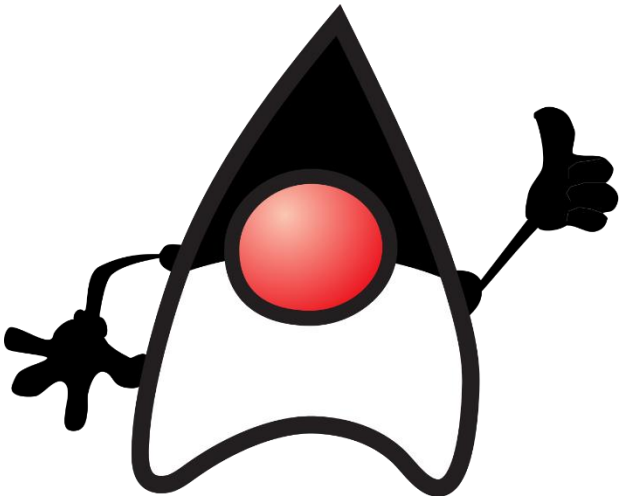
INITIATION A LA PROGRAMMATION ORIENTEE-OBJET AVEC LE LANGAGE JAVA



Dr TREY

Enseignante chercheure

Table des matières

Chapitre 1 : INTRODUCTION A JAVA

Java est un langage de programmation à usage général, évolué et orienté objet dont la syntaxe est proche du C. Ses caractéristiques ainsi que la richesse de son écosystème et de sa communauté lui ont permis d'être très largement utilisé pour le développement d'applications de types très disparates. Java est notamment largement utilisé pour le développement d'applications d'entreprises et mobiles.

Un programmeur Java écrit son code source, sous la forme de classes, dans des fichiers dont l'extension est .java. Ce code source est alors compilé par le compilateur *javac* en un langage appelé bytecode et enregistre le résultat dans un fichier dont l'extension est.***class***. Le bytecode ainsi obtenu n'est pas directement utilisable. Il doit être interprété par la machine virtuelle de Java qui transforme alors le code compilé en code machine compréhensible par le système d'exploitation. C'est la raison pour laquelle Java est un langage portable : le bytecode reste le même quel que soit l'environnement d'exécution.

1. Les caractéristiques

1.1. Présentation de Java

Java est un langage de programmation orienté objet créé par Sun Microsystems en 1995. La société Sun étant racheté en 2009 par la société Oracle qui détient et maintient désormais Java.

Le langage Java a la particularité principale d'être portable sur plusieurs systèmes d'exploitation tels que Windows, MacOS ou Linux. C'est la plateforme qui garantit la portabilité des applications développées en Java. Le langage reprend en grande partie la syntaxe du langage C++, très utilisé par les informaticiens.

Java permet de développer des applications autonomes mais aussi, et surtout, des applications client-serveur, coté client, les applets sont à l'origine de la notoriété du langage. C'est surtout coté serveur que Java s'est imposé dans le milieu de l'entreprise grâce aux servlets et plus récemment les JSP (Java Server Pages) qui peuvent se substituer à PHP et ASP.

Les applications Java peuvent être exécutées sur tous les systèmes exploitation pour lesquels a été développée une plateforme Java, dont le nom technique est JRE (Java Runtime Environment - Environnement d'exécution Java). Cette dernière est constituée d'une JVM (Java Virtual Machine- Machine Virtuelle Java), le programme qui interprète le code Java et le convertit en code natif

1.2. Notion de langage compilés et interprétés

Il existe deux types de langages : les langages compilés et les langages interprétés. Pour rappel, la différence majeure est que pour les langages compilés, on transforme le code source du programme directement en langage compréhensible par la machine à l'aide

d'un compilateur tandis que pour les langages interprétés, on a un programme, l'interpréteur qui va lire chaque ligne du code source et l'exécuter.

- Java est un langage **interprété**, ce qui signifie qu'il n'est pas directement exécutable par le système d'exploitation mais il doit être interprété par un autre programme, qu'on appelle interpréteur. La figure 1 illustre ce fonctionnement.

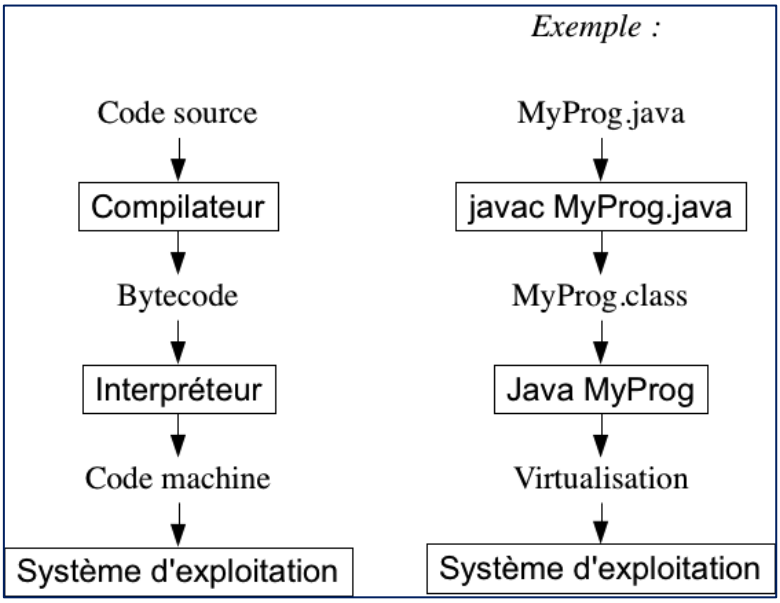


Figure 1 : Interprétation du langage

- Java est portable : il est indépendant de toute plate-forme. Il n'y a pas de compilation spécifique pour chaque plateforme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine.

- Java est orienté objet : comme la plupart des langages récents, Java est orienté objet. Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application. Java n'est pas complètement objet car il définit des types primitifs (entier, caractère, flottant, booléen...).

- Java est simple : le choix de ses auteurs a été d'abandonner des éléments mal compris ou mal exploités des autres langages tels que la notion de pointeurs (pour éviter les incidents en manipulant directement la mémoire), l'héritage multiple et la surcharge des opérateurs.

- Java est fortement typé : toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données. Si une telle conversion doit être

réalisée, le développeur doit obligatoirement utiliser un ***cast*** ou une méthode statique fournie en standard pour la réaliser.

- Java assure la gestion de la mémoire : l'allocation de la mémoire pour un objet est automatique à sa création et Java récupère automatiquement la mémoire inutilisée grâce au ***garbage collector*** qui restitue les zones de mémoire laissées libres à la suite de la destruction des objets.

- Java est sûr : la sécurité fait partie intégrante du système d'exécution et du compilateur. Un programme Java planté ne menace pas le système d'exploitation. Il ne peut pas y avoir d'accès direct à la mémoire. L'accès au disque dur est réglementé dans une applet.

- Java est économe : le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.

- Java est multitâche : il permet l'utilisation de threads qui sont des unités d'exécutions isolées. La JVM, elle-même, utilise plusieurs threads.

Trois plate-formes d'exécution (ou éditions) Java sont définies pour des cibles distinctes selon les besoins des applications à développer :

- Java Standard Edition (J2SE / Java SE) : environnement d'exécution et ensemble complet d'API pour des applications de type desktop. Cette plate-forme sert de base en tout ou partie aux autres plate-formes

- Java Enterprise Edition (J2EE / Java EE) : environnement d'exécution reposant intégralement sur Java SE pour le développement d'applications d'entreprises

- Java Micro Edition (J2ME / Java ME) : environnement d'exécution et API pour le développement d'applications sur appareils mobiles et embarqués dont les capacités ne permettent pas la mise en œuvre de Java SE

Chapitre 2 : LA SYNTAXE ET LES ELEMENTS DE BASES DE JAVA

1. Les règles de base

Java est sensible à la casse. Les blocs de code sont encadrés par des accolades **{ }**. Chaque instruction se termine par un caractère '**;**' (point virgule). Une instruction peut tenir sur plusieurs lignes :

Exemple

```
public void exempleMethode() {
            int somme = $variable1 + _variable2;
      }
```

L'indentation est ignorée du compilateur mais elle permet une meilleure compréhension du code par le programmeur.

Exemple

```
public void exempleMethode() {
            int somme;
            somme = $variable1 + _variable2;
      int produit;
                        produit = $variable1 * _variable2;
      }
```

Identique au pcode ci-dessous

```
public void exempleMethode() {
      int somme;
      somme = $variable1 + _variable2;
      int produit;
      produit = $variable1 * _variable2;
}
```

2. Les mots réservés du langage Java

Abstract	continue	For	new	Switch
assert (Java 1.4)	default	Goto	package	Synchronized
Boolean	do	If	private	This
Break	double	implements	protected	Throw
Byte	else	import	public	Throws
Case	enum (Java 5)	instanceof	return	Transient
Catch	extends	Int	short	Try
Char	final	interface	static	Void
Class	finally	Long	strictfp (Java 1.2)	Volatile
Const	float	Native	super	While

Les mots const et goto ne sont pas actuellement utilisés par le langage mais sont définis dans la liste des mots réservés du langage Java.

Les mots true, false et null ne sont pas des mots réservés mais des littéraux respectivement pour des valeurs booléennes ou pour déclarer une référence non définie à un objet.

3. Les types élémentaires

On distingue 4 catégories de types primitifs (entier, réel, booléens, caractères). L'intervalle de valeurs représentables pour chacun des types peut varier en fonction de l'espace mémoire qu'ils occupent. Les types élémentaires commencent tous par une minuscule.

Type	Désignation	Longueur	Valeurs	Commentaires
boolean	valeur logique : true ou false	1 bit	true ou false	pas de conversion possible vers un autre type
byte	octet signé	8 bits	-128 à 127	
short	entier court signé	16 bits	-32768 à 32767	
char	caractère Unicode	16 bits	\u0000 à \uFFFF	entouré de cotes simples dans du code Java
Int	entier signé	32 bits	-2147483648 à 2147483647	
float	virgule flottante simple précision (IEEE754)	32 bits	1.401e-045 à 3.40282e+038	
double	virgule flottante double précision (IEEE754)	64 bits	2.22507e-308 à 1.79769e+308	
long	entier long	64 bits	-9223372036854775808 à 9223372036854775807	

4. Les identificateurs

Un identificateur est un nom qui permet d'identifier une classe, une variable ou une méthode dans un programme. Il doit être valide, c'est-à-dire composé seulement de caractères Unicode, chiffres, symbole "$", et le caractère de soulignement "_" (Underscore en anglais).

- Un identificateur doit commencer soit par une lettre, le symbole "$", ou un caractère de soulignement "_".
- Il ne doit pas commencer par un chiffre.
- Après le premier caractère, il peut contenir n'importe quelle combinaison de lettres, dollar "**$**", caractère de soulignement "_", ou des chiffres.

Rappel : Java est sensible à la casse. Un identificateur ne peut pas appartenir à la liste des mots réservés du langage Java ni correspondre aux littéraux true, false et null.

5. Les commentaires

Ils ne sont pas pris en compte par le compilateur donc ils ne sont pas inclus dans le pseudo code. Ils ne se terminent pas par un caractère ";". Il existe trois types de commentaire en Java :

Type de commentaire	Exemple
Commentaire abrégé	`//Déclaration de variable` `int variable1; //Déclaration de la variable 1`

Commentaire multiligne	`/* private int longueur ;`
	`private int largeur ;`
	`private int origine_x ;`
	`private int origine_y ;*/`

Commentaire de documentation automatique	`/**`
	` * @author ALekerand`
	` *@since 21/04/2018`
	` *@version 1`
	` */`

6. La déclaration et l'utilisation de variables

6.1. Déclaration et portée de variables

Une variable possède un nom (identificateur), un type et une valeur (***type*** *nomvariable*). Une variable est utilisable dans le bloc où elle est définie. On parle de la portée e la variable. La déclaration d'une variable permet de réserver la mémoire pour en stocker la valeur. Le type d'une variable peut être :

- soit un type élémentaire dit aussi type primitif déclaré sous la forme ***type_élémentaire*** variable;
- soit une classe déclarée sous la forme ***classe*** variable ;

Exemple

```
Scanner monScaner;
int $variable1, _variable2;
int somme;
somme = $variable1 + _variable2;
int produit;
produit = $variable1 * _variable2;
```

La portée des variables `monScaner, $variable1` et `_variable2` s'étend dans tout le code compris ente de la première et la dernière accolade. Celle de la variable `somme` n'est accessible que dans le bloc de la méthode `exempleMethode()`.

Rappel : les noms de variables obéissent à la même règle que celle des identificateurs.

6.2. L'affectation

Le signe **=** est l'opérateur d'affectation et s'utilise avec une expression de la forme :

variable = *expression*.

Exemple :

```
produit = $variable1;
```

Il existe des opérateurs qui permettent de simplifier l'écriture d'une opération d'affectation associée à un opérateur mathématique :

Opérateur	Exemple	Signification
=	a=10	équivalent à : a = 10
+=	a+=10	équivalent à : a = a + 10
-=	a-=10	équivalent à : a = a – 10
***=**	a*=10	équivalent à : a = a * 10
/=	a/=10	équivalent à : a = a / 10
%=	a%=10	reste de la division
^=	a^=10	équivalent à : a = a ^ 10
<< =	a<<=10	équivalent à : a = a << 10 a est complété par des zéros à droite
>>=	a>>=10	équivalent à : a = a >> 10 a est complété par des zéros à gauche
>>>=	a>>>=10	équivalent à : a = a >>> 10 décalage à gauche non signé

6.3. Les comparaisons

Java propose des opérateurs pour toutes les comparaisons :

Opérateur	Exemple	Signification
>	a > 10	strictement supérieur
<	a < 10	strictement inférieur
>=	a >= 10	supérieur ou égal
<=	a <= 10	inférieur ou égal
==	a == 10	Egalité
!=	a != 10	diffèrent de
&	a & b	ET binaire
^	a ^ b	OU exclusif binaire
|	a | b	OU binaire
&&	a && b	ET logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient fausse
||	a || b	OU logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient vraie
? :	a ? b : c	opérateur conditionnel : renvoie la valeur b ou c selon l'évaluation de l'expression a (si a alors b sinon c) : b et c doivent retourner le même type

7. Les opérations arithmétiques

Les opérateurs arithmétiques se notent + (addition), - (soustraction), * (multiplication), / (division) et % (reste de la division). Ils peuvent se combiner à l'opérateur d'affectation.

8. Les structures de contrôles

Les structures de contrôle permettent d'exécuter un bloc d'instructions soit plusieurs fois (instructions itératives) soit selon la valeur d'une expression (instructions conditionnelles ou de choix multiple). Dans tous ces cas, un bloc d'instruction est

- soit une instruction unique ;
- soit une suite d'instructions commençant par une accolade ouvrante "{" et se terminant par une accolade fermante "}".

Elles sont de 2 natures : les Instructions conditionnelles et les Instructions itératives.

8.1. Instructions conditionnelles

8.1.1. Instruction conditionnelle *if*

L'instruction if prend pour paramètre une expression booléenne placée obligatoirement entre parenthèses.

if *(expression) instruction;*

ou

if *(expression) {*
instruction 1;
instruction 2;
...
instruction n;
}

Exemple

```
	if(variable2 != 0) {
		double monResultat=(variable1/variable2);
	}
```

8.1.2. Instruction conditionnelle *else*

L'instruction else permet de compléter une instruction if lorsque la valeur retournée par l'expression est false.

```
if (expression) {
   bloc d'instructions 1;
}
else {
   bloc d'instructions 2;
}
```

Exemple :

```
      if(_variable2 != 0) {
         double resultat = ($variable1/_variable2);
      }else {
         System.out.print("La division par zéro n'est pas admise");
```

8.1.3. Les instructions conditionnelles imbriquées *If...else if*

Il est possible d'écrire plus simplement des imbrications successives d'instructions.

```
if (expression1) {
   bloc1;
}
else if (expression 2) {
   bloc2;
}
else if (expression3) {
   bloc3;
}
```

Exemple:

```
      if(_variable2 != 0) {
          double resultat = ($variable1/_variable2);
      }else if ($variable1 == _variable2) {
          System.out.print("Le resultat est toujours égale à 1");
      }else if(_variable2 == 0){
          System.out.print("La division par zéro n'est pas admise");
      }
```

Mais dans la pratique il n'est pas conseillé d'avoir un grand nombre d'imbrication de *else if*. Cela pourrait être une difficulté pour la lecture du code donc sa compréhension. L'instruction *switch* permet de palier à cette imbrication. Nous la verrons dans les titres suivants.

8.2. Instructions itératives

Les instructions itératives permettent d'exécuter plusieurs fois un bloc d'instructions, et ce, jusqu'à ce qu'une condition donnée soit fausse. Les trois types d'instruction itératives sont les suivantes :

8.2.1. L'instruction *for*

L'instruction ***for*** permet d'exécuter plusieurs fois la même série d'instructions : c'est une boucle. La syntaxe est la suivante :

for *(compteur; condition; modification du compteur) {*
liste d'instructions
}

Exemple :

```
for (int i = 0; i < n; i++) {
      System.out.println(i+1);
}
```

8.2.2. L'instruction while (Tant que…faire)

L'instruction ***while*** permet d'itérer un bloc d'instructions tant qu'une condition est vérifiée. Elle présente de plus l'avantage de tester la condition avant la première exécution. Il ne faut pas mettre de ; après la condition sinon le corps de la boucle ne sera jamais exécuté. La syntaxe est la suivante :

while *(expression) {*
instruction1;
instruction2;
…
instructionn;
}

Exemple :

```
int i=0;
while (i<15) {
      i++;
      System.out.println(i);
}
```

8.2.3. L'instruction *do while*

La boucle ***do while*** est une variante de la boucle ***while***, où la condition d'arrêt est testée après que les instructions aient été exécutées. Cette boucle est au moins exécutée une fois quelle que soit la valeur du booléen. La syntaxe est la suivante :

do *{*
liste d'instructions
} ***while*** *(condition réalisée);*

```
int i=0;
do {
	i++;
	System.out.println(i);
}while(i<15);
```

Exemple :

```
int i=0;
do {
	i++;
	System.out.println(i);
}while(i<15);
```

8.2.4. L'instruction *switch*

Switch/Case

L'instruction ***switch*** permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. On ne peut utiliser switch qu'avec des types primitifs d'une taille maximum de 32 bits (byte, short, int, char) et leurs wrappers, le type enum et la classe String. La syntaxe est la suivante :

switch *(variable) {*

case *valeur1 :*
Liste d'instructions
break;

case *valeur2 :*
Liste d'instructions
break;

case *valeurN... :*
Liste d'instructions
break;

default:
Liste d'instructions
}

Exemple :

```
public  enum Jour{
                Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche
        }
```

```
Jour jour = Jour.Samedi;

                switch (jour) {
            case Lundi:
                System.out.println("Jour de cours terrible");
                break;
            case Mardi:
                System.out.println("Jour de cours normal");
                break;
            case Mercredi:
                System.out.println("Jour de cours normal");
                break;
            case Jeudi:
                System.out.println("Jour de cours avec fatigue");
                break;
            case Vendredi:
                System.out.println("Jour de cours avec fatigue");
                break;
            case Samedi:
                System.out.println("Jour de cours relaxe");
                break;
            case Dimanche:
                System.out.println("Pas de cours. C'est le repos");
                break;
            default:
                System.out.println("Aucune activité");
                break;
        }
```

Le code précédent peut être réécrit de la manière suivante en tenant compte des traitements identiques dans certains cas :

```
Jour jour = Jour.Mercredi;

              switch (jour) {
            case Lundi:
               System.out.println("Jour de cours terrible");
               break;
            case Mardi, Mercredi:
               System.out.println("Jour de cours normal");
               break;
            case Jeudi, Vendredi:
               System.out.println("Jour de cours avec fatigue");
               break;
            case Samedi:
               System.out.println("Jour de cours relaxe");
               break;
            case Dimanche:
               System.out.println("Pas de cours. C'est le repos");
               break;
            default:
               System.out.println("Aucune activité");
               break;
        }
```

Ou avec une imbrication des cas (factorisation) :

```
Jour jour = Jour.Mercredi;

              switch (jour) {
            case Lundi:
               System.out.println("Jour de cours terrible");
               break;
            case Mardi:
            case Mercredi:
               System.out.println("Jour de cours normal");
               break;
            case Jeudi:
            case Vendredi:
               System.out.println("Jour de cours avec fatigue");
               break;
            case Samedi:
               System.out.println("Jour de cours relaxe");
               break;
            case Dimanche:
               System.out.println("Pas de cours. C'est le repos");
               break;
            default:
               System.out.println("Aucune activité");
               break;
        }
```

Pour ce même code on peut également fait usage des lambdas :

```
Jour jour = Jour.Mercredi;
              switch (jour) {
            case Lundi -> System.out.println("Jour de cours terrible");
            case Mardi, Mercredi -> System.out.println("Jour de cours normal");
            case Jeudi, Vendredi -> System.out.println("Jour de cours avec fatigue");
            case Samedi -> System.out.println("Jour de cours relaxe");
            case Dimanche -> System.out.println("Pas de cours. C'est le repos");
            default -> System.out.println("Aucune activité");
        }
```

Switch expressions

Une nouvelle fonctionnalité importante a été introduite en Java depuis le JDK 12 et a été améliorée dans la version 13 : Le switch expression :

```
Jour jour = Jour.Samedi;
              String action = switch (jour) {
            case Lundi -> "Jour de cours terrible";
            case Mardi, Mercredi -> "Jour de cours normal";
            case Jeudi, Vendredi -> "Jour de cours avec fatigue";
            case Samedi -> "Jour de cours relaxe";
            case Dimanche -> "Pas de cours. C'est le repos";
            default -> "Aucune activité";
        };
        System.out.println(action);
        }
```

9. Instructions break et continue

L'instruction break est utilisée pour sortir immédiatement d'un bloc d'instructions (sans traiter les instructions restantes dans ce bloc). Dans le cas d'une boucle on peut également utiliser l'instruction continue avec la différence suivante :

break : l'exécution se poursuit après la boucle (comme si la condition d'arrêt devenait vraie) ;

continue : l'exécution du bloc est arrêtée mais pas celle de la boucle. Une nouvelle itération du bloc commence si la condition d'arrêt est toujours vraie.

Exemple:

```
for (int i = 0, j = 0 ; i < 100 ; i++) {
        if (i > tab.length) {
        break ;
        }
        if (tab[i] == null) {
        continue ;
        }
```

```
        tab2[j] = tab[i];
        j++;
        }
```

10. Les tableaux

Un tableau est un ensemble indexé de données d'un même type. Un tableau n'est pas un objet ni un type simple. L'utilisation d'un tableau se décompose en trois parties : la création du tableau, le remplissage du tableau, lecture du tableau. Ils ne peuvent pas être redimensionnés une fois créés mais un tableau peut contenir aussi bien des objets que des types primitifs. Le premier élément d'un tableau possède l'indice 0.
La déclaration d'un tableau se faire sous la forme suivante :
Déclaration
char tableau[] ou char[] tableau

10.1. Création d'un tableau

Pour déclarer un tableau, on définit un type comme pour une variable de base, auquel on ajoute des crochets ouvrant et fermant [ ] pour indiquer qu'il s'agit d'un tableau. Sa taille est donnée à son instanciation.

Exemples :

```
int tableau[] = new int[50]; // déclaration et allocation

//OU
int[] tableau = new int[50];

//OU

int tab[]; // déclaration
tab = new int[50]; //allocation
```

La taille du tableau n'est pas obligatoire si le tableau est initialisé à sa création.

```
int tableau[] = {10,20,30,40,50};
```

Les tableaux existent en 1 et 2 dimensions. Les tableaux à 2 dimensions sont considérés comme des tableaux de tableaux.

```
float tableau[][] = new float[10][10]; //Tableau à 2 dimensions

int tableau[3][2] = {{5,1},{6,2},{7,3}}; //Tableau à 2 dimensions
```

10.2. Le parcours d'un tableau

Le parcours des tableaux se faire à l'aide de la boucle for. L'exemple suivant montre comment initialiser l'ensemble d'un tableau à l'aide d'une boucle for :

```
int[] monTableau = new int[ 10 ];
for ( int i = 0; i < monTableau.length; i++ ){
      monTableau[i] = i + 1;
}
```

La variable **length** retourne le nombre d'éléments du tableau.

monTableau[i] désigne le contenu du tableau à l'indice i.

Lorsque dans la boucle, on n'a pas besoin de connaître l'indice de chaque élément à traiter, on peut utiliser la syntaxe de type **for each** (depuis Java 5). L'exemple suivant montre comment afficher à l'écran l'ensemble des éléments du tableau d'entiers :

```
int monTableau [] = {35,45,10,56,11,6,80};
    for (int i : monTableau) {
          System.out.println(i);
    }
```

11. Les chaînes de caractères

11.1. Déclaration de la chaine de caractère

Les chaînes de caractères ne sont pas considérées en Java comme un type primitif ou comme un tableau. On utilise une classe particulière, nommée String, fournie dans le package java.lang. Les variables de type String ont les caractéristiques suivantes :

- leur valeur ne peut pas être modifiée
- on peut utiliser l'opérateur "+" pour concaténer deux chaînes de caractères. Il est possible d'utiliser l'opérateur "+=" :

```
String s1 = "hello";
String s2 = "world";
String s3 = s1 + " " + s2;
//Ou
String texte = " ";
      texte += " Hello ";
      texte += " ARSTM ";
```

- L'initialisation d'une chaîne de caractères s'écrit :

```
String s = new String(); //pour une chaine vide
String s2 = new String("hello world"); // pour une chaîne de valeur ”hello world”
```

11.2. Manipulation des chaines de caractère

La comparaison de deux chaines

La comparaison de deux chaines de caractères se fait avec la méthode equals(). Le résultat attendu de la comparaison est de type booléen : true ou false.

```
String texte1 = " texte 1 ";
      String texte2 = " texte 2 ";
      if ( texte1.equals(texte2) )
```

La longueur d'une chaine

La méthode **length()** permet de déterminer la longueur d'une chaine :

```
String texte = "ARSTM";
      longueurChaine = texte.length();// Taille = 5
```

La modification de la casse d'une chaine

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

```
String texte = " texte ";
      String textemaj = texte.toUpperCase();
```

Concaténation

L'opérateur de concaténation est le +, comme on peut s'y attendre. Là encore, le principe de non modification d'une chaîne de caractères s'applique. Examinons le code suivant.

```
String s1 =  "Bonjour" ;
String s2 =  "le monde !" ;
String s3 = s1 +  " " + s2 ;
```

Les trois operations de concatenation s'enchaînent. Une première chaîne de caractères est créée, concaténation de s1 et " ", puis une deuxième, concaténation de cette première chaîne et de s2.

Les classes StringBuffer et StringBuilder

StringBuilder et StringBuffer sont très similaires, la différence est que toutes les méthodes StringBuffer sont synchronisées, il est approprié lorsque vous travaillez avec une application multi-thread, car plusieurs threads peuvent accéder à un objet StringBuffer en même temps.

Pendant ce temps, StringBuilder a des méthodes similaires, mais elles ne sont pas synchronisées, mais comme ses performances sont plus élevées, vous devriez utiliser StringBuilder dans des applications à thread unique ou l'utiliser comme variable locale dans une méthode.

12. Le type Enumeration

Une énumération est un type de données particulier, dans lequel une variable ne peut prendre qu'un nombre restreint de valeurs. Ces valeurs sont des constantes nommées, par exemple MADAME, MADEMOISELLE, MONSIEUR.

Déclaration d'une énumération

La façon la plus simple de déclarer une énumération consiste à l'écrire lorsque l'on crée une variable, comme dans le code qui suit.

```
enum Civilite {MADAME, MADEMOISELLE, MONSIEUR} ;
    Civilite civilite = Civilite.MADAME ;
```

Déclarer une énumération de cette façon est possible, mais ne permet pas d'utiliser l'énumération Civilite ailleurs que dans la classe où elle est définie. Il est donc possible de déclarer une énumération dans un fichier séparé, comme une classe, en remplaçant simplement le mot-clé class par le mot-clé enum.

```
public enum Civilite {  // dans le fichier Civilite.java
    MADAME, MADEMOISELLE, MONSIEUR
}
```

Une énumération est en fait une classe, d'où cette appellation de classe énumération. Cette classe étend la classe Enum, qui elle-même étend la classe Object, comme toutes les classes en Java.

Les valeurs d'une énumération sont les seules instances possibles de cette classe. Dans notre exemple, Civilite comporte trois instances et trois seulement : MADAME, MADEMOISELLE, MONSIEUR. On peut donc comparer ces instances à l'aide d'un == de façon sûre, même si la comparaison à l'aide de la méthode equals() reste possible.

Chapitre 3 : LA PROGRAMMATION ORIENTE OBJET EN JAVA

1. Le concept d'objet et de classe

1.1. La classe

Une classe est le support de *l'encapsulation* : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des *méthodes*. 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. Une classe peut contenir des variables (primitives ou objets), des classes internes, des méthodes, des constructeurs, et des finaliseurs. La déclaration d'une classe se fait de la façon suivante :

[Modificateurs] class NomClasse
{
corps de la classe
}

1.2. Le constructeur

Un constructeur est une méthode automatiquement appelée au moment de la création de l'objet. Il est utile pour procéder à toutes les initialisations nécessaires lors de la création de la classe. Le constructeur porte le même nom que la classe et n'a pas de valeur de retour.

Toute classe possède au moins un constructeur. Si le programmeur ne l'écrit pas, il en existe un par défaut, sans paramètres, de code vide. Pour une même classe, il peut y avoir plusieurs constructeurs, de signatures différentes.

Exemple :

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

	//Constructeur avec le paramètre nom
	public Personne(String nom) {
		this.nom = nom;
	}

	//Constructeur avec les paramètres nom & prénom
	public Personne(String nom, String prenom) {
		this.nom = nom;
		this.prenom = prenom;
	}

	//Constructeur avec le paramètre age
	public Personne(int age) {
		this.age = age;
	}

}
```

L'appel de ces constructeurs est réalisé avec le **new** auquel on fait passer les paramètres. *Si le programmeur crée un constructeur, le constructeur par défaut n'est plus disponible.*

L'objet

Un objet n'est rien d'autre qu'une instanciation de la classe (moule de l'objet). Il contient des attributs et des méthodes. Cette instanciation se fait grâce à l'opérateur ***new***. La déclaration est de la forme *nom_de_classe nom_de_variable*. L'instanciation d'un objet se fait de la manière suivante : *MaClasse nom_objet = new MaClasse();*

Exemple :

```
StringBuffer monObjet = new StringBuffer();
```

Lorsqu'un objet est construit, les données qui le constitue sont placées dans un endroit de la mémoire et cela grâce à l'opérateur new qui fait appel à une méthode particulière de cet objet : **le constructeur**.

1.3. Les références et la comparaison d'objets

Les variables de type objet que l'on déclare ne contiennent pas un objet mais une référence vers cet objet. Lorsque l'on écrit c1 = c2 (c1 et c2 sont des objets), on copie la référence de l'objet c2 dans c1. c1 et c2 font référence au même objet : ils pointent sur le même objet. L'opérateur **==** compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

Exemple :

```
Rectangle r1 = new Rectangle(100,50);
      Rectangle r2 = new Rectangle(100,50);
      if (r1 == r1) { ... } // vrai
      if (r1 == r2) { ... } // faux
```

Le littéral **null** est utilisable partout où il est possible d'utiliser une référence à un objet. Il n'appartient pas à une classe mais il peut être utilisé à la place d'un objet de n'importe quel type ou comme paramètre. Le fait d'affecter null une variable référençant un objet pourra permettre au ramasse-miettes de libérer la mémoire allouée à l'objet si aucune autre référence n'existe encore sur lui.

2. Le mot-clé this

Le mot-clé this représente une référence sur l'objet courant, celui qui est en train d'exécuter la méthode dans laquelle se trouvent les instructions concernées.

this peut être utiliser :

- pour lever des ambiguïtés entre des noms d'attributs et des noms d'arguments de méthodes.

```
public class Calculateur{
    protected int valeur;

    public void calcule(int valeur)  {
        this.valeur = this.valeur + valeur;
    }
}
```

```
Rectangle(int origine_x, int origine_y) {
            this.origine_x = origine_x;
            this.origine_y = origine_y;
      }
```

- Pour référencer un constructeur depuis un autre constructeur.

3. Les modificateurs d'accès

En java, les modificateurs d'accès sont utilisés pour protéger l'accessibilité des variables et des méthodes. Ils ne peuvent pas être utilisés pour qualifier des variables locales.

Il existe 4 modificateurs qui peuvent être utilisés pour définir les attributs de visibilité des entités (classes, méthodes ou attributs) : **public**, **private**, **protected** et par defaut **(absence de modificateur)**. Leur utilisation permet de définir des niveaux de protection différents (présentés dans un ordre croissant de niveau de protection offert) :

Modificateur	Rôle
Public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets.
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 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 composants ne sont visibles qu'à l'intérieur de la classe. Ils ne peuvent être modifiés que par des méthodes définies dans la classe et prévues à cet effet. Les méthodes déclarées private ne peuvent pas être en même temps déclarées abstract car elles ne peuvent pas être redéfinies dans les classes filles.

Ces modificateurs d'accès sont mutuellement exclusifs.

3.1. Le mot-clé static

Le mot clé static s'applique aux variables et aux méthodes. Il permet de spécifier que la variable et la méthode sont respectivement une variable de classe et une méthode de classe.

Exemple:

```
public class Cercle {
static float pi = 3.1416f;
	float rayon;
	public Cercle(float rayon) {
		this.rayon = rayon;
		}
	public float surface() {
		return rayon * rayon * pi;
		}
	}
```

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

	}
```

Ces méthodes peuvent être utilisées sans instancier un objet de la classe. Les méthodes ainsi définies peuvent être appelées avec la notation classe.methode() au lieu de objet.methode().

3.2. Le mot-clé Final

Le mot clé **final** s'applique aux variables de classe ou d'instance ou locales, aux méthodes, aux paramètres d'une méthode et aux classes. Il permet de rendre l'entité sur laquelle il s'applique non modifiable une fois qu'elle est déclarée pour une méthode ou une classe et initialisée pour une variable. Une variable qualifiée de final signifie que la valeur de la variable ne peut plus être modifiée une fois que celle-ci est initialisée. Elle est utilisée pour déclarer les constantes en Java.

Exemple :

```
// Déclaration de constantes
   public final int constante = 28;
   public static final float PI = 3.141f;
```

Une méthode déclarée final ne peut pas être redéfinie dans une sous-classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous-classée.

3.3. Le mot clé abstract,

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

Abstract indique que la classe ne pourra être instanciée telle quelle. De plus, toutes les méthodes de cette classe abstract ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous-classes. Toutes les classes dérivées pourront profiter des méthodes héritées et n'auront à implémenter que les méthodes déclarées abstract.

3.4. Le mot-clé Synchronized

Il permet de gérer l'accès concurrent aux variables et méthodes lors de traitements de threads (exécution « simultanée » de plusieurs petites parties de code du programme)

3.5. Le mot clé volatile

Le mot clé volatile permet de signifier à la JVM qu'une variable sera modifiée par plusieurs Threads. En déclarant une variable volatile, sa valeur ne sera jamais placée dans le cache local à la Thread courante d'exécution. Chaque lecture et chaque écriture passeront forcément par la mémoire partagée entre les Threads. De fait, l'accès à la variable elle-même devient implicitement synchronisé

C'est une synchronisation implicite. Il garantit la synchronisation d'une variable dans un contexte multithread.

3.6. Le mot-clé native

Une méthode native est une méthode qui est implémentée dans un autre langage que Java. L'utilisation de ce type de méthode limite la portabilité du code mais permet une vitesse d'exécution plus rapide.

4. Les attributs

4.1. Variables de classe

Elles ne sont définies qu'une seule fois quel que soit le nombre d'objets instanciés de la classe. Leur déclaration est accompagnée du mot clé **static**.

Exemple :

```
static int maximumTemperature = 35 ;
```

4.2. Variable d'instance

Une variable d'instance est une variable qui existe pour chaque instance d'une classe, et qui contient une valeur relative à l'instance à laquelle elle appartient.

Exemple :

```
public class Rectangle {
	//Variables d'instances
	private int longueur ;
	private int largeur ;

	//Méthode pour le calcul de surface
	public int surface() {
	return this.longueur * this.largeur ;
```

```
        }
}
```

5. Les méthodes

5.1. Déclaration

Les méthodes sont des fonctions qui implémentent les traitements de la classe. La syntaxe de la déclaration d'une méthode est :

modificateurs type_retourne nom_méthode (arg1, ...) {

// définition des variables locales et du bloc d'instructions

}

Exemple :

```
//Méthode de calcul de surface du rectangle
public int surface() {
        int surface = this.largeur * this.largeur;
        return surface;
}
```

Les modificateurs de méthodes sont :

Modificateur	Rôle
Public	la méthode est accessible aux méthodes des autres classes
Private	l'usage de la méthode est réservé aux autres méthodes de la même classe
Protected	la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous-classes
Final	la méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
Static	la méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
Native	le code source de la méthode est écrit dans un autre langage

Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.

5.2. La surcharge de méthode

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments.

6. Le principe de l'encapsulation

En Java, comme dans beaucoup de langages orientés objet, les classes, les attributs et les méthodes bénéficient de niveaux d'accessibilité, qui indiquent dans quelles circonstances on peut accéder à ces éléments. L'encapsulation est la pratique consistant à regrouper des attributs au sein d'une même classe de telle sorte que ses attributs ne puissent pas être directement manipulés de l'extérieur de la classe, mais seulement indirectement par l'intermédiaire des méthodes (accesseurs et mutateurs).

Exemple :

```
package coursJava;

public class Voiture {
	private int compteur;
	private int imatriculation;
//Constructeurs
	public Voiture(int imatriculation) {
		this.imatriculation = imatriculation;
	}

	public Voiture(int compteur, int imatriculation) {
		this.compteur = compteur;
		this.imatriculation = imatriculation;
	}

//Méthodes d'accès aux attributs
	public int getCompteur() {
		return compteur;
	}

	public void setCompteur(int compteur) {
		this.compteur = compteur;
	}

	public int getImatriculation() {
		return imatriculation;
	}

	public void setImatriculation(int imatriculation) {
		this.imatriculation = imatriculation;
	}
}
```

Dans la classe voiture, les attributs sont de visibilité private. Les méthodes d'accès sont de visibilité publique. L'accès en lecture et en modification des valeurs des attributs par les autres classes se fera par ces méthodes.

7. L'héritage

7.1. Principe

L'héritage, est l'un des mécanismes les plus puissants de la programmation orientée objet. Il permet de reprendre des membres d'une classe (appelée superclasse ou classe mère) dans une autre classe (appelée sous-classe, classe fille ou encore classe dérivée), qui en hérite. C'est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :

- une classe mère ou super-classe
- une classe fille ou sous-classe qui hérite de sa classe mère

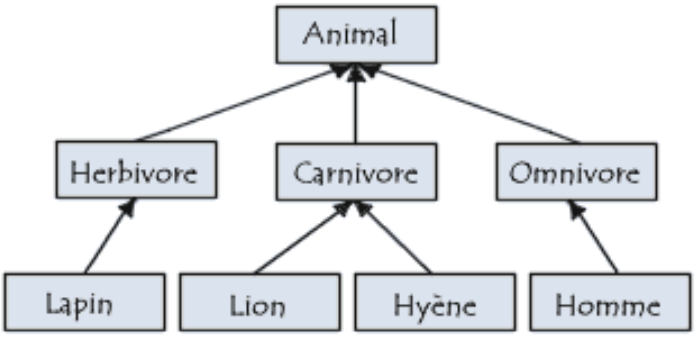


Illustration du principe d'héritage

L'idée principale de l'héritage est d'organiser les classes de manière hiérarchique. La relation d'héritage est unidirectionnelle et, si une classe B (classe fille) hérite d'une classe A (Classe mère), on dira que B est une sousclasse de A. Avec l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parente et peuvent les étendre.

On utilise le mot clé **extends** pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe mère.

La déclaration de l'héritage est le suivant : class Fille extends Mere { ... }

Exemple :

```
public class Avion extends Voiture {
	private int envergure;
	private int masseDecollage;

	private Avion(int envergure, int masseDecollage) {
```

```
            super();
            this.envergure = envergure;
            this.masseDecollage = masseDecollage;
        }
}
```

Au même titre que le mot-clé this permet de faire référence à l'objet en cours, le mot-clé super permet de désigner la superclasse, c'est-à-dire qu'à l'aide du mot-clé super, il est possible de manipuler les données membres et les méthodes de la superclasse.

Pour manipuler une propriété de la superclasse, il suffit d'utiliser la syntaxe suivante :

super.nom_de_la_propriete.

De la même façon, pour manipuler une méthode de la superclasse, il suffit d'utiliser la syntaxe suivante : *super.nom_de_la_methode().*

7.2. Redéfinition de méthode héritée

Lorsqu'une classe hérite de sa superclasse, elle hérite de ses méthodes, c'est-à-dire qu'elle possède les mêmes méthodes que sa superclasse.

La redéfinition d'une méthode consiste à réécrire totalement la méthode initiale. La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques). Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.

8. Le polymorphisme

Le nom de *polymorphisme* vient du grec et signifie *qui peut prendre plusieurs formes*. Cette caractéristique est un des concepts essentiels de la programmation orientée objet. Alors que l'héritage concerne les classes (et leur hiérarchie), le polymorphisme est relatif aux méthodes des objets.

Le polymorphisme qui signifie plusieurs formes, désigne le fait que l'objet puisse avoir plusieurs formes. Cette capacité lui vient directement du principe d'héritage. En effet comme vu précédemment un objet peut hériter des attributs et méthodes de ses parents. Mais un objet a aussi la possibilité de redéfinir une méthode en la réécrivant complètement ou juste en la complétant on appelle ça la surcharge. Survient alors le principe de polymorphisme : choisir en fonction du contexte la méthode parente adaptée. Le polymorphisme est donc le fait de pouvoir choisir la méthode qui correspond au type de l'objet en cours.

Par exemple prenons un objet Animal et ses classes filles poisson, tigre, panda, possédant tous une méthode Manger, le système appellera la méthode Manger adaptée en fonction du fait que ce soit l'objet tigre, panda ou poisson.

9. Les interfaces

Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite. C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclarées dans celle-ci. Plusieurs interfaces peuvent être implémentées dans une même classe.

- Toutes les méthodes d'une interface sont abstraites : elles sont implicitement déclarées comme telles.
- Les seules variables que l'on peut définir dans une interface sont des variables de classe qui doivent être constantes : elles sont donc implicitement déclarées avec le modificateur static et final même si elles sont définies avec d'autres modificateurs.

Les interfaces sont implicitement déclarées avec le modificateur abstract et se déclarent avec le mot clé **interface.**

```
public interface IDAO {

        public void addObject(Object object);

        public void updateObject(Object object);

        public void deleteObject(Object object);

        public List<Object> getObjects(Object object);

        public  List getObjects(String objet);

  }
```

Implémentation d'une interface s'effectue avec le mot-clé **implements.**

Exemple :

```
public class ImpDao implements IDao {

      @Override
      public void addObject(Object object) {
            //Bloc d'instruction ici
      }

      @Override
      public void updateObject(Object object) {
            //Bloc d'instruction ici
      }

      @Override
      public void deleteObject(Object object) {
            //Bloc d'instruction ici
```

```
	}

	@Override
	public List<Object> getObjects(Object object) {
		//Bloc d'instruction ici
		return null;
	}

	@Override
	public List getObjects(String objet) {
		//Bloc d'instruction ici
		return null;
	}
}
```

Ici la classe implémente l'interface et donne un corps aux différentes méthodes héritées.

Chapitre 4 : LA GESTION DES EXCEPTIONS

Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java. Il se compose d'objets représentant les erreurs et d'un ensemble de trois mots clés qui permettent de détecter et de traiter ces erreurs (try, catch et finally) mais aussi de les lever ou les propager (throw et throws). 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.

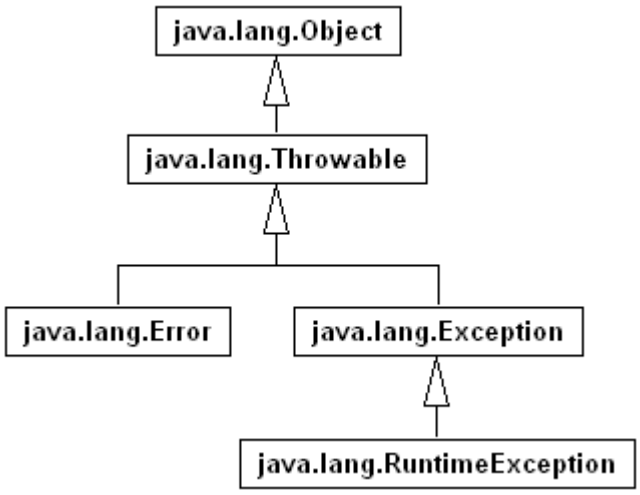
En Java, on distingue trois types d'erreurs, qui sont de degrés de gravité différents, à savoir :

- Les erreurs graves qui causent généralement l'arrêt du programme et qui sont représentées par la classe **java.lang.Error**.

- Les erreurs qui doivent généralement être traitées et qui sont représentées par la classe **java.lang.Exception**.

- Les erreurs qui peuvent ne pas être traitées et qui sont des objets de la classe **java.lang.RuntimeException** qui hérite de **java.lang.Exception**.

Toutes ces classes héritent directement ou indirectement de la classe **java.lang.Throwable**. Voici un petit diagramme récapitulatif de tout cela :



Exemple :

```
public void diviser() {
            int resultat = variable1 / variable2;
            System.out.println("Le resultat de la division est:"+resultat);
      }
```

Exception levée à la console

```
Resultat sortie en console
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at lesexeptions.TestExeption.diviser(TestExeption.java:8)
      at lesexeptions.TestExeption.main(TestExeption.java:19)
```

1. La classe Error

Cette classe est instanciée lorsqu'une erreur grave survient, c'est à dire une erreur empêchant la JVM de faire correctement son travail. Les objets de type **Error** ne sont pas destinés à être traités et il est même déconseillé de le faire. Un exemple récurrent est **java.lang.OutOfMemoryError** qui signifie que la machine virtuelle java ne dispose plus d'assez de mémoire pour pouvoir allouer des objets.

Exemple :

```
public class ErreurMemoire {
public static void main(String[] args) {
String[] tableau=new String[1000000000];
}
}
```

Les erreurs, lorsqu'elles surviennent, ont la particularité d'arrêter le thread en cours, sauf si elles sont traitées par un **catch**. Ainsi n'importe quel type de **Throwable** peut être "catché". Le code précédent deviendrait alors :

```
try {
 String[] tableau = new String[1000000000]; // OutOfMemoryError
} catch (Error e) {
System.out.println("Oups ! Une erreur est survenue : " + e);
}
System.out.println("Fin du programme");
```

2. La classe Exception

Les objets de type Exception ou bien de l'une de ses sous-classes sont instanciés lorsqu'une erreur au niveau applicatif survient. On dit, dans ce cas-là, qu'une exception est levée. Lorsqu'une exception est levée, elle se propage dans le code en ignorant les instructions qui suivent et si aucun traitement survient, elle débouche sur la sortie standard. Voici un bout de code illustrant cela :

```
public class PropagationException {
public static void main(String[] args) {
String chemin="/Un/chemin/vers/une/classe/qui/n'existe/pas";
Class.forName(chemin);//levée d'une ClassNotFoundException
System.out.println("fin du programme");
}
}
```

2.1. Traitement des exceptions

Les exceptions sont traitées via des blocs **try/catch** qui veulent littéralement dire essayer/attraper. On exécute les instructions susceptibles de lever une exception dans le bloc try et en cas d'erreur ce sont les instructions du bloc catch qui seront exécutées, pourvu qu'on attrape bien l'exception levée. Reprenons notre exemple de tout à l'heure et traitons l'exception. Ce qui donne le code suivant :

```
public class PropagationException {
public static void main(String[] args) {
try{
String chemin="/Un/chemin/vers/une/classe/qui/n'existe/pas";
Class.forName(chemin);//levée d'une ClassNotFoundException
System.out.println("fin du programme");
}catch(ClassNotFoundException ex){
System.out.println("Une exception est survenue");
}
}
}
```

On aurait pu simplement déclarer une exception de type Exception. Cela aurait pour effet d'attraper toutes les exceptions levées dans le bloc try car l'ensemble des exceptions déclarées dans la JDK hérite de cette classe (Exception). On peut également mettre plusieurs blocs catch qui se suivent afin de fournir un traitement spécifique pour chaque type d'exception. Cela doit être fait en respectant la hiérarchie des exceptions.

Exemple :

```
public class PropagationException {
 public static void main(String[] args) {
      try{
        String chemin="/Un/chemin/vers/une/classe/qui/n'existe/pas";
        Class.forName(chemin);//levée d'une ClassNotFoundException
         System.out.println("fin du programme");
      }catch(ClassNotFoundException ex){
        System.out.println("Une exception est survenue");
      }catch(Exception e){
       //traitement
       //pas d'erreur de compilation
      }
   }
}
```

2.2. La clause finally

Le mot clé **finally**, généralement associé à un **try**, permet l'exécution du code situé dans son bloc et ceci quel que soit la manière dont s'est déroulé l'exécution du bloc **try**.

```
public class PropagationException {
     public static void main(String[] args) {
          try{
              Object chaine="bonjour";
              Integer i=(Integer)chaine;//levée d'une ClassCastException
              System.out.println("fin du programme")
;
          }finally{
              System.out.println("on passe par le bloc finally");
             }
       }
}
```

3. La classe RuntimeException

Les exceptions héritant de **java.lang.RuntimeException** représentent des erreurs qui peuvent survenir lors de l'exécution du programme. Le compilateur n'oblige pas le programmeur ni à les traiter ni à les déclarer dans une clause **throws**. Les classes **java.lang.ArithmeticException** (qui peut survenir lors d'une division par 0 par exemple) et la classe

java.lang.ArrayIndexOutOfBoundException (qui survient lors d'un dépassement d'indice dans un tableau) sont des exemples de RuntimeException.

Exemple :

```
public class CompilationRuntimeException {
      public static void main(String[] args) {
          String[] tableau={"A","B","C"};
          for(int i=0;i<=3;i++){
             System.out.println(tableau[i]);
          }
      }
}
```

Chapitre 5 : Gestion des entrées/sorties simples

Le package **java.io** propose un ensemble de classes permettant de gérer la plupart des entrées/sorties d'un programme. Cette gestion consiste à créer un objet flux dans lequel transitent les données à envoyer ou à recevoir. Un flux connecte un objet Java à un autre élément..

1. Flux d'entrée

1.1. Lecture des entrées clavier

Les données provenant de l'utilisation du clavier sont transmises dans un flux d'entrée créé automatiquement pour toute application Java. On accède à ce flux par la variable statique de la classe **java.lang.System** qui s'appelle **in**. Ce flux est alors utilisé comme paramètre d'entrée du constructeur d'un autre flux d'entrée. Pour cet autre flux, on utilise généralement une sous-classe de Reader pour récupérer les entrées de l'utilisateur sous la forme d'une chaîne de caractères. La classe Clavier en donne un exemple :

```
import java.io.* ;
public class Clavier {
public static void main(String[] args) {
      try {
          BufferedReader flux = new BufferedReader(
          new InputStreamReader(System.in));
          System.out.print("Entrez votre prenom : ");
          String prenom = flux.readLine();
          System.out.println("Bonjour " + prenom);
          flux.close();
      } catch (IOException ioe) {
        System.err.println(ioe);
      }
    }
}
```

1.2. Lecture à partir d'un fichier

Un fichier est représenté par un objet de la classe **java.io.File**. Le constructeur de cette classe prend en paramètre d'entrée le chemin d'accès du fichier. Le flux d'entrée est alors créé à l'aide de la classe **FileInputStream** sur lequel on peut lire caractère par caractère grâce à la méthode **read()**. L'exemple suivant présente une méthode pour afficher le contenu d'un fichier :

Exemple :

```
import java.io.* ;
public class LectureFichier {
      public static void main(String[] args) {
            try {
               File fichier = new File(”monFichier.txt”);
               FileInputStream flux = new FileInputStream(fichier);
               int c ;
               while ((c = flux.read()) > -1) {
                    System.out.write(c);
               }
               flux.close();
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

Il arrive souvent d'enregistrer des données dans des fichiers textes. Il peut alors être utile d'utiliser un **BufferedReader** ou un **Scanner** pour effectuer la lecture.

2. Flux de sortie

Un flux de sortie est une instance d'une sous-classe de **OutputStream**. Comme pour les flux d'entrée, chaque classe de flux de sortie a son propre mode d'écriture de données. Les classes les plus couramment utilisées sont :

- **ByteArrayOutputStream** permet d'écrire des octets vers le flux de sortie ;
- **DataOutputStream** permet d'écrire des types de données primitifs de Java vers le flux de sortie.
- **FileOutputStream** est utilisé pour écrire dans un fichier. Les objets de cette classe sont souvent encapsulés dans un autre objet de classe OutputStream qui définit le format des données à écrire.
- **ObjectOutputStream** permet d'écrire des objets (c-à-d des instances de classes Java) vers le flux de sortie, si ces objets implémentent les interfaces Serializable ou Externalizable.
- Writer n'est pas une sous-classe de **OutputStream** mais représente un flux de sortie pour chaînes de caractères. Plusieurs sous-classes de Writer permettent la création de flux pour chaînes de caractères.

L'écriture de données vers un flux de sortie suit le même déroulement que la lecture d'un flux d'entrée :

1- **Ouverture du flux :** Elle se produit lors de la création d'un objet de la classe `OutputStream`.

2- **Ecriture de données :** Des données sont écrites vers le flux au moyen de la méthode `write()` ou d'une méthode équivalente. La méthode précise à employer dépend du type de flux ouvert.

3- **Fermeture du flux :** Quand le flux n'est plus nécessaire, il doit être fermé par la méthode **close()**.

2.1. Ecriture sur la sortie standard "écran"

Comme pour les entrées du clavier, l'écriture vers l'écran fait appel à la variable statique out de la classe System. On appelle généralement la méthode **System.out.print** ou **System.out.println** comme cela a été fait dans de nombreux exemples de ce livret.

2.2. Ecriture dans un fichier

L'écriture dans un fichier se fait par un flux de la classe **FileOutputStream** qui prend en entrée un fichier (instance de la classe File). Ce flux de sortie permet d'écrire des caractères dans le fichier grâce à la méthode **write()**. L'exemple suivant présente une méthode pour écrire un texte dans un fichier :

```
import java.io.* ;
public class EcritureFichier {
       public static void main(String[] args) {
              try {
                 File fichier = new File("monFichier.txt");
                 FileOutputStream flux = new FileOutputStream(fichier);
                 String texte = "Hello World!" ;
                 for (int i = 0 ; i < texte.length(); i++) {
                    flux.write(texte.charAt(i));
              }
                 flux.close();
              } catch (FileNotFoundException e) {
                 e.printStackTrace();
              } catch (IOException e) {
                 e.printStackTrace();
              }
       }
}
```

On peut également utiliser un Writer comme un **FileWriter** pour écrire des chaînes de caractères dans un fichier assez simplement.

2.3. Ecriture d'objets

Le flux de sortie peut également être encapsulé dans un flux de type **ObjectOutputStream**, comme le montre l'exemple suivant pour écrire la date courante dans un fichier :

```
import java.io.* ;
import java.util.Date ;
public class EcritureDate {
    public static void main(String[] args) {
        try {
           File fichier = new File("monFichier.dat");
           ObjectOutputStream flux = new ObjectOutputStream(
           new FileOutputStream(fichier));
           flux.writeObject(new Date());
           flux.close();
        } catch (IOException ioe) {
           System.err.println(ioe);
     }
   }
}
```

Ce fichier peut ensuite être lu par la classe LectureDate.

Remarque importante : Nous avons présenté dans ce chapitre plusieurs exemples d'entrée/sortie utilisant différents modes de lecture/écriture (avec ou sans flux). Nous conseillons toutefois d'utiliser en *priorité* un **Scanner** pour la lecture dans un fichier, et un **FileWriter** pour l'écriture dans un fichier.

Chapitre 6 : Les collections

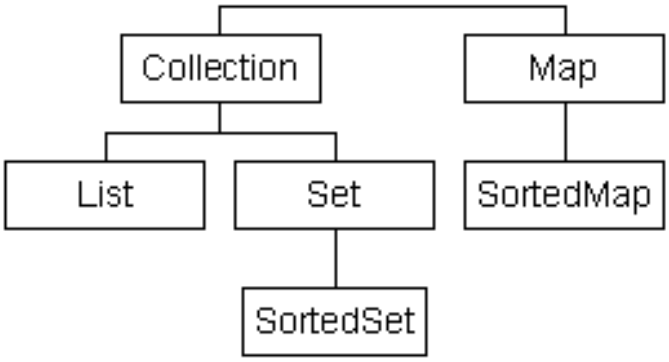
L'API Collections propose un ensemble d'interfaces et de classes dont le but est de stocker de multiples objets. Elle propose quatre grandes familles de collections, chacune définie par une interface de base :

- List : collection d'éléments ordonnés qui accepte les doublons
- Set : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons
- Map : collection sous la forme d'une association de paires clé/valeur
- Queue Dequeue: collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement

La bibliothèque standard Java fournit un certain nombre de collections dans ce qui s'appelle le Java Collections Framework (JCF). Tout son contenu se trouve dans le paquetage java.util.

1. Les interfaces des collections

Le framework de Java définit 6 interfaces en relation directe avec les collections qui sont regroupées dans deux arborescences :



L'interface Collection sert de super-interface commune aux listes (interface L**ist**) et à leurs variantes, ainsi qu'aux ensembles (interface **Set**).

Comme nous l'avons vu un peu plus haut, les listes sont ordonnées mais les ensembles ne le sont pas. Dès lors, seules les méthodes qui ne dépendent pas d'une notion d'ordre sont définies dans l'interface Collection.

L'interface Collection définit plusieurs méthodes :

Méthode	Rôle

boolean add(E e)	Ajouter un élément à la collection (optionnelle)
boolean addAll(Collection<?extends E> c)	Ajouter tous les éléments de la collection fournie en paramètre dans la collection (optionnelle)
void clear()	Supprimer tous les éléments de la collection (optionnelle)
boolean contains(Object o)	Retourner un booléen qui précise si l'élément est présent dans la collection
boolean containsAll(Collection<?> c)	Retourner un booléen qui précise si tous les éléments fournis en paramètres sont présents dans la collection
boolean equals(Object o)	Vérifier l'égalité avec la collection fournie en paramètre
int hashCode()	Retourner la valeur de hachage de la collection
boolean isEmpty()	Retourner un booléen qui précise si la collection est vide
Iterator<E> iterator()	Retourner un Iterator qui permet le parcours des éléments de la collection
boolean remove(Object o)	Supprimer un élément de la collection s'il est présent (optionnelle)
boolean removeAll(Collection<?> c)	Supprimer tous les éléments fournis en paramètres de la collection s'ils sont présents (optionnelle)
boolean retainAll(Collection<?> c)	Ne laisser dans la collection que les éléments fournis en paramètres : les autres éléments sont supprimés (optionnelle). Elle renvoie un booléen qui précise si le contenu de la collection a été modifié
nt size()	Retourner le nombre d'éléments contenus dans la collection
Object[] toArray()	Retourner un tableau contenant tous les éléments de la collection
<T> T[] toArray(T[] a)	Retourner un tableau typé de tous les éléments de la collection

2. L'interface Iterator

Cette interface définit des méthodes pour des objets capables de parcourir les données d'une collection. La définition de cette nouvelle interface par rapport à l'interface **Enumeration** a été justifiée par l'ajout de la fonctionnalité de suppression et la réduction des noms de méthodes.

Méthode	Méthode Rôle
boolean hasNext()	Indiquer s'il reste au moins un élément à parcourir dans la collection
Object next()	Renvoyer le prochain élément dans la collection
void remove()	Supprimer le dernier élément parcouru

3. Les collections de type List : les listes

Une liste (list) est une séquence ordonnée et dynamique (donc à taille variable) d'objets. Les listes sont très similaires aux tableaux, au point que la différence entre les deux est souvent

floue. Toutefois, les tableaux sont généralement de taille fixe et à accès aléatoire tandis que les listes sont de taille variable et à accès séquentiel.

3.1. L'interface List

Une collection de type List permet :

- de contenir des doublons
- d'interagir avec un élément de la collection en utilisant sa position
- d'insérer des éléments null

Pour les listes, une interface particulière est définie pour permettre le parcours dans les deux sens de la liste et réaliser des mises à jour : l'interface **ListIterator**

L'interface List définit plusieurs méthodes qui permettent un accès aux éléments de la liste à partir d'un index, de gérer les éléments, de rechercher la position d'un élément, d'obtenir une liste partielle (sublist) et d'obtenir des Iterator :

Constructeur	Rôle
void add(int index, E e)	Ajouter un élément à la position fournie en paramètre
boolean addAll(int index, Collection<? extends E> c)	Ajouter des éléments à la position fournie en paramètre
E get(int index)	Retourner l'élément à la position fournie en paramètre
int indexOf(Object o)	Retourner la première position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
int lastIndexOf(Object o)	Retourner la dernière position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
ListIterator<E> listIterator()	Renvoyer un Iterator positionné sur le premier élément de la liste
ListIterator<E> listIterator(int indx)	Renvoyer un Iterator positionné sur l'élément dont l'index est fourni en paramètre
E remove(int index)	Supprimer l'élément à la position fournie en paramètre
E set(int index, E e)	Remplacer l'élément à la position fournie en paramètre
List<E> subList(int fromIndex, int toIndex)	Obtenir une liste partielle de la collection contenant les éléments compris entre les index fromIndex inclus et toIndex exclus fournis en paramètres

3.2. La classe ArrayList

La classe ArrayList est l'implémentation la plus simple de l'interface List. Elle présente plusieurs caractéristiques :

- elle n'est pas thread-safe

- elle utilise un tableau pour stocker ses éléments : le premier élément de la collection possède l'index 0
- l'accès à un élément se fait grâce à son index
- elle implémente toutes les méthodes de l'interface List
- elle autorise l'ajout d'éléments null

La classe ArrayList dispose de plusieurs constructeurs :

Constructeur	Rôle
ArrayList()	Créer une instance vide de la collection avec une capacité initiale de 10
ArrayList(Collection<? extends E> c)	Créer une instance contenant les éléments de la collection fournie en paramètre dans l'ordre obtenu en utilisant son iterator
ArrayList(int initialCapacity)	Créer une instance vide de la collection avec la capacité initiale fournie en paramètre

Elle définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
boolean add(Object)	Ajouter un élément à la fin du tableau
boolean addAll(Collection)	Ajouter tous les éléments de la collection fournie en paramètre à la fin du tableau
boolean addAll(int, Collection)	Ajouter tous les éléments de la collection fournie en paramètre dans la collection à partir de la position précisée
void clear()	Supprimer tous les éléments du tableau
void ensureCapacity(int)	Augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
Object get(index)	Renvoyer l'élément du tableau dont la position est précisée
int indexOf(Object)	Renvoyer la position de la première occurrence de l'élément fourni en paramètre
boolean isEmpty()	Indiquer si le tableau est vide
int lastIndexOf(Object)	Renvoyer la position de la dernière occurrence de l'élément fourni en paramètre
Object remove(int)	Supprimer dans le tableau l'élément fourni en paramètre
void removeRange(int, int)	Supprimer tous les éléments du tableau de la première position fournie incluse jusqu'à la dernière position fournie exclue
Object set(int, Object)	Remplacer l'élément à la position indiquée par celui fourni en paramètre
int size()	Renvoyer le nombre d'éléments du tableau
void trimToSize()	Ajuster la capacité du tableau sur sa taille actuelle

3.3. Les listes chaînées : la classe LinkedList

La classe LinkedList, ajoutée à Java 1.2, est une implémentation d'une liste doublement chaînée dans laquelle les éléments de la collection sont reliés par des pointeurs. La suppression ou l'ajout d'un élément se fait simplement en modifiant des pointeurs.

Elle présente plusieurs caractéristiques :

- elle n'a pas besoin d'être redimensionnée quel que soit le nombre d'éléments qu'elle contient
- elle permet l'ajout d'un élément null.

Il existe plusieurs différences entre une ArrayList et une LinkedList :

une ArrayList stocke ses éléments en interne dans un tableau à taille fixe alors qu'une LinkedList stocke ses éléments dans une liste doublement chaînée :

- une ArrayList permet un accès direct à un élément alors qu'une LinkedList doit parcourir ses éléments pour obtenir celui désiré, ce qui est particulièrement contre performant

- le coût de variation de la capacité d'une collection de type ArrayList est important car il implique une copie du tableau de stockage interne de ses éléments

- l'ajout d'un élément en début ou en fin d'une collection de type LinkedList est particulièrement performant et son temps d'exécution est constant dans le temps (LinkedList implémente aussi l'interface Deque)

La classe LinkedList possède plusieurs constructeurs :

Constructeur	Rôle
LinkedList()	Créer une nouvelle instance vide
LinkedList(Collection<? Extends E> c)	Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre triés dans l'ordre obtenu par son Iterator

Plusieurs méthodes pour ajouter, supprimer ou obtenir le premier ou le dernier élément de la liste permettent d'utiliser cette classe pour gérer une pile ou une file :

Méthode	Rôle
void addFirst(Object)	Insèrer l'objet au début de la liste
void addLast(Object)	Insèrer l'objet à la fin de la liste
Object getFirst()	Renvoyer le premier élément de la liste
Object getLast()	Renvoyer le dernier élément de la liste
Object removeFirst()	Supprimer le premier élément de la liste et renvoie l'élément qui est devenu le premier
Object removeLast()	Supprimer le dernier élément de la liste et renvoie l'élément qui est devenu le dernier

3.4. L'interface ListIterator

L'interface ListIterator définit des fonctionnalités d'un Iterator permettant aussi le parcours en sens inverse de la collection, l'ajout d'un élément ou la modification du courant.

En plus des méthodes définies dans l'interface Iterator dont elle hérite, l'interface ListIterator définit plusieurs méthodes :

Méthode	Rôle

void add(E e)	Ajouter un élément dans la collection
boolean hasPrevious()	Retourner true si l'élément courant possède un élément précédent
int nextIndex()	Retourner l'index de l'élément qui serait retourné en invoquant la méthode next()
E previous()	Retourner l'élément précédent dans la liste
int previousIndex()	Retourner l'index de l'élément qui serait retourné en invoquant la méthode previous()
void set(E e)	Remplacer l'élément courant par celui fourni en paramètre

4. Les collections de type Set : les ensembles

Une collection de type Set ne permet pas l'ajout de doublons ni l'accès direct à un élément de la collection. Les fonctionnalités de base de ce type de collection sont définies dans l'interface **java.util.Set**. L'interface Set possède deux interfaces filles : SortedSet et NavigableSet.

Set hérite donc de Collection, mais n'autorise pas la duplication. SortedSet est un Set trié.

L'API Collections propose plusieurs implémentations de l'interface Set: HashSet, TreeSet, CopyOnWriteArraySet, EnumSet, LinkedHashSet, ConcurrentSkipListSet. (Voir la documemmentation sur les set).

5. Les collections de type Map : les associations de type clé/valeur

Les collections de type Map sont définies et implémentées comme des dictionnaires sous la forme d'associations de paires de type clés/valeurs. La clé doit être unique. En revanche, la même valeur peut être associée à plusieurs clés différentes. Un objet de type Map permet de lier un objet avec une clé qui peut être un type primitif ou un autre objet. Il est ainsi possible d'obtenir un objet à partir de sa clé.

L'interface Map possède plusieurs interfaces filles : SortedMap, NavigableMap, ConcurrentMap et ConcurrentNavigableMap. (Voir la documemmentation sur les Map).

6. Les collections de type Queue : les files

Une Queue est une collection qui stocke des éléments dans un certain ordre avant d'être consommés pour être traités. La plupart des implémentations proposées par le framework Collection utilise l'ordre FIFO (First In, First Out) mais l'ordre peut être différent.

Une pile (stack) est une liste dans laquelle les éléments sont toujours insérés ou supprimés de la même extrémité, appelée le sommet (top) de la pile. Une queue (queue) est une liste dans laquelle les éléments sont toujours ajoutés à une extrémité et retirés de l'autre extrémité.

Une « **deque** » (néologisme anglais tiré de double-ended queue) est une généralisation d'une queue dans laquelle les éléments peuvent être ajoutés ou supprimés à n'importe laquelle des deux extrémités mais nulle part ailleurs.

Parcours des

7. Parcours des collections

Il est très fréquent de devoir parcourir les éléments d'une collection. Comment faire ?

Ce parcours peut se faire avec une boucle for particulière (*for each*) et aussi avec un itérateur.

7.1. Parcours par boucle *for-each*

Les listes — et d'autres collections — peuvent heureusement être parcourues au moyen de la boucle **for-each**. Voici un exemple de parcours de list ci-dessous :

```
public void parcourir() {
             for (String s: maListe)
             System.out.println(s) ;
      }
```

7.2. Parcours par itérateur

Un itérateur (iterator) ou curseur (cursor) est un objet qui désigne un élément d'une collection. Un itérateur permet d'une part d'obtenir l'élément qu'il désigne, et sait d'autre part se déplacer efficacement sur l'élément suivant et parfois précédent de la collection.

Dans la bibliothèque Java, le concept d'itérateur est décrit par l'interface générique **Iterator**. Son paramètre de type représente le type des éléments de la collection parcourue par l'itérateur. L'interface Iterator est très simple et ne contient que trois méthodes :

- **boolean hasNext()** : retourne vrai si et seulement si il reste au moins un élément à parcourir.

- **E next()** : retourne l'élément suivant et avance l'itérateur sur son successeur, ou lève une exception s'il ne reste plus d'éléments.

- **void remove()** : supprime le dernier élément retourné par next, ou lève une exception si next n'a pas encore été appelée, ou si remove a déjà été appelée une fois depuis le dernier appel à next.

Voici un exemple de parcours de collection avec un itérateur :

```
public void pacourirIterateur() {
             Iterator<String> i = maListe.iterator();
             while (i.hasNext()) {
                    String s = i.next();
                    System.out.println(s);
                    i.remove();

      }
```