prix, String marque) {
       this.code = code;
       this.prix = prix;
       this.marque = marque;    }

 // accesseurs
   public int getCode() {        return code;    }
   public void setCode(int code) {        this.code = code;    }

   public String getMarque() {        return marque;    }
   public void setMarque(String marque) {        this.marque = marque;    }

   public float getPrix() {        return prix;    }
   public void setPrix(float prix) {        this.prix = prix;    }

   // méthode afficher
   public void afficher() {
      System.out.println( "Je suis un matériel informatique"); }
```

```
}// Fin classe
```

2)

```
class Ordinateur extends MaterielInformatique {
// attributs privés
    private int tailleRAM;
    private int tailleDisqueDur;

    // constructeur
    public Ordinateur(int code, float prix, String marque, int tailleRAM, int tailleDisqueDur) {
        super(code, prix, marque); // appel au constructeur de la classe de base
        this.tailleRAM = tailleRAM;
        this.tailleDisqueDur = tailleDisqueDur;    }

//  les accesseurs
    public int getTailleDisqueDur() {        return tailleDisqueDur;    }

    public void setTailleDisqueDur(int tailleDisqueDur) {
        this.tailleDisqueDur = tailleDisqueDur;    }

    public int getTailleRAM() {        return tailleRAM;    }

    public void setTailleRAM(int tailleRAM) {        this.tailleRAM = tailleRAM;    }
}// fin classe
```

3)

```
public class TestHéritage {
   public static void main(String[] args) {
      Ordinateur ord = new Ordinateur(234, 1800, "Dell", 1024, 700);
      ord.afficher();
      // on utilise la méthode afficher comme si elle est définie dans la classe fille
   }
}
```

Exécution

```
Je suis un materiel informatique
```

4) La méthode d'affichage redéfinie est :

```
public void afficher() {
     super.afficher();
      System.out.println( " Précisément, je suis un ordinateur");
}
```

Exercice 3

Soit les classes et les objets suivants:

```
class C1{}
class C2 extends C1{}
class C3 extends C2{}

class Test {
public static void main (String args[ ]){
   C1 a = new C1();
   C2 b = new C2();
   C3 c = new  C3();
   C2 d = new C3();
}
```

Les instructions suivantes sont elles correctes? Si non, pourquoi?

```
//tests
1. a=b;
2. a=c;
3. c=a;
4. c=b;
5. b=c;
```

Solution

1. a=b : correcte car le type de b (C2) est une sous classe du type de a(C1).
2. a=c : correcte car C3 est une sous classe de C1.
3. c=a; : erreur de compilation car a n'appartient pas à la classe de c ni à une classe descendante de celle-ci.
4. c=d; erreur de compilation car d est de type C2 (bien que instanciée par le constructeur de C3), donc d n'appartient pas à la classe de c ni à une classe descendante de celle-ci.
5. b=c; correcte car le type de c (C3) est une sous classe du type de b(C2).

Chapitre 6. Les classes abstraites et les interfaces

Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Comprendre la notion de classes et de méthodes abstraites
- Savoir écrire un programme contenant des classes héritant d'une classe abstraite.
- Comprendre la notion d'interfaces
- Savoir écrire un programme contenant des classes implémentant une interface

Pré-requis

- Classes et objets en Java
- L'héritage
- Le polymorphisme

Eléments de contenu

1. Classes et méthodes abstraites
    1.1. Les méthodes abstraites
    1.2. Les classes abstraites
2. Les interfaces
    2.1. Déclaration des interfaces
    2.2. Implémenter les Interfaces
    2.3. Héritage entre interfaces

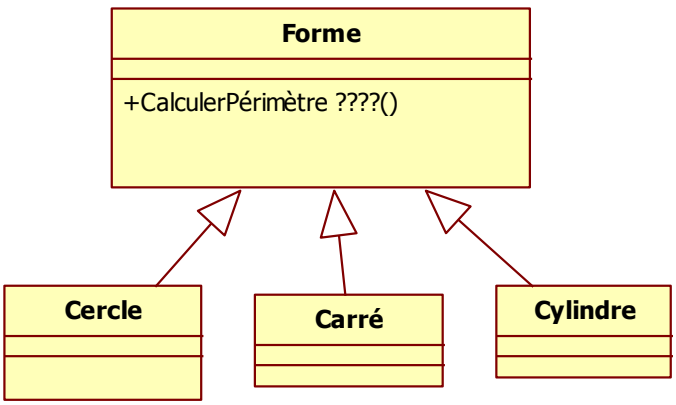
1. Classes et méthodes abstraites

1.1. Les méthodes abstraites

Une méthode abstraite est une méthode dont on connait le prototype sans décrire l'implémentation. Elle est définie comme suit

[modificateur] ***abstract*** *void nomMéthode() ;*

Exemple d'utilisation : soit une classe Forme avec des méthodes comme calculerPérimètre, calculerSuperficie, etc . est les classes Cercle, Carré et cylindre qui en hérite.



➔ On ne sait pas implanter la méthode *calculerPérimètre* dans le cas d'une forme générale. Par contre, une fois on précise la forme (un carré, un cercle, etc.), on sait implanter cette méthode pour cette forme. Autrement dit, un objet de type forme n'a aucun intérêt en soi. Les objets utiles sont les carrés, les cercles, etc. Par contre, il existe souvent des attribut et méthodes communs à toutes les formes.

On dit alors que la méthode *calculerPérimètre* est **abstraite:** sa définition est supposée être donnée par redéfinition dans les classes dérivées. La classe de base ne peut fournir une méthode par défaut.

1.2. Les classes abstraites

Une classe abstraite est une classe dont la définition est précédée par le mot clé *abstract.* Il s'agit généralement d'une classe partiellement implantée c'est-à-dire que certaines de ses méthodes sont abstraites. La syntaxe est la suivante:

```
abstract class  NomClasse {
// attributs et méthodes
}
```

Les règles suivantes sont à respecter:

- ✓ Le langage Java impose de qualifier la classe d'abstraite lorsqu'une de ses méthode est abstraite. La classe Forme est donc abstraite.

- ✓ Java autorise de déclarer une classe abstraite sans avoir de méthode abstraite, mais le contraire est interdit: Si l'on définit une sous classes sans implanter toutes les méthodes abstraites de la classe de base, une erreur de compilation est générée.

- ✓ Il est à noter qu'une **classe abstraite ne peut pas être instanciée.** Dans notre exemple, il ne sera jamais possible de créer un objet de type Forme. Les objets qui sont susceptible d'exister sont des formes bien précises : des carrés, des cercles, des lignes, etc. Ces formes effectives, seront des objets des classes obtenus en dérivant la classe Forme.

Exemple

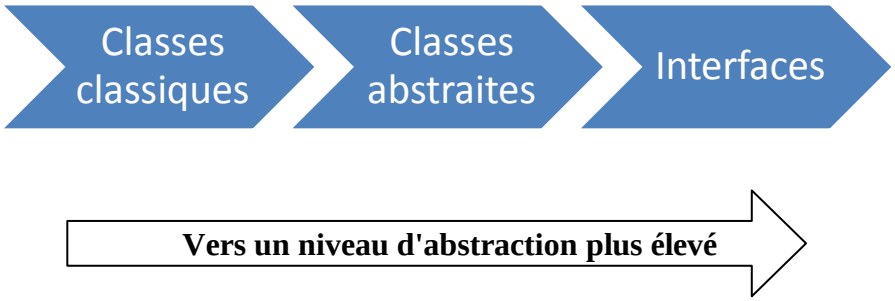
```
abstract class Forme {
...
public abstract void calculPérimètre() ;
... }

class Carrée extends Forme {
...
public void calculPérimètre () { ... }  // Carrée doit implanter la méthode calculPérimètre.
... }
```

Le modificateur de visibilité d'une méthode abstraite peut être: protected ou public , mais pas private, parce qu'on aura besoin d'implanter la méthode abstraite dans les classes filles.

L'utilité du principe d'abstraction est de permettre de regrouper des données et méthodes communes dans une classe et de spécifier les méthodes qu'une classe dérivée de celle-ci doit absolument implanter.

2. Les interfaces

Les interfaces sont un moyen de préciser les services qu'une classe peut rendre. Autrement dit, la définition d'une interface consiste à donner une collection de constantes et de méthodes abstraites à implémenter par d'autres classes. → On dit qu'une interface est une sorte **de contrat de services entre ces classes**.



2.1. Déclaration des interfaces

Comme les classes, les interfaces sont constituées de champs et de méthodes ; mais, comme nous l'avons déjà dit, il existe de très fortes contraintes sur la nature des membres d'une interface :

- Toutes les méthodes qui sont déclarées dans cette interface sont abstraites ; aucune implantation n'est donnée dans la définition de l'interface. Toutes les méthodes étant publiques et abstraites. Les mots clés public et abstract n'apparaissent pas : ils sont implicites.

- Toutes les méthodes d'une interfaces sont toujours non statiques.

- Tous les champs d'une interface sont public, static et final. Ils sont là pour définir des constantes qui sont parfois utilisées dans les méthodes de l'interface. Les mots clés static et final ne figurent pas dans la définition des champs, ils sont implicites.

On doit utiliser le mot réservé **interface** pour la déclaration d'une interface.

```
interface NomInterface {
// Déclaration des constantes
// Déclaration des méthodes abstraites
}
```

Remarque: Une interface peut être qualifiée de public. Une interface public peut être utilisée par n'importe quelle classe. En l'absence de ce qualifier, elle ne peut être utilisée que par les seules classes appartenant au même package que l'interface.

Exemple:

```
interface Service
{
        int MAX = 1024 ;  // c'est une constante
        int une_méthode(...) ; // c'est uen méthode abstraite
        ...
```

```
}
```

2.2. Implémenter les Interfaces :

Les interfaces définissent des ``promesses de services''. Mais c'est au niveau de la classe qu'on peut rendre effectivement les services qu'une interface promet. Autrement dit, l'interface toute seule ne sert à rien : il nous faut une classe qui implémente l'interface. On dit:

- **Une classe hérite d'une classe**
- **Une classe implémante une interface**

La syntaxe est la suivante:

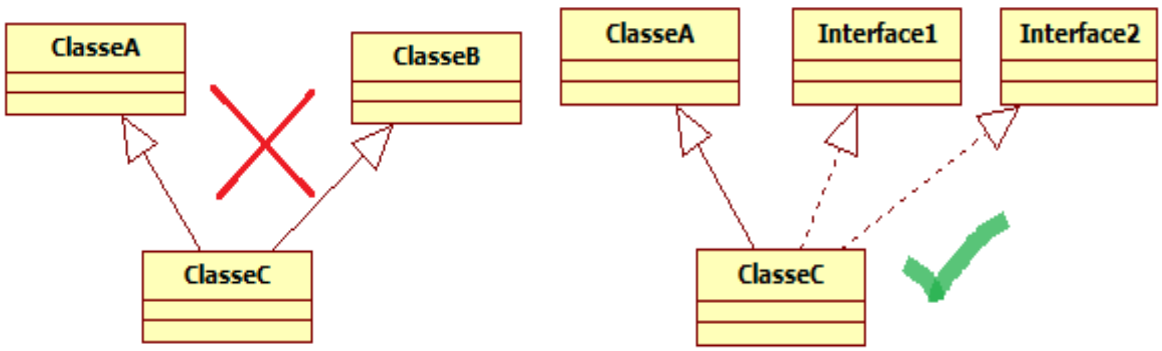
```
class NomClasse  implements NomInterface
{ ...
    //          Implémentation des méthodes de l'interface
 }
```

Par cette déclaration, la classe promet d'implanter toutes les méthodes déclarées dans l'interface en respectant les règles suivantes

- ✓ La signature des méthodes doit évidemment être la même que celle promise par l'interface. Dans le cas contraire, la méthode est considérée comme une méthode de la classe et non une implantation de l'interface.
- ✓ Si une méthode de même signature existe dans la classe mais avec un type de retour différent une erreur de compilation est générée.

Une interface peut être implémentée par plusieurs classes. Ses méthodes sont donc **polymorphes** qui qu'elles sont définies de manières différentes dans les classes.

Remarque: Le langage Java ne permet pas l'héritage multiple: une classe ne peut pas hériter de plusieurs classes. Il pallie ce manque par l'introduction des interfaces.



Toute variable de type interface peut être vue comme instance de touts les classes implémentant cette interface.

2.3. Héritage entre interfaces

Tout comme les classes, les interfaces peuvent être organisées de manière hiérarchique à l'aide de l'héritage. Une classe ne peut être dérivée que d'une autre classe ; de même, une interface ne peut être dérivée que d'une autre interface. Mais, contrairement aux classes, une interface peut étendre plusieurs interfaces.

```
        interface A extends B {... }
        interface A extends B, C, D {... }
// aves A, B, C et D des interfaces
```

Une interface dérivée hérite de toutes les constantes et méthodes des interfaces ancêtres, à moins qu'un autre champ de même nom ou une autre méthode de même signature soit redéfinie dans l'interface dérivée.

3. Série d'exercices

Exercice 1

1) Définir une classe abstraite Forme ayant une méthode abstraite calculerSuperficie et une redéfinition de la méthode toString.
2) Définir une classe Carré qui hérite de Forme.
3) Tester les méthodes toString et calculerSuperficie de deux manières.

Solution

```
1) public abstract class  Forme {

// la méthode abstraite
   public abstract   double calculerPérimètre();

// et une méthode non abstraite
   public String toString() {
      return "je suis une Forme";
   }
}
```

```
2) class Carré extends  Forme {
// attribut
private   double coté;
```

```
//constructeur
   public Carré(double coté) {
      this.coté = coté;    }

//méthodes
   public double calculerPérimètre() {
      return 4 * coté;     }

   public String toString() {
      return "je suis un carré";
   }
}
```

```
3) public class TestForme {

   public static void main(String[] args) {
      Forme forme;
     // forme= new Forme(); --> interdit car forme est abstraite
      Carré carre = new Carré(3);
      System.out.println(carre);
      System.out.println("Périmètre du carré " + carre.calculerPérimètre());

//       carre=forme; --> faux
      forme = carre;  // juste, forme est devenue un carré
      System.out.println(forme);
      System.out.println("Périmètre du carré " + carre.calculerPérimètre());

   }
}
```

```
Exécution
je suis un carré
Périmètre du carré 12.0
je suis un carré
Périmètre de la forme carrée 12.0
```

Exercice 2:

1) Définir une interface Forme ayant une seule méthode calculerSuperficie.
2) Définir une classe Carré qui implémente Forme.
3) Tester la méthode calculerSuperficie.

Solution

```
1) interface Forme {
   double calculerPérimètre();
}
```

```
2) class Carré implements Forme {
// attribut
   private double coté;

//constructeur
   public Carré(double coté) {
      this.coté = coté;    }

// méthode
   public double calculerPérimètre() {
      return 4 * coté;
   }
}
```

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

      Forme f;
      Carré carre = new Carré(3);

//       carre=f; --> faux
      f = carre;  // juste
      System.out.println("Périmètre du carré " + carre.calculerPérimètre());
      System.out.println("Périmètre de la forme carrée " + f.calculerPérimètre());
   }
}
```

Exécution

Périmètre du carré 12.0

Périmètre de la forme carrée 12.0

Chapitre 7. La gestion des exceptions

Objectifs

Le but de ce chapitre est de comprendre les exceptions et comment les gérer en Java. Il aborde aussi la création des exceptions personnalisées.

Pré-requis

- Classes et objets en Java
- Héritage entre classes

Eléments de contenu

1. Présentation des exceptions

Une exception est une erreur qui se produit lors de l'exécution d'un programme, et qui va provoquer un fonctionnement anormal de ce dernier (par exemple l'arrêt du programme).

Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java. Il se compose d'objets représentant les erreurs. Lors de la détection d'une erreur, un objet qui hérite de la classe Exception est créé (on dit qu'une exception est levée) et propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité. Ces mécanismes permettent de renforcer la sécurité du code Java.

Exemple: Soit le programme suivant :

```
1 public class Div {

2  public static int divint (int x, int y) {
3    return (x/y);
4    }

5  public static void main (String [] args) {
6    int c=0,a=1,b=0;
7    c= divint(a,b);  // division par zéro
8    System.out.println("res: " + c);
9    System.exit(0); } }
```

L'exécution affiche l'erreur suivante:

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

L'erreur est à la ligne 3 dans la méthode divint, appelée par la ligne 7 dans main.

2. Capturer des exceptions

Le mécanisme de la gestion des exceptions de Java utilise le couple de mots clé try/catch:

- On essaye d'exécuter le bloc pouvant provoquer l'erreur (le bloc **try**), ainsi ce bloc devient isolé du reste du programme.
- Si une erreur se présente dedans, on va automatiquement exécuter les instructions du bloc **Catch** (le mot catch rappelle le fait de "**attrapper**" l'exception).
- Si aucune erreur n'est détectée, le bloc catch est ignoré.

La syntaxe est la suivante :

```
try {
    //Instructions pouvant provoquer l'erreur;
}

catch (TypeException e) {
  // traitement de l'Exception
}
```

Dans un bloc catch, on peut:

a) corriger l'erreur qui a provoqué l'exception
b) proposer un traitement alternatif
c) retourner une valeur particulière
d) sortir de l'application
e) faire qu'une partie du traitement et la méthode appelante fera le reste

…

Retour à l'exemple:

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

S'il y a plusieurs types d'erreurs et d'exceptions à intercepter, on peut définir un bloc catch par type d'exception, ceci permet de filtrer les exceptions. Cependant, il faut faire attention à l'ordre des clauses catch pour traiter en premier les exceptions les plus précises (sous classes) avant les exceptions plus générales, c'est-à-dire qu'un type d'exception de ne doit pas être capturé après un type d'une exception d'une super classe. Une erreur de compilation est provoquée dans le cas contraire.

Exercice

Sachant que la classe *ArithmeticException* hérite de la classe *Exception*, le code suivant est il valide? Sinon, comme le corriger.

```
public class Test {
   public static void main(String[] args) {
      int i = 3;      int j = 0;
      try {
           System.out.println("résultat = " + (i / j));
      }
      catch (Exception e) {}
      catch (ArithmeticException e) {}
   }
}
```

Solution

Ce code est invalide. Il fallait inverser l'ordre des blocs catch.

3. Le bloc finally

Le bloc finally est un bloc optionnel. On peut le trouver :

- ✓ Soit après un bloc try
- ✓ Soit après un bloc try suivi d'un bloc catch

Il s'exécute que ce soit une exception s'est produite ou non. Son exécution a lieu :

- ✓ après l'exécution complète du bloc Try
- ✓ après l'exécution du bloc Catch

Tout code qui a besoin d'être exécuté avant de quitter le programme doit être ajouté au bloc *finally*. Il s'agit des opérations de nettoyages telle que la fermeture des fichiers ouvert**.** La syntaxe est la suivante:

```
try {
  //operation_risquée;
} catch ( UneException e) {
  //traitements
} finally {
  //traitement_pour_terminer_proprement;
}
```

4. Hiérarchie des exceptions

4.1. La classe Throwable

La classe throwable est la classe mère de toutes les exceptions et erreurs. Elle descend directement de la classe Object, c'est la classe de base pour les traitements des

erreurs. Les différents types d'exceptions héritent donc ses méthodes et attributs. Les principales méthodes de la classe Throwable sont :

Méthode	Rôle
String getMessage( )	lecture du message
void printStackTrace()	affiche l'exception et l'état de la pile d'exécution au moment de son appel.
void printStackTrace(PrintStreams)	Idem mais envoie le résultat dans un flux

Elles servent généralement à identifier l'exception dans le bloc catch.

Exercice:

Deviner le résultat d'exécution de ce programme.

```
public class Test  {
public static void main( String[] args) {
int i = 3;
int j = 0;
try {
System.out.println("résultat = " + (i / j));
}
catch (ArithmeticException e) {
System.out.println("getmessage");
System.out.println(e.getMessage());

System.out.println("toString");
System.out.println(e.toString());

System.out.println("printStackTrace");
e.printStackTrace();
}
}
}
```

Solution

Le résultat est:

getmessage
/ by zero
toString
java.lang.ArithmeticException: / by zero
printStackTrace
java.lang.ArithmeticException: / by zero

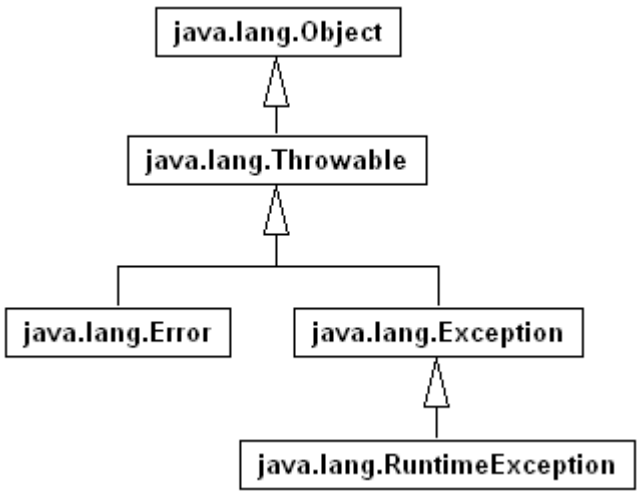
at Test.main(Test.java:12)

4.2. Les exceptions contrôlées/ non contrôlées

Il existe deux types d'exceptions : les exceptions contrôlées et les exceptions non contrôlées:

- ✓ Une exception contrôlée est une exception qui doit être capturée par le programme.
- ✓ Une exception non contrôlée ne nécessite pas d'être capturée.

En Java, les exceptions non contrôlées se subdivisent en deux catégories Error et RuntimeException. Toutes les autres exceptions sont contrôlées.



Deux classes principales héritent de Throwable (en fait, toutes les exceptions dérivent de la classe Throwable)

- La classe Error représente une erreur grave intervenue dans la machine virtuelle Java ou dans un sous système Java. L'application Java s'arrête instantanément dès l'apparition d'une exception de la classe Error.
- La classe Exception représente des erreurs moins graves. Les exceptions héritant de la classe RuntimeException n'ont pas besoin d'être détectées impérativement par des blocs try/catch.

5. Propagation des Exceptions

Le principe de propagation des exceptions est le suivant:

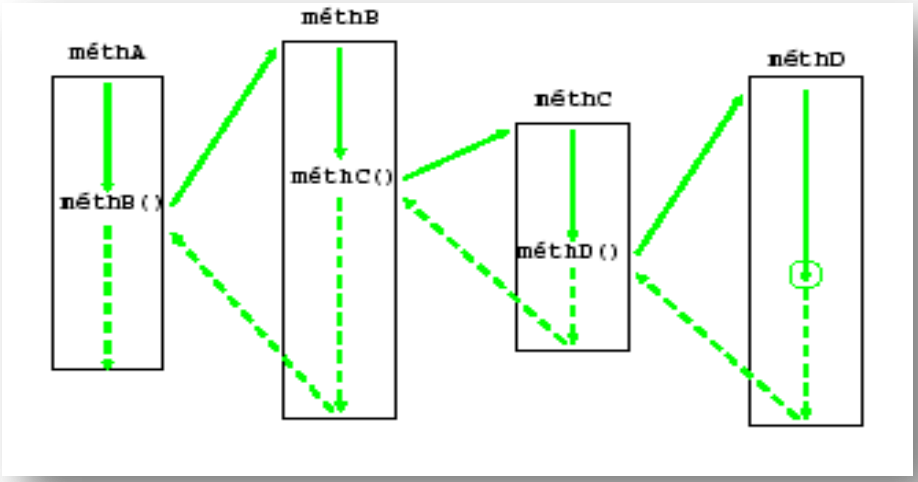
- Lorsqu'une exception est levée dans une méthode donnée, les instructions qui suivent le lancement de l'exception, se trouvant dans cette méthode, sont ignorées.

- L'exception peut-être attrapée par un bloc catch (s'il existe un try) se trouvant dans cette méthode.

- Si l'exception n'a pas été capturée, le traitement de cette exception remonte vers la méthode appelante, jusqu'à être attrapée ou bien on est arrivé à la fin du programme.

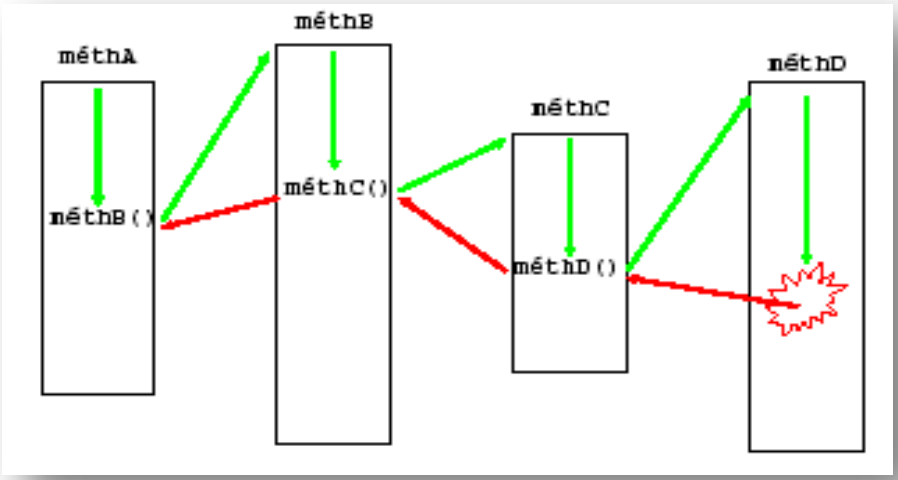
Les figures suivantes illustrent ce mécanisme.

a) Pas d'exception



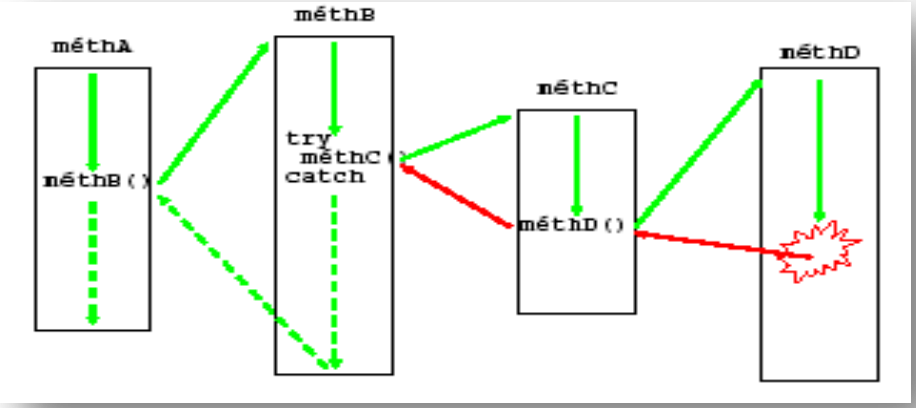
- La méthode méthA appelle méthB qui apppelle à son tour méthC. Cette dernière appelle méthD.
- Aucune exception n'est levée
- le résultat est donc retourné par la méthode méthD à sa méthode appelante qui termine son exécution et retourne à son tour le résultat à sa méthode appelante.
- L'exécution du programme se termine sans problème.

b) Exception non capturée:



- Une exception est **levée et non capturée** au niveau de la méthode méthD.
- méthD ne termine pas son exécution. Le traitement de l'exception remonte vers la méthode appelante, qui lui même ne capture pas l'exception et délègue le traitement à sa méthode appelante, et ainsi de suite jusqu'à arriver à méthA.
- Le programme s'arrête donc sans terminer l'exécution de ses méthodes.

c) Exception capturée:



- Une exception est **levée et non capturée** au niveau de la méthode méthD.
- méthD ne termine pas son exécution. Le traitement de l'exception remonte vers la méthode appelante, qui lui même ne capture pas l'exception et délègue le traitement à sa méthode appelante méthB.
- méthB capture l'exception, on éxécute le bloc catch et on termine normalement l'exécution du programme.

Pour résumer, dans le cas où il n'y pas eu d'exception levée par aucune des instructions du bloc try, l'exécution du programme se poursuit après le dernier bloc catch. Comme si aucun bloc catch n'a été défini dans le programme.
Par contre, si une exception est levée, deux cas se présentent:

- **S'il existe un bloc catch** qui peut capturer cette exception, il sera exécuté en premier, puis le programme poursuit après le dernier bloc catch.
- **Si aucun bloc catch ne peut capturer cette exception**, la main est donnée à la méthode appelante. À elle de voir si elle peut traiter cette exception, sinon on remonte de méthode en méthode à la recherche d'un bloc catch adéquat.

6. Lever une Exception

Une exception est levée si une opération illégale risquerait d'avoir lieu. Le développeur peut lancer une exception dans le code en utilisant le mot clé throw. Pour lever une exception, on doit créer une instance de la classe où réside cette exception. La syntaxe est la suivante :

throw new ClasseException();

Pour lancer l'exception, on peut utiliser aussi le constructeur Exception (String s) qui construit une exception et stocke le message dans l'exception créée.

Exemple:

```
public class Div
{
        public static int divint (int x, int y) {
                if (y==0) throw new ArithmeticException(); // lever une exception
                return (x/y);
        }
        public static void main (String [] args)
        {
                int c=0,a=1,b=0;
                try
                { c= divint(a,b);   }
                catch (ArithmeticException e)
                {  System.out.println("Erreur a été capturée");   }

                System.out.println("res: " + c);
        }
}
```

Il est à noter que si on crée une méthode susceptible de lever une exception qu'elle ne traite pas localement, cette méthode doit mentionner le type de cette exception dans son en-tête en utilisant le mot réservé *throws*. Ainsi, soit l'exception est traitée localement, soit elle est propagée par throws.

Retour à l'exemple:

```
public class Div3
{
        public int divint (int x, int y) throws MaArithmeticException
        {       if (y==0) throw new MaArithmeticException();
                return (x/y);
        }
```

```
        public int carré(int x, int y) throws MaArithmeticException
        { return divint(x,y)*divint(x,y); }
        public static void main (String [] args)
        {       int c=0,a=1,b=0;
                try  { c= divint(a,b);   }
                catch (MaArithmeticException e)  {
                        System.out.println("Erreur a été capturée");}
                System.out.println("res: " + c);
        }
}
```

7. Définir sa propre exception

Si on veut pouvoir signaler un évènement exceptionnel d'un type non défine en Java, on peut créer ses propres exceptions. Il faut donc créer une classe qui **hérite de la classe *Exception*** *(*qui se trouve dans le package *java.lang.)*. Il est préférable (par convention) d'inclure le mot « Exception » dans le nom de la nouvelle classe.

Les exceptions crées par les programmeurs sont toutes **non contrôlées**, elles doivent donc être toujours capturées. Si elles ne sont pas traitées dans les méthodes qui les lancent, on doit ajouter *throws* dans l'en-tête de ces méthodes.

8. Série d'exercices

Exercice 1

Lire le code ci-dessous et répondre aux questions suivantes:

1) Pourquoi l'entête de la méthode moyenne contient l'instruction *throws*?
2) Quels sont les exceptions qui peuvent être lancées dans ce programme? Déterminez celles prédéfinies et celle définies par le programmeur.
3) Que donne ce programme si on initialise le tableau liste dans la méthode par les valeurs suivantes:
   - String[] liste={"8", "-8", "22", "k"};
   - String[] liste={ "a", "b", "22", "c" };

```
public class LancerException {
   static int moyenne(String[] liste) throws ToutInvalideException {
      int somme = 0, entier, nbNotes = 0;
```

```
        int i;
        for (i = 0; i < liste.length; i++) {
           try {
              entier = Integer.parseInt(liste[i]);
              if (entier < 0) {
                 throw new IntervalleException("note petite");
              } else if (entier > 20) {
                 throw new IntervalleException("notre tres grande");
              }
              somme += entier;
              nbNotes++;
           } catch (NumberFormatException e) {
              System.out.println("La " + (i + 1) + " eme note n'est pas entière");
           } catch (IntervalleException e) {
              System.out.println(e.getMessage());

           }
        }
        if (nbNotes == 0) {
           throw new ToutInvalideException();
        }
        return somme / nbNotes;
     }

// la method main
     public static void main(String[] arg) {
        String[] liste={"8", "-8", "22", "k"};
        try {
           System.out.println("La moyenne est " + moyenne(liste));
        } catch (ToutInvalideException e) {
           System.out.println(e);
        }
     }
}


class ToutInvalideException extends Exception {
     public String toString() {
        return "Aucune Note n'est valide";
     }
}
```

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

Solution:

1) Exceptions prédéfinies: NumberFormatException
   Exception définies par le programmeur: IntervalleException, ToutInvalideException

2) Si on teste avec "8", "-8", "22", "k" , l'exécution serait :

```
note petite
notre tres grande
La 4 eme note n'est pas entière
La moyenne est 8
```

Si on teste avec "a", "b", "22", "c", l'exécution serait :

```
La 1 eme note n'est pas entière
La 2 eme note n'est pas entière
notre tres grande
La 4 eme note n'est pas entière
Aucune Note n'est valide
```

Exercice 2

On va reprendre le problème de l'étagère des livres vu précédemment en lui ajoutant quelques exceptions.

1) La méthode ajoute des livres ("boolean ajouter(Livre)") à la fin de l'étagère. Si l'étagère est pleine on veut que l'exception EtagerePleineException soit lancée et non capturée.

   a. Créer la classe *EtagerePleineException* dont le message est *"Problème: étagère pleine."*

   b. Donner la nouvelle méthode étagère.

2) La méthode de signature "Livre getLivre(int)" permet de récupérer un livre dont on donne la position sur l'étagère. On va la modifier pour qu'elle lance une exception *IndiceInvalideException*, sans la capturer, si l'indice fournit est inférieur au nombre de livres existants.

3) Capturer ces exceptions dans la méthode main.

Solution

1) On doit ajouter *throws EtagerePleineException* dans l'entête de la méthode car on ne capture pas cette exception dans la méthode.

```
// ajouter un livre à la fin de l'étagère
public void ajouter(Livre l) throws EtagerePleineException{
   if (nbLivres < livres.length) {
      livres[nbLivres++] = l;
   } else {
      throw  new EtagerePleineException();
      //System.err.println("Etagere pleine");
   }
}
```

La classe de l'exception est la suivante:

```
class EtagerePleineException extends Exception{
// contructeur
   public EtagerePleineException() {
      super("Problème: étagère pleine");
   }

}
```

2) De même, on doit ajouter le mot clé *throws* parce qu'on ne capture pas cette exception dans la méthode.

```
// méthode qui renvoie le livre placé sur l'étagère dans la position indiquée.
 public Livre getLivre(int position) throws IndiceInvalideException {
    if (position > 0 && position <= nbLivres) {
       return livres[position - 1];
    } else {
     throw new IndiceInvalideException();
    }
 }
```

La classe de l'exception est la suivante

```
class IndiceInvalideException extends Exception{
   public IndiceInvalideException() {
      super("Problème: Indice Invalide");
   }
}
```

3) On doit capturer les deux exceptions dans la méthode main.

```
public static void main(String[] args) {
      Livre l1 = new Livre("a1", "t1", 200, 200);
      Livre l2 = new Livre("a2", "t2", 100, 100);
      Livre l3 = new Livre("a3", "t3", 399, 160.8);
      Etagere etagere1 = new Etagere(4);

      try {
         etagere1.ajouter(l1);
         etagere1.ajouter(l2);
         etagere1.ajouter(l3);
      } catch (EtagerePleineException e) {  // --- Capturer EtagerePleineException
         System.out.println(e.getMessage());
      }

      try {
         System.out.println(etagere1.getLivre(5));
      } catch (IndiceInvalideException ex) { // --- Capturer IndiceInvalideException
         System.out.println(ex.getMessage());
      }
```

Bibliographie

● Livres
– Programmer en Java, 7e Edition, Claude Delannoy, Eyrolles, 2011
– The Java Tutorial : A Short Course on the Basics, 5th Edition, Collectif, Prentice Hall, 2006
– Effective Java, 2nd Edition, Joshua Bloch, Prentice Hall, 2008
– Java in a nutshell, 5th edition, David Flanagan, O'Reilly, 2009
- Cours programmation en Java, Mohamed Salah BOUHLEL, ISETJ, 2010
- Initiation à la programmation orientée objet en Java: Rappels de cours et exercices corrigés, Chiraz ZRIBI, centre de publication universitaire, 2003

● Sites web
– Le site officiel Java, http://www.oracle.com/technetwork/java/index.html
– Le tutorial Java, http://docs.oracle.com/javase/tutorial/
– l'API du JDK 1.7, http://docs.oracle.com/javase/7/docs/api/